Introduction to Bidirectional Transformations

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

[1] Department of Computer Science, University of Oxford
firstname.lastname@cs.ox.ac.uk
[2] School of Informatics, University of Edinburgh
firstname.lastname@ed.ac.uk

Abstract. Bidirectional transformations (BX) serve to maintain consistency between different representations of related and often overlapping information, translating changes in one representation to the others. We present a brief introduction to the field, in order to provide some common background to the remainder of this volume, which constitutes the lecture notes from the *Summer School on Bidirectional Transformations*, held in Oxford in July 2016 as one of the closing activities of the UK EPSRC-funded project *A Theory of Least Change for Bidirectional Transformations*.

1 Introduction

Many tasks and problems in software engineering revolve around maintaining consistency between different representations of abstractly 'the same' underlying data in some system. Stable states of the system can be modelled by a relation, characterizing which states of the components of the system are considered 'consistent'. More interestingly, one also needs to resolve inconsistencies, modifying the states of one or more components in order to restore the system as a whole to a consistent compound state. In the general case, the compound system consists of multiple components; in this chapter, we will restrict attention to the simpler binary case, with just two components.

One may solve these problems from first principles, by providing separate programs that check for consistency, and that restore consistency in each possible direction—three programs, in the binary case. However, this approach is wasteful of effort, and presents a software maintenance challenge, because essentially the same information—the consistency relation—is duplicated in each of the separate programs. (Of course, redundancy might have some benefits too.) *Bidirectional transformations* (BX) attempt to eliminate the duplication, by arranging matters so that a single specification of the relationship between components may serve simultaneously to determine the consistency check and the various consistency restorers.

The history of BX may be traced back at least to the work of Bancilhon and Spyratos [8] in the 1980s on what has become called the *view–update problem* in databases. We say more about this motivating scenario in Section 2.2; but

in a nutshell, a complex database may provide a simplified *view* of a subset of the source data, and a user may reasonably want to specify an *update* of the source data in terms of the view. A richer variation of a similar need arises in model-driven development, when developers independently modify simpler projections of a composite system model, and expect their local modifications to be reflected in the shared composite. This particular variation has motivated two decades of work by Schürr and colleagues on *triple-graph grammars* [55], which provide grammars for consistent pairs of graphs linked by collections of triples; we discuss this approach in Section 3.4. More recently, Pierce and others [26] have spearheaded a fruitful line of work on *lenses*, programming abstractions for data references supporting 'get' and 'put' operations; we discuss this approach in more detail in Section 3.2.

This volume represents the lecture notes from a *Summer School on Bidirectional Transformations*, held in Oxford in July 2016 as one of the closing activities of the UK EPSRC-funded project *A Theory of Least Change for Bidirectional Transformations* (TLCBX). Our particular focus in the project was to investigate the so-called *principle of least change*, first identified by Meertens [44, 45]. One of the primary axioms that BX should satisfy formalises the idea that 'if nothing needs to change (because the overall system is already consistent), then nothing should be changed'. But if something does need to change, because consistency must be restored, then this axiom does not constrain the behaviour of the BX at all. There may be many different ways of restoring consistency, some better than others from the points of view of the users of the BX. A least-change principle attempts to formalise the intuitive idea that the BX should not change *more* than is necessary. However, providing a formal property that captures this intuition turns out to be a knottier problem than at first appears; we discuss it in Section 4.2.

The purpose of this chapter is to provide a brief introduction to the BX landscape, sufficient to allow the first-time visitor to find their way around the rest of this volume. In Section 2, we describe a number of motivating scenarios for BX. In Section 3, we sketch some of the main approaches that have been used to model and implement BX. Finally, in Section 4, we summarize some of the contributions made in the course of the TLCBX project: the *entangled state* monad, steps towards formalizing a *principle of least change*, and the fact that BX are *proof-relevant bisimulations*. In all cases, our aim is more to provide signposts to the important highlights than to present a complete study; we give appropriate references to the primary literature, where more details may be found. Complementary introductions to the field may be found in the Grace report [21], the report on a Dagstuhl seminar [32], and a chapter in the lecture notes from the Summer School on Generic and Indexed Programming [27].

2 Scenarios

2.1 Data conversions

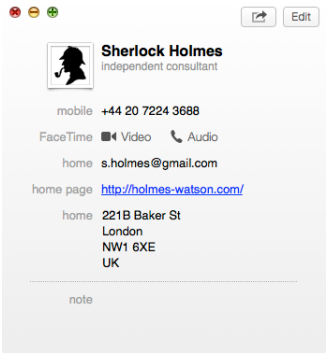
A very simple degenerate class of BX arises between two data sources when each maintains a faithful record of the whole overall state. The overall system may then be thought of as mere data conversion between different formats. Crucially, when one side is updated, the other may simply be discarded and recreated from scratch, with no loss of information.

Consider the example of an address book application, illustrated in Figure 1. The application maintains a collection of address cards; the graphical user interface of the application mediates between a textual representation of the cards and a more accessible pictorial representation. A card is typically edited via the pictorial representation, but stored on disc or exported in the textual representation. When the pictorial representation is edited and saved, the old textual representation is overwritten with a new one; conversely, if the textual representation is updated and reloaded, the old pictorial representation is replaced with a new one.

Naive approaches for designing graphical applications such as the address book entail two unidirectional transformations: a *parser*, which reads the textual representation and constructs the *view*; and a *pretty printer*, which writes the content of the view back to the textual representation. Higher-level form abstractions such as XForms [14], Windows Presentation Foundation [46], and Formlets [20] more or less successfully allow the application developer to express in one place the correspondence between the textual representation and its graphical layout.

```
BEGIN:VCARD
VERSION:3.0
N:Holmes;Sherlock;;;
FN:Sherlock Holmes
ORG:independent consultant;
EMAIL;type=HOME:s.holmes@gmail.com
TEL;type=VOICE:+44 20 7224 3688
ADR:;;221B Baker St;London;;NW1 6XE;UK
URL:http://holmes-watson.com/
PHOTO;ENCODING=b;TYPE=JPEG:/9j/4AAQSkZ
 ....................................
 iigAooooAKKKKACiiigAooooAKKKKAP//Z
X-ABUID:0772CAAE-D097B7BB4451:ABPerson
END:VCARD
```

(a)



(b)

Fig. 1. Data conversion: (a) vCard format and (b) an address book application

2.2 View–update

The primary historical precedent for the study of BX is the work starting with Bancilhon and Spyratos [8] already cited above, on the view–update problem in databases. Consider the three database tables shown in Figure 2: two source tables *Staff* and *Projects*, and a *View* table generated from them by the query

```
SELECT Name, Room, Role
FROM   Staff, Projects
WHERE  Name=Person
AND    Code="Plum"
```

The view–update problem is to translate an edit on the *View* table back to appropriate updates on the source tables *Staff* and *Projects*. Of course, the problem is not in general well posed; there may be multiple translations that would work (if one deletes Sam from *View*, should that entail moving Sam from the Plum project to Pear in *Projects*, or removing Sam from all projects?), or none (if one introduces a new person in *View*, what should their salary be in *Staff*?). There is a wealth of work on identifying and implementing the cases that do make sense [22].

Staff

Name	*Room*	*Salary*
Sam	314	£30k
Pat	159	£25k
Max	265	£25k

(a)

Projects

Code	*Person*	*Role*
Plum	Sam	Lead
Plum	Pat	Test
Pear	Pat	Lead

(b)

View

Name	*Room*	*Role*
Sam	314	Lead
Pat	159	Test

(c)

Fig. 2. The view–update problem: source tables (a) and (b), and a view (c)

2.3 Model-driven development

Model-driven development is fertile ground for BX, revolving as it does around models from different perspectives of a composite system design. Multiple developers, or the same developer wearing multiple hats, wish to work on individual models, focussing on the concerns at hand and ignoring those that are temporarily irrelevant. Having made some edits to one model, the other models should be updated to restore consistency.

Consider for example the issue of object–relational mapping (ORM). This is typically used when an application with business logic written in an object-oriented language should manipulate a data layer stored in a relational database. Rather than manually reading data from and writing it to the database, it is preferable to use some kind of tool support that allows the developer abstractly to specify the relationship between the two layers, with the actual transformations

back and forth being generated from this specification. There are a small number of well-understood ORM strategies [5], and also a good understanding of the challenges of ORM [48].

Figure 3(a) presents two meta-models. The left-hand metamodel states that classes have attributes, classes are in a super-/sub-class relationship, and attributes are in a next/previous relationship—the idea (not entirely captured in the metamodel) being that classes are grouped into single-inheritance hierarchies, and that the attributes of a class are linearly ordered. The right-hand metamodel states more simply that a table similarly has a sequence of columns. Figure 3(b) presents a class model, conforming to the class metamodel, with four classes arranged into two hierarchies. Figure 3(c) presents a table model, conforming to the table metamodel, following the 'one table per hierarchy' ORM strategy: one table named A corresponds to the hierarchy rooted at class A, the other table named D corresponds to the isolated class D. Note that neither model is definitive: the class model contains names for subclasses, which are lost in the table model; and the table model records a linear ordering on all the attributes in a hierarchy, which is only partially maintained in the class model. (This example is inspired by Schürr [56], and documented in the BX Examples Repository [7] that was established as an early step in the TLCBX project [18]; it will be revisited in Section 3.4.)

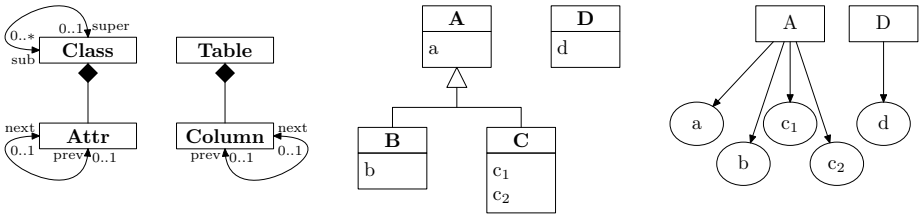


Fig. 3. (a) Two metamodels, (b) a class model, and (c) a table model

2.4 Composers

Composers [61] is a classical simple BX example that has been used by various authors over the years [12, 59] to illustrate BX concepts. In this example, there are two sets of models

$$M = \{\, \mathit{Name} \times \mathit{Dates} \times \mathit{Nationality} \,\}$$
$$N = [\, \mathit{Name} \times \mathit{Nationality} \,]$$

of a collection of musical composers. A model $m : M$ is a set of triples, recording the name, dates of birth and death, and nationality of each composer; a model n:

N is a sequence of pairs, recording only names and nationalities, but in some order. Models m and n are consistent if they have the same *set* of *Name* × *Nationality* pairs; for example:

$$\begin{aligned} m = \{\, &(\text{“Jean Sibelius”}, && 1865\text{–}1957, \text{Finnish}),\\ &(\text{“Aaron Copland”}, && 1910\text{–}1990, \text{American}),\\ &(\text{“Benjamin Britten”}, && 1913\text{–}1976, \text{English}) \,\} \end{aligned}$$

$$\begin{aligned} n = [\, &(\text{“Benjamin Britten”}, && \text{English}),\\ &(\text{“Aaron Copland”}, && \text{American}),\\ &(\text{“Jean Sibelius”}, && \text{Finnish}) \,] \end{aligned}$$

Again, neither model is definitive (model M lacks the ordering, whereas model N lacks the dates). Consequently, there is a variety of ways of restoring consistency: from M to N, one needs to worry about the ordering, and from N to M, one needs to worry about the dates.

3 Approaches

3.1 Relational

Stevens [59, 60] has pioneered a simple relational model of BX, in order to focus on the essence of the relationships between the model spaces and consistency restoration. According to this approach, a BX between model sets M, N is a triple $(R, \overrightarrow{R}, \overleftarrow{R})$ consisting of a *consistency relation* $R \subseteq M \times N$, a *forwards consistency restorer* $\overrightarrow{R} : M \times N \to N$, and a *backwards consistency restorer* $\overleftarrow{R}: M \times N \to M$. We may write $(R, \overrightarrow{R}, \overleftarrow{R}): M \nleftrightarrows N$. The idea is that given possibly inconsistent models m', n (arising perhaps from an originally consistent pair m, n in which m has been edited to m'), forwards consistency restoration yields $n' = \overrightarrow{R}(m', n)$ such that $R(m', n')$ holds; and symmetrically, given m, n', backwards consistency restoration yields $m' = \overleftarrow{R}(m, n')$ such that again $R(m', n')$ holds.

(To be complete, one ought also consider a distinguished ‘no information’ model in each model set, which is used as an argument to the consistency restorers when one model must be created ab initio from the other. But for simplicity, we will not discuss this further.)

We say that the BX is *correct* if consistency is indeed restored by the consistency restorers:

$$\forall m', n.\ R\ (m', \overrightarrow{R}(m', n))$$
$$\forall m, n'.\ R\ (\overleftarrow{R}(m, n'), n')$$

and *hippocratic* (‘do no harm’) if restoration does nothing when the models are already consistent:

$$\forall m, n.\ R(m, n) \Rightarrow \overrightarrow{R}(m, n) = n$$
$$\forall m, n.\ R(m, n) \Rightarrow \overleftarrow{R}(m, n) = m$$

In addition, the BX is *history-ignorant* (rather a strong condition) if

$$\forall m, m', n.\ \overrightarrow{R}(m, \overrightarrow{R}(m', n)) = \overrightarrow{R}(m, n)$$
$$\forall m, n, n'.\ \overleftarrow{R}(\overleftarrow{R}(m, n'), n) = \overleftarrow{R}(m, n)$$

—informally, a later consistency restoration completely overwrites an earlier one.

To illustrate, the Composers example from Section 2.4 would be represented by the model sets

$$M = \{\, \mathit{Name} \times \mathit{Dates} \times \mathit{Nationality} \,\}$$
$$N = [\, \mathit{Name} \times \mathit{Nationality} \,]$$

as before. The consistency relation R would be such that $R(m, n)$ holds precisely when the set of name–nationality pairs obtained by projecting away all the dates in m coincides with the set of name–nationality pairs obtained by taking the elements of the list n. For $\overrightarrow{R}(m, n)$ to be correct, the elements of the list produced are completely determined, but not the ordering, nor multiplicity in the case that there are two triples in m that share name and nationality. For the forwards restorer to be hippocratic, it suffices that the name–nationality pairs present in both m and n are returned in the same order as in n, with additional pairs placed anywhere. Conversely, for $\overleftarrow{R}(m, n)$ to be correct, the names and nationalities in the set produced are completely determined; for it to be hippocratic, it suffices for the name–nationality pairs in common retain the dates recorded in m, and any additional entries may have arbitrary dates. But it is hard to make the restorers history-ignorant; informally, this entails reconstructing discarded information. For example, with states m, n as in Section 2.4, and m', n' the corresponding states with Copland missing:

$$\begin{aligned} m' = \{\, &(\text{“Jean Sibelius”}, \quad 1865\text{–}1957, \text{Finnish}), \\ &(\text{“Benjamin Britten”}, 1913\text{–}1976, \text{English}) \,\} \end{aligned}$$
$$\begin{aligned} n' = [\, &(\text{“Benjamin Britten”}, \text{English}), \\ &(\text{“Jean Sibelius”}, \quad \text{Finnish}) \,] \end{aligned}$$

then for correctness' sake, $\overrightarrow{R}(m', n)$ must be a list omitting Copland, such as n'; and so $\overrightarrow{R}(m, n')$ must restore Copland to the list, in the same position as it was in n, without having access to that information. Conversely, $\overleftarrow{R}(m, n')$ must be a set omitting Copland, such as m'; and $\overleftarrow{R}(m', n')$ must somehow restore Copland's dates, without having access to those dates.

3.2 Lenses

BX notions were brought to the attention of the programming languages community principally through a series of papers [26, 13, 12, 9, 30, 31] by Pierce *et al.* on *lenses*. An asymmetric lens $(\mathit{get}, \mathit{put}) : S \nleftarrow V$ from source S to view V consists of two functions

$$get : S \to V$$
$$put : S \times V \to S$$

The idea is that $get\ s$ projects a view from source s, and $put\ (s, v')$ restores a modified view v' into existing source s. The lens can be seen as a reference to a V component 'inside' an S composite. It is 'asymmetric' because the source S is primary, determining the secondary view V (via the get function), but in general the view does not determine the source. (The full story involves also a $create : V \to S$ function, analogous to the 'no information' models in the relational approach.)

A simple example is given by projection from pairs: when $S = A \times B$ and $V = A$, with $get\ (a, b) = a$ extracting the first component of the pair, and $put\ ((a, b), a') = (a', b)$ updating the first component.

The lens is *well-behaved* if it satisfies

$$\forall s, v. \quad put\ (s, get\ s) \quad = s \qquad (\mathsf{GetPut})$$
$$\forall s, v. \quad get\ (put\ (s, v)) \quad = v \qquad (\mathsf{PutGet})$$

Informally, if one gets view $v = get\ s$ from source s then immediately puts it back again, s does not change; and having put view v into source s, it is indeed faithfully stored there, and will be retrieved by a subsequent get.

It is instructive to compare this approach with the relational one from Section 3.1. The consistency relationship R being maintained is

$$R(s, v) \Leftrightarrow (get\ s = v)$$

Forwards consistency restoration $\overrightarrow{R} : S \times V \to V$ is trivial, $\overrightarrow{R}(s, v) = get\ s$, because the source completely determines the view; backwards consistency restoration $\overleftarrow{R} = put : S \times V \to S$ is just the put function. Property (GetPut) is analogous to hippocracy (when reconciling view $v = get\ s$ with source s with which it is already consistent, do nothing); property (PutGet) is analogous to correctness (having reconciled s with v, the state is consistent).

A well-behaved asymmetric lens is *very well-behaved* if in addition it satisfies

$$\forall s, v, v'.\ put\ (put\ (s, v), v') = put\ (s, v') \quad (\mathsf{PutPut})$$

Informally, having put view v into source s, immediately putting another view v' will completely overwrite v, so that the net effect is the same as simply having put v' in the first place. The projection lens above is very well-behaved; indeed, it is a folklore result that any very well-behaved lens $S \nleftarrow V$ induces an isomorphism $S \simeq V \times C$ for some *complement* type C that is not touched by the put function—hence the term '*constant complement*' [8] is sometimes used.

It is this constant complement consequence that makes very well-behavedness or history ignorance such a strong property. For example, consider a simplified, asymmetric version of the Composers example from Section 2.4, with both state spaces being lists:

$$M = [Name \times Dates \times Nationality]$$
$$N = [Name \times Nationality]$$

so that M determines N, via a *get* function that projects away the dates. It is straightforward to define *put* to yield a well-behaved lens; but there is no definition of *put* that yields a very well-behaved lens, because the source type M does not factorize perfectly into $N \times C$ for any complement type C.

Hofmann *et al.* [30] introduced a symmetric variation of lenses, whereby lens $(putr, putl) : A \nleftrightarrow_C B$ between A and B with complements C consists of a pair of functions

$$putr : A \times C \to B \times C$$
$$putl : B \times C \to A \times C$$

Now neither A nor B determines the other; neither is definitive. Think of C as a record of the information in A that is missing from B, together with the information in B that is missing from A, so that $A \times C$ determines B and $B \times C$ determines A. One might simply take the complement to be $C = A \times B$, but usually the point of the exercise is that A and B have some information in common, and this common information need not be represented also in C.

Function *putr* reads the B-relevant part of C and updates the A-relevant part; dually, *putl* reads the A-relevant part and updates the B-relevant part. Thus, *putr* transfers information from left to right; it takes a modified left-hand value $a' : A$ and a complement $(c_A, c_B) : C$, and constructs an updated right-hand value $b' : B$ from a' and c_B, together with an updated complement (c'_A, c_B); and symmetrically for *putl*, from right to left.

A symmetric lens is *well-behaved* if it satisfies

$$\forall a, b, c, c'. \quad putr\ (a, c) = (b, c') \Rightarrow putl\ (b, c') = (a, c') \qquad \text{(PutRL)}$$
$$\forall a, b, c, c'. \quad putl\ (b, c) = (a, c') \Rightarrow putr\ (a, c') = (b, c') \qquad \text{(PutLR)}$$

These conditions induce *consistent* states (a, c, b) such that $putr\ (a, c) = (b, c)$ and $putl\ (b, c) = (a, c)$. A well-behaved symmetric lens is *very well-behaved* if in addition

$$\forall a, a', b, c, c'.\ putr\ (a, c) = (b, c') \Rightarrow putr\ (a', c') = putr\ (a', c) \qquad \text{(PutPutR)}$$
$$\forall a, b, b', c, c'.\ putl\ (b, c) = (a, c') \Rightarrow putl\ (b', c') = putl\ (b', c) \qquad \text{(PutPutL)}$$

(and as before, very well-behavedness is a very strong condition).

An asymmetric lens $S \nleftarrow V$ is effectively a special case $S \nleftrightarrow_S V$ of symmetric lenses: there is no information in V that is missing from S, so the complement just can be S, or more efficiently some smaller complement C such that $V \times C$ determines S. The Composers example is not representable as an asymmetric lens (because neither state space determines the other); but it is representable as a symmetric lens, with complement $C = [Name \times Dates \times Nationality]$ that records both the ordering absent from M and the dates absent from N.

3.3 Ordered, delta-based, categorical

The problem with put–put laws (history ignorance, very well-behavedness) is that they demand a strong property about *combining* two updates into one

update with the same overall effect. As we have seen, this is apparently too much to expect, at least in the case of combining two arbitrary updates. But perhaps it is more reasonable for certain special classes of updates?

Hegner [28] took this observation as the inspiration for a more nuanced look at what he called *ordered* updates. In this setting, the state spaces have a natural ordering, and certain updates are monotonic with respect to this ordering. For example, the states might be states of a database, modelled as sets of tuples, with the sets ordered by inclusion; insertion of some tuples into the set is monotonic with respect to inclusion. We have seen that combinations of deletions and insertions tend not to compose well—in particular, deletion of an item entails deletion also of any complementary information about that item from the system, and re-insertion of morally 'the same' item requires the complementary information somehow to be restored, if the net effect is to leave the system as it was. It is less drastic to insist on the special case that two consecutive insertions of small sets of tuples is equivalent to one insertion of the union of those sets.

An alternative perspective, argued first by Diskin [23, 24], is that the put–put problem arises from taking a *state-based* approach to BX (as exemplified by the relational and lens work described above). In this state-based approach, the consistency restoration operations are given only old and new states, and so the restoration process consists of two steps: *alignment*, to find out what has changed on one side (not to be confused with the problem of matching up models from different spaces to identify correspondences, which is also sometimes called 'alignment'), and *propagation*, to translate that change to the other side. A *delta-based* approach separates those two tasks; in particular, the input to consistency restoration is not just a new state a', the result of an update, but the update $\delta : a \mapsto a'$ itself, so the alignment information is provided as an input to consistency restoration, and no longer needs to be reconstructed during restoration.

The situation is as illustrated in Figure 4. Forwards propagation takes an initially consistent pair of states (a, b) and an update $\delta_A : a \mapsto a'$ on the A side, and yields an update $\delta_B : b \mapsto b'$ on the B side and a new consistent pair of states (a', b'). More concretely, one might want to maintain not merely the bare information that states a, b are consistent, but also the *correspondence* $c : a \leftrightarrow b$ that witnesses to their consistency; for example, when the states are sets of model elements, the correspondence c might be a set of triples, recording which A-elements are related to which B-elements, and by what relation. Symmetrically, backwards propagation takes $c : a \leftrightarrow b$ and $\delta_B : b \mapsto b'$ to $\delta_A : a \mapsto a'$ and $c' : a' \leftrightarrow b'$. One of the benefits of the delta-based approach is being able to relax the expectation that one can transit from any state to any other, as is implicit in the state-based approach.

Johnson, Rosebrugh and others have been pioneering a line of work [33, 36, 34, 35] to provide a categorical unification and generalization of the ordered and delta-based approaches. In their approach, one represents a state space A and its transitions $\delta : a \mapsto a'$ as a category $\mathcal{A}$. The arrows in the category represent

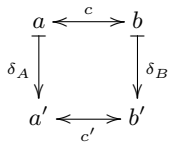


Fig. 4. Delta-based consistency

the allowable transitions; as with the delta-based approach, one need not allow all possible transitions.

The relevant constructions involve the objects $|\mathcal{A}|$ of a category $\mathcal{A}$, the set $|\mathcal{A}^2|$ of arrows of $\mathcal{A}$, and the comma category $G/\mathcal{B}$ for a functor $G : \mathcal{A} \to \mathcal{B}$, which has objects (A, β) where A is an object of $\mathcal{A}$ and $\beta : G(A) \to B$ is an arrow of $\mathcal{B}$. It would take us too far out of our way here to present a complete tutorial in category theory sufficient to provide much intuition for these constructions; but one would not go far wrong in thinking of the category as a directed graph, its objects (the states) as vertices in the graph, and its arrows (the allowable transitions between states) as paths in the graph.

An (asymmetric, delta-) lens $(G, P) : \mathcal{A} \rightleftharpoons \mathcal{B}$ is then a pair in which $G : \mathcal{A} \to \mathcal{B}$ is a functor, and $P : |G/\mathcal{B}| \to |\mathcal{A}^2|$ is a function, taking a pair $(A, \beta : G(A) \to B)$ to a transition $\alpha : A \to A'$. One should think of the pair $(A, \beta : G(A) \to B)$ as a transition in $\mathcal{B}$ from an initial state $G(A)$ that corresponds to a given state A in $\mathcal{A}$.

The lens is *well-behaved* if it satisfies the first three of the following four properties, for all $\beta : G(A) \to B$ and $\beta' : G(A') \to B'$, and *very well-behaved* if it satisfies all four:

- the domain of $P(A, \beta)$ is A;
- $P(A, id_{G(A)}) = id_A$;
- $G(P(A, \beta)) = \beta$;
- $P(A, \beta' \cdot \beta) = P(A', \beta') \cdot P(A, \beta)$, where A' is the codomain of $P(A, \beta)$ and so $G(A') = B$.

The first says that when $\beta : G(A) \to B$ is propagated back by P, it does indeed yield an allowable $\mathcal{A}$-transition from A; this is a basic requirement for coherence, when not all transitions are allowed from every state. The second says that the identity transition in $\mathcal{B}$ propagates back to the identity transition in $\mathcal{A}$; this is analogous to hippocracy. The third says that the $\mathcal{B}$-transition α is faithfully propagated back to an $\mathcal{A}$ transition; this is analogous to correctness. And the fourth says propagating the composite arrow $\beta' \cdot \beta$ is equivalent to propagating β' after β. The fourth property is analogous to history-ignorance, although the term no longer seems adequate; crucially, because one may focus attention only on certain compatible sets of transitions, it is no longer an unreasonably strong condition.

One can recover the set-based approach via the *codiscrete* category, which has precisely one arrow between any pair of objects, representing the fact that

a transition is available from any state to any other state. And one can recover the ordered approach by considering the ordered set as a category, with at most one arrow between any pair of objects. Symmetric lenses arise from *spans* of asymmetric lenses [34]; for example, a symmetric lens between $\mathcal{A}$ and $\mathcal{B}$ can be constructed from two asymmetric lenses $\mathcal{C} \nleftarrow \mathcal{A}$ and $\mathcal{C} \nrightarrow \mathcal{B}$ from some common source $\mathcal{C}$ representing the 'union' of the information provided by $\mathcal{A}$ and $\mathcal{B}$.

3.4 Triple-graph grammars

The *triple-graph grammar* approach to BX arose by combining the work of Rozenberg, Ehrig and others in *graph grammars, graph rewriting, and graph transformations* [54, 25] with earlier ideas from Pratt on *pair grammars* [53]. A grammar specifies a language, and can be used both to determine whether a term is in that language, and also to generate terms from that language. A pair grammar consists of a pair of grammars, whose rules and non-terminals are paired in a correspondence that models a translation between the two languages. *Triple-graphs* are a special kind of graph, with both 'object-level' vertices and edges as usual, but also labelled 'meta-level' edges that link object-level entities. These labelled meta-level edges are the *triples* of the name; they provide the *correspondence* structure relating two object-level graphs.

Consider the object–relational mapping example from Section 2.3 [7]. A class model and a table model are consistent if there is an appropriate correspondence relationship between their model elements, as illustrated in Figure 5. Here, the two models are as shown in Figure 3(b,c). The correspondence is given by the set of broken edges linking the model elements: dotted lines for the CT correspondences between class and table elements, and dashed lines for the AC correspondences between attribute and column elements. The triple-graph itself conforms to the metamodel shown in Figure 6, which simply consists of the union of the two metamodels from Figure 3(a) together with the two correspondence associations.

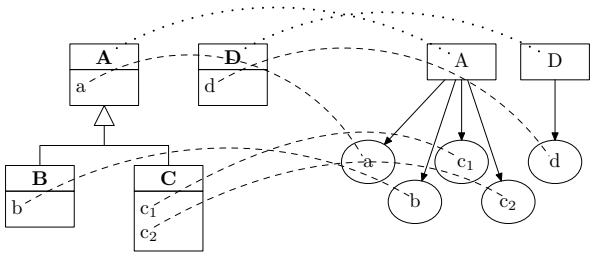


Fig. 5. Two models, with correspondences

This triple of metamodels could be used to derive a BX as follows. Forwards transformation takes a class model, analyses it according to the class metamodel, then uses the associated correspondences to generate a corresponding

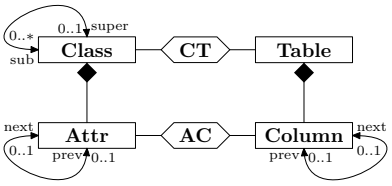


Fig. 6. Two metamodels, with correspondences

table model. Conversely, backwards propagation constructs a class model by analysing a table model.

Experience suggests that although this constraint-based process works in principle, in practice it is highly non-deterministic, and therefore difficult to use with predictable results. One therefore works with production rules, as in traditional grammars, rather than metamodels, in order to gain more control over the non-determinism. A suitable collection of rules for the object–relational example is shown in Figure 7. The idea is that the black items match against existing model elements, and then the green items labelled with ++ specify which new model elements are to be introduced; moreover, the elements crossed out in red are required not to exist for the rule to be applicable ('negative application conditions').

Thus, Rule 1 says that one can introduce a new *Class*, linked to a new *Table*; this starts a new hierarchy. Rule 2 says that when there exists a *Class* linked to a particular *Table*, one may introduce a new *Class* as a subclass, and link it to the same *Table*; this adds a new class to an existing hierarchy. Rule 3 says that if there is a *Class* with no *Attr*, linked to a *Table* with no *Column*, then one may introduce a new *Attr* for the *Class* linked to a *Column* for the *Table*; this introduces a first attribute into a hierarchy. Rule 4 says that if there is a *Class* with no *Attr*, linked to a *Table* that has a *Column* that is previous to no other *Column*, then one may introduce a new *Attr* for the class and *Column* for the *Table*; this *Class* will presumably be a subclass of some other *Class*, and the existing *Column*s will correspond to *Attr*s of other *Class*es in the same hierarchy; and by construction, the new *Column* will be introduced as the last one in the *Table*. Finally, Rule 5 says that if there is a *Class* with an *Attr* that is previous to no other *Attr*, linked to a *Table* with a *Column* that is previous to no other *Column*, then one may introduce a new *Attr* for the *Class* and *Column* for the *Table*; this will be the last attribute in the class and the last column in the table.

Note how Rules 4 and 5 cut down the non-determinism by ensuring that new entries for sequences are added at the end of the sequence. However, the process is not completely deterministic; there is nothing to specify the order in which class hierarchies are explored, except that parents must precede children, and nothing to specify the relative ordering of attributes, except that they must agree with the ordering within an individual class. Moreover, each rule is monotonic,

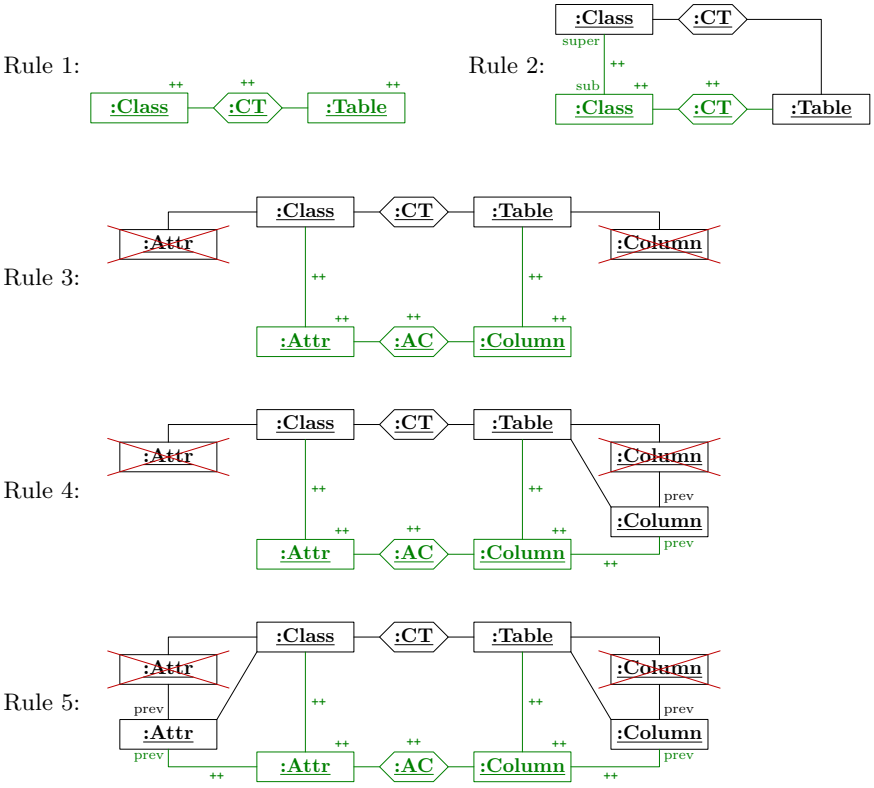


Fig. 7. A triple-graph grammar

creating new elements without deleting anything; this is relevant for turning the matching process into an efficient graph translation algorithm.

4 Contributions

In this section we summarise the main contributions of our recent research on bidirectional transformations:

- *entangled state monads*, which can be used to provide a principled foundation for *bidirectional transformations with effects* [1],
- steps towards formalizing a *principle of least change*, and
- the fact that BX are *proof-relevant bisimulations*.

4.1 Entangled state and monadic bidirectional transformations

Since the pioneering work of Moggi [47], *monads* have been explored extensively to provide semantics for *computational effects*, such as exceptions, mutable state, and I/O. Computational effects are a particular challenge in purely functional programming languages such as Haskell, and the influential work of Wadler and Peyton Jones [37] has led the Haskell community to use monads extensively to separate pure 'functions' from 'commands' that may read or write mutable state, behave nondeterministically, or interact with the outside world. In this section, we summarize recent results showing that bidirectional transformations can be considered as a form of computational effect and formalized using monads. We will not give a complete review or explanation of monads here, but instead refer to existing tutorials on Haskell programming with monads [63, 11].

We will briefly review the *State monad*. The state monad captures the idea of a mutable state of a given type S.

$$\mathbf{data}\ \mathit{State}\ \sigma\ \alpha = \mathit{State}\ \{\mathit{runState} :: \sigma \to (\alpha, \sigma)\}$$

A computation in the state monad $\mathit{State}\ S\ A$ is a function that takes the initial value $s :: S$ and produces a result $a :: A$ together with a (possibly) updated state $s' :: S$.

The basic monadic operations *return* and $\gg\!=$ (pronounced 'bind') can be defined easily for the state monad:

$$\begin{aligned}&\mathit{return}\ a = \mathit{State}\ (\lambda s \to (a, s))\\ &m \gg\!= f\ \ = \mathit{State}\ (\lambda s \to \mathbf{let}\ (a, s') = \mathit{runState}\ m\ s\ \mathbf{in}\ \mathit{runState}\ (f\ a)\ s')\end{aligned}$$

Here, the *return* operation is a stateful computation that returns a pure value, while $m \gg\!= f$ sequentially composes a stateful computation $m :: \mathit{State}\ S\ A$ with a function $f :: A \to \mathit{State}\ S\ B$, passing the value returned by m to f. In addition, we frequently use the following definition for convenience, to sequentially compose computations with no value dependency:

$$m \gg n = m \gg\!= \backslash\_ \to n$$

The two additional primitive operations which the state monad provides are the ability to *read* the state (and use it as part of some other computation), and to *write* to the state, replacing the old state value with a new one. These operations are often called *get* and *set*. (These operations should not be confused with the *get* and *put* operations of lenses, although, as discussed below, they are related.)

$$
\begin{aligned}
&\mathit{get} &&:: \mathit{State}\ \sigma\ \sigma \\
&\mathit{get} &&= \mathit{State}\ (\lambda s \to (s, s)) \\
&\mathit{set} &&:: \sigma \to \mathit{State}\ \sigma\ () \\
&\mathit{set}\ s' &&= \mathit{State}\ (\lambda s \to ((), s'))
\end{aligned}
$$

One can easily verify that these operations satisfy a number of equations:

$$
\begin{aligned}
&\mathit{get} \gg\!= \lambda s_1 \to \mathit{get} \gg\!= \lambda s_2 \to k\ s_1\ s_2 &&= \mathit{get} \gg\!= \lambda s \to k\ s\ s && (\mathsf{GetGet}) \\
&\mathit{set}\ s \gg \mathit{get} &&= \mathit{set}\ s \gg \mathit{return}\ s && (\mathsf{SetGet}) \\
&\mathit{get} \gg\!= \mathit{set} &&= \mathit{return}\ () && (\mathsf{GetSet}) \\
&\mathit{set}\ s_1 \gg \mathit{set}\ s_2 &&= \mathit{set}\ s_2 && (\mathsf{SetSet})
\end{aligned}
$$

The first equation says that *get* has no side-effect on the state, so doing two *get*s in sequence is the same as doing one and reusing the value twice. The second equation says that *set s* changes the state to s, so that subsequent *get*s see that value. The third says that setting the state to its current value has no effect. The final equation says that if multiple *set*s are performed in sequence, the overall effect is just that of the last one.

The state monad is a concrete example of a more abstract idea: we can axiomatize the idea of a 'monad with state S' purely in terms of the operations and equations they should satisfy [52]. We say that a monad M has a state interface $(\mathit{get}_S, \mathit{set}_S)$ of type S if it supports these two operations, satisfying the laws (GetGet), (SetGet), (GetSet), and (SetSet).

Furthermore, we can consider a single monad M that provides two state interfaces $(\mathit{get}_A, \mathit{set}_A)$ and $(\mathit{get}_B, \mathit{set}_B)$ of (possibly) different types A and B. In addition to the above laws that say how get_A and set_A interact in isolation and likewise for get_B and set_B, we should consider interactions between the two pairs of operations. One natural expectation is that get_A and get_B should commute:

$$
\begin{aligned}
&\mathit{get}_A \gg\!= \lambda a \to \mathit{get}_B \gg\!= \lambda b \to k\ a\ b \\
&= \mathit{get}_B \gg\!= \lambda b \to \mathit{get}_A \gg\!= \lambda a \to k\ a\ b && (\mathsf{GetComm})
\end{aligned}
$$

Another natural expectation one might have is that the two states are independent: that is, updating A has no effect on B and vice versa.

$$
\begin{aligned}
&\mathit{set}_A\ a \gg \mathit{set}_B\ b &&= \mathit{set}_B\ b \gg \mathit{set}_A\ a && (\mathsf{SetASetB}) \\
&\mathit{set}_A\ a \gg \mathit{get}_B &&= \mathit{get}_B \gg\!= \lambda b \to \mathit{set}_A\ a \gg \mathit{return}\ b && (\mathsf{SetAGetB}) \\
&\mathit{set}_B\ b \gg \mathit{get}_A &&= \mathit{get}_A \gg\!= \lambda a \to \mathit{set}_B\ b \gg \mathit{return}\ a && (\mathsf{SetBGetA})
\end{aligned}
$$

If all of these properties hold, then M essentially provides separate copies of A and B, just as if implemented by storing a pair (A, B) and defining get and set operations that read or update the first or second element of the pair respectively.

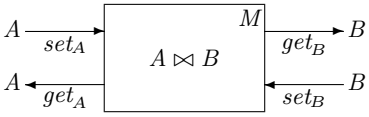


Fig. 8. Monadic BX on sources A, B over monad M

Our interest in such monads in the context of bidirectional transformations arises from omitting some of the above laws, to allow interference between the two states. If such interference is allowed, we call the state monad *entangled* (by a very loose analogy with entangled states in quantum systems). We will show that several forms of bidirectional transformation can be defined in terms of *entangled state monads*.

A *monadic BX* between A and B is a monad M equipped with state interfaces (get_A, set_A) and (get_B, set_B) satisfying the laws (GetGet), (SetGet), (GetSet) for A and for B, and also law (GetComm) about their interaction. We say that M is *very well-behaved* if in addition the laws (SetSet) hold. We write $bx : A \nleftrightarrows_M B$, and think of bx as a record with four fields $get_A\ bx, set_A\ bx, get_B\ bx, set_B\ bx.$ for the four operations.

Figure 8 illustrates the interface of a monadic BX; we visualize M as a box containing an 'entangled pair' of A and B values, written $A \bowtie B$. The arrows indicate that the get_A, get_B operations allow us to inspect the current value of A or B respectively, while set_A and set_B allow us to set the new value of one side, with possible side-effects on the other side.

Lenses as entangled state monads Well-behaved lenses can be viewed as transforming one "mutable state" to another, in the following sense. Given a plain lens $(get, put) : S \nleftrightarrows V$, we can define operations as follows:

$$
\begin{array}{ll}
get_V & :: State\ S\ V \\
get_V & = get_S \gg\!= \lambda s \to return\ (get\ s) \\
set_V & :: V \to State\ S\ () \\
set_V\ v & = get_S \gg\!= \lambda s \to set\ (put\ (s, v))
\end{array}
$$

where $get_S :: State\ S\ S$ and $putS :: S \to State\ S\ ()$ are the get and set operations of $State\ S$. That is, get_V gets the source state and applies the get operation of the lens, while set_V gets the old source state, uses put to compute the new source state, and sets that. Moreover, it is readily verified that these operations form a monadic BX relating S and V. When the lens (get, put) is very well-behaved, the resulting monadic BX is also, and it can be shown that it induces a *monad morphism* from $State\ V$ to $State\ S$ that preserves the get and put operations. (This observation is due to Shkaravska; it is further discussed and applied in [57].)

Given a symmetric lens (in the sense of Hofmann et al. [30]), we can view the $putr :: (A, C) \to (B, C)$ and $putl :: (B, C) \to (A, C)$ operations as operations in the state monad, where the state is the complement type C, as follows:

$$
\begin{aligned}
&\mathit{putr}' &&:: A \to \mathit{State}\ C\ B\\
&\mathit{putr}'\ a &&= \mathit{State}\ (\lambda c \to \mathit{putr}\ (a, c))\\
&\mathit{putl}' &&:: B \to \mathit{State}\ C\ A\\
&\mathit{putl}'\ b &&= \mathit{State}\ (\lambda c \to \mathit{putl}\ (b, c))
\end{aligned}
$$

We could translate the laws for symmetric lenses into laws for the above *State* operations. Instead, however, we show how to construct an entangled-state monadic BX from a symmetric lens. We take M to be $\mathit{State}\ S$ where $S = \{(A, B, C) \mid \mathit{putr}\ (A, C) = (B, C)\}$ (which is equal to $\{(A, B, C) \mid \mathit{putl}\ (B, C) = (A, C)\}$ thanks to the symmetric lens laws). Then the *get* and *set* operations are as follows:

$$
\begin{aligned}
&\mathit{get}_A &&:: \mathit{State}\ S\ A\\
&\mathit{get}_A &&= \mathit{State}\ (\lambda(a, b, c) \to (a, (a, b, c)))\\
&\mathit{set}_A &&:: A \to \mathit{State}\ C\ ()\\
&\mathit{set}_A\ a' &&= \mathit{State}\ (\lambda(a, b, c) \to \mathbf{let}\ (b', c') = \mathit{putr}\ (a, c)\ \mathbf{in}\ ((), (a', b', c')))
\end{aligned}
$$

and symmetrically for $\mathit{get}_B, \mathit{set}_B$.

All of these observations extend to the case of (symmetric) lenses that are very well-behaved. In that case, the monadic BX satisfy the (SetSet) laws, that is, they are very well-behaved.

Bidirectional transformations with effects As noted earlier, monads are used to model (and, in Haskell, to program with) side-effects in a pure setting. This means that the *get* and *set* operations associated with an entangled-state monadic BX might have other effects besides reading from or updating (part of) the state. For example, monadic BX can also capture 'nondeterministic bidirectional transformations' [19], which give a *set* of possible consistent models for the user to choose among when restoring consistency.

One complication that arises if we consider monadic BX over arbitrary monads is how to compose them. This is straightforward in the concrete case of transformations based on state monads $\mathit{State}\ S$, but given two monadic BX of types $A \nleftarrow_M B$ and $B \nleftarrow_N C$, where M and N are different monads, it is not obvious how to form their composition, in part because it is not always clear how to combine two monads M, N. Even if we consider monadic BX over the same base monad M, it is not clear how to define the composition if we do not know anything about the structure of M. Instead, we have shown how to define composition for the special case of monads of the form

$$\mathbf{data}\ \mathit{StateT}\ \tau\ \sigma\ \alpha = \mathit{StateT}\ \{\mathit{runStateT} :: \sigma \to \tau\ (\alpha, \sigma)\}$$

StateT is a standard construction called the *state monad transformer* [38]. Intuitively, $\mathit{StateT}\ M\ S$ is a monad which can behave as M and in addition provides a state S (separate from the capabilities of M). We also make an additional assumption: the get_A and get_B operations are assumed to read from S but have no other effects. That is, we assume that these operations are of the following forms:

$$get_A = StateT\ \{\lambda s \to return\ (read_A\ s, s)\}$$
$$get_B = StateT\ \{\lambda s \to return\ (read_B\ s, s)\}$$

for suitable functions $read_A :: S \to A$ and $read_B :: S \to B$. The intuition for this assumption is that when we are composing monadic BX, the two components need to communicate across the shared interface in order to implement the *set* operations for the composition. If we do not have a side-effect-free way to access the current state then it is not clear how to define composition so that the monadic BX laws are preserved.

Concretely, given two monadic BX $bx_1 : A \nleftarrow_{M_1} B$ and $bx_2 : B \nleftarrow_{M_2} C$ over monads $M_1 = StateT\ M\ S_1$ and $M_2 = StateT\ M\ S_2$ respectively, we can define their composition over $StateT\ M\ (S_1, S_2)$ as follows:

$$\begin{aligned}
get_A\ \ &= get \gg\!= \lambda(s_1, s_2) \to return\ (read_A\ s_1)\\
set_A\ a &= get \gg\!= \lambda(s_1, s_2) \to\\
&\quad\ runStateT\ (set_A\ bx_1\ a)\ s_1 \gg\!= \lambda((), s_1') \to\\
&\quad\ runStateT\ (set_B\ bx_2\ (read_B\ bx_1\ s_1'))\ s_2 \gg\!= \lambda((), s_2') \to\\
&\quad\ set\ (s_1', s_2')
\end{aligned}$$

and symmetrically for get_C and set_C. Intuitively, get_A applies *read* to the first component of the state, while set_A first applies $set_A\ bx_1$ to the first component, then applies $set_B\ bx_2$ to the second component using the new B value of the updated state s_1'. Finally, both components of the state are updated. Technically, this construction is not guaranteed to be correct for arbitrary initial state pairs, so we impose a further constraint that state pairs s_1, s_2 are always *compatible*, in the sense that $read_B\ bx_1\ s_1 = read_B\ bx_2\ s_2$. So the composition is actually defined as an monadic BX over $StateT\ (S_1 \bowtie S_2)$, where $(S_1 \bowtie S_2 = \{(s_1, s_2) \mid read_B\ bx_1\ s_1 = read_B\ bx_2\ s_2\}$. Roughly speaking, the reason why we impose the constraint that *get* operations are side-effect free is to ensure that this state space is well-defined. However, it is not clear that this restriction is really necessary.

Although the entangled state monad formalism is appealing in one sense, the difficulties with defining composition in the general case have motivated consideration of other approaches. One approach is to augment the classical notion of (asymmetric) 'lens' to allow for the possibility of monadic effects during consistency restoration [2]. Another possibility is to consider (symmetric) bidirectional transformations from a coalgebraic perspective, which also naturally extends to allow for the possibility of monadic effects [3, 4]. Interestingly, though these two approaches seem superficially different, one can connect them by viewing coalgebraic BX (or equivalently monadic BX) as spans of monadic lenses. Others such as Pacheco et al. [51] have also considered BX with monadic effects, and [2] discusses and compares the proposals to date.

There are a number of open questions. For example, it would be interesting to find a way to define composition for arbitrary entangled-state monadic BX, for which the *get* operations can have side-effects, or to establish that composition requires $StateT$ structure. There are also unresolved questions regarding the right definition(s) of equivalence of monadic BX or spans of monadic lenses [4].

Finally, it would also be interesting to extend monadic BX to generalize delta-lenses [23, 24] or edit lenses [31].

4.2 Least change

In this section we briefly introduce the ideas behind principles of Least Change for BX; for a fuller discussion, we refer the reader to our paper [17].

The purpose of separating information into separate models or views is to manage information overload: we want to present a human with just the information they need, in order to apply their expertise most effectively.

When we use a BX to maintain consistency between the model used by a human and other information that they do not typically wish to see, there is a risk that we confuse the human. If their model changes as a result of changes elsewhere, their work may be disrupted.

To some extent this is inevitable. It raises, however, a very important question: how can we keep the disruption to a minimum? Intuitively, our BX should not change more than it has to in order to restore consistency between the information our expert sees and everything else.

Meertens, who called BX "constraint maintainers", formulated this as follows [44]:

> The action taken by the maintainer of a constraint after a violation should change no more than is needed to restore the constraint.

Sometimes, what this means is clear. If we accept this formulation it implies, for example, the hippocraticness property introduced in Section 3.1; if there is no need to change anything in order to restore consistency, then the BX should not change anything. In considering examples, we often find similarly clear cases where we feel that the BX should not change information which is irrelevant to restoring consistency. In the Composers examples of Section 2.4, when a composer is added or deleted, we do not expect the dates of a different composer to change, even though they could change arbitrarily without affecting consistency. The BX's job is *only* to restore consistency: it must not change the models beyond what it necessary to do this, even if there might be an argument that a further change was an improvement. (For example, we do not accept that a BX relating a UML model to Java code should reformat the Java code, or that the Composers BX should correct an error in the dates of a composer.)

Going beyond this is tricky. The first problem we encounter is that there may be different ways to measure the size of a change, and hence, to judge which way of restoring consistency involves the least change. We illustrate this by way of an example used at the summer school by Zhenjiang Hu. Suppose we have a BX that relates rectangles, given by their width and height (w, h), with their heights h. We start with a four by four square $(4, 4)$, consistently related with its height, 4. Now the height is changed to 2. What should a BX that obeys a least change principle do? Should it leave the width alone and change the square to a rectangle $(4, 2)$? Or should it leave the shape of the rectangle alone, and

change the square to a smaller square $(2, 2)$? Or perhaps it should minimise change to the perimeter of the rectangle, replacing the $(4, 4)$ square by a $(6, 2)$ rectangle? We see that we have not specified how the size of a change is measured: does the "distance" between two rectangles depend on their width and height independently, for example, or does it depend on the rectangle's shape? This problem can be addressed by making the dependency on the way of measuring change explicit, leading to a notion we call metric least change, which has been explored and implemented by Macedo and Cunha [40].

Definition. A bx $R : M \leftrightarrow N$ is *metric-least*, with respect to given metrics d_M, d_N on M and N, if for all $m \in M$ and for all $n, n' \in N$, we have

$$R(m, n') \;\Rightarrow\; d_N(n, n') \geq d_N(n, \overrightarrow{R}(m, n))$$

and dually.

Unfortunately, as the rectangle example illustrates, there may not be a canonical metric on a space of models. This is a problem in real life too, not only in artifical examples. A user of a modelling tool has an intuitive idea that the distance between two models corresponds to the length of time it would take to edit one model into the other. Different tools provide different capabilities, and hence different times, so we could expect difficulty if we tried to define a standard metric on, say, the space of UML models.

We could assume that this problem could be overcome if necessary, but there are other drawbacks to the metric least change approach. The most fundamental is that, even if there is a clear understanding that one change that could restore consistency is smaller than another, it is not necessarily sensible for the BX to apply the smaller change. The ModelTests example from the Bx Examples Repository [58] illustrates. We refer the reader to the repository for details, but in brief: if a consistency condition applies only to model elements with a particular stereotype, then removing the stereotype may be a consistency-restoring change that is tiny according to the user's intuitive metric, and yet not be desirable behaviour of a BX.

A further problem with metric least change is that, in a sense formalised and proved in our paper [17], computing metric least change is NP hard. Another is that the composition of (lens-like) metric least change BX is not necessarily metric least. Let us look further.

Beyond metric least change: least surprise When people cooperate, each working on their own model and applying a BX periodically to restore consistency, they will most likely have negotiated a way of working that involves applying the BX often enough that neither does a lot of outdated work, but not so often as to be destabilising. Hidden behind this idea is the assumption that the behaviour of the BX is predictable in that a sufficiently small change to one model will cause only a small change to the other. This idea is formalised in mathematical analysis as *continuity*. In our paper [17] we consider several variants of continuity, the most promising of which is Hölder continuity.

Other ideas Several ideas may repay further study. We may consider not only changes to the models whose consistency is being maintained, but also to certain sets of auxiliary information, such as the *witness structures* that help to demonstrate consistency.

Finally, exploiting the observation that, even in our tricky examples, there tend to be many situations in which it *is* clear what a BX should do, we may envisage a BX *tool that goes beep*. This tool knows its own limitations. From some states of the world (e.g. some pairs (m, n) of models that fall into a particular subspace pair) it knows how to restore consistency in an unsurprising way. If asked to, it will do so silently. From other states of the world, it doesn't know what to do, or isn't confident about it, so it will beep. The user may want to check, perhaps even amend, what the tool did; or the tool may have given up and done nothing. In this way, the human user's effort may be saved for the situations that are hard to automate.

4.3 Dependently typed BX

Proof-relevance The basic picture of a consistency relation $R \subseteq M \times N$ and two consistency restorers $\overrightarrow{R} : M \times N \to N$ and $\overleftarrow{R} : M \times N \to M$ from Section 3.1 leaves implicit the underlying notion of *update*, relying on a scenario in which inputs to $\overrightarrow{R}$ or $\overleftarrow{R}$ reflect an updated value in the M, respectively N, argument. We now consider *proof-relevant* interpretations, via the established identification of propositions as types in dependent type theory: both updates *and* (proofs of) consistency are represented by families of types, with:

- $\partial^A_{a\,a'}$, representing those updates (edits) δ which transform $a : A$ into $a' : A$, symbolically $\delta : a \mapsto a'$; similarly for $\partial^B_{b\,b'}$; for classical state-based formalisms such as asymmetric lenses, take $\partial^A_{a\,a'}$ to be the trivial singleton family, inhabited everywhere by a dummy witness to each state update $a \mapsto a'$;
- $R_{a\,b}$, representing witnesses r to the proposition '$R(a, b)$'; such families R may be seen as generalising the notion of *lens complement* [43] (itself generalising view-update "with constant complement" [8]; for lenses, the consistency relation is implicit, but recoverable as the *graph relation* of the *get* operation).

As well as dependent types themselves (corresponding set-theoretically to indexed families of sets), the 'propositions-as-types' correspondence, where type inhabitants correspond to *proofs* (witnesses, as above) of the corresponding proposition, crucially extends to embrace:

- *existential* quantification, via the 'Σ-type' $\Sigma_{x:X}\, Y$, where Y is a family indexed over X, whose inhabitants are *pairs* (x, y) such that $x : X$ and $y : Y(x)$; the familiar Cartesian product $X \times Y$ arises as an instance of $\Sigma_{x:X}\, Y$ for the *constant* family Y over X. Iterated Σ-types are inhabited by nested pairs; but in the interests of readability, we suppress nested parentheses in favour of tuple notation;

- *universal* quantification, via the 'Π-type' $\Pi_{x:X}\ Y$, where Y is a family indexed over X, whose inhabitants are *functions* f such that $x : X$ implies $f\ x : Y(x)$; λ-abstraction provides a way to construct such functions in canonical form; similarly to the above, we treat iterated λ-abstraction and application in the usual way. Implication between propositions, given the usual function space $X \Rightarrow Y$, arises as an instance of $\Pi_{x:X}\ Y$ for the constant family Y over X.
- *equality*, as a distinguished relation/type family, satisfying a strong intensional form of the Leibniz property.

For a detailed treatment of this interpretation, as a basis for constructive mathematics and functional programming, we refer the interested reader to standard texts [49, 39].

The above set-up, of proof-relevant consistency relations and dependent type families of updates, is interpretable in the *bicategory* [10] $\mathcal{R}el$ of (proof-relevant) *relations*: in type theory, with 0-cells given by types A, B; with 1-cells given by relations R, identity and composition given by the *identity* type, and relational composition $(R \otimes S)_{a\ c} =_{\text{def}} \Sigma_{b:B}\ R_{a\ b} \times S_{b\ c}$; and with 2-cells the *proof-relevant* inclusions $R \subseteq_p S =_{\text{def}} p : \Pi_{a:A,b:B}\ R_{a\ b} \Rightarrow S_{a\ b}$.

Space forbids a detailed discussion of bicategories, but the reader may gain some intuition for them by regarding them as a common generalisation of the 'ordinary' set-theoretic structure of composition and inclusion on relations, and that of *monoidal* categories, just as 'ordinary' categories may be regarded as a common generalisation of the theory of composition of set-theoretic functions, and that of preorders on the one hand, and monoids on the other.

BX are bisimulations Now, forward consistency restoration specifies that: given R-consistent a, b (witnessed by r), and an A-update $\delta : a \mapsto a'$, there should exist a B-update $\delta' : b \mapsto b'$, together with a new witness r' to the R-consistency of a', b' (and vice versa in the backward direction). That is to say, for each pair a', b, forward consistency restoration transforms the triple (a, δ, r) to the triple (b', r, δ'), where we consider those triples as inhabiting the types $\Sigma_{a:A}\ \partial^A_{a\ a'} \times R_{a\ b}$, respectively $\Sigma_{b':B}\ R_{a'\ b'} \times \partial^B_{b\ b'}$, using Σ-types to package up the existential quantifications.

In terms of proof-relevant inclusions, we thus have (with ∂^{A° the *opposite* relation to ∂^A) characterised the forward and backward restoration functions as having the following types:

$$\partial^{A^\circ} \otimes R \subseteq_{\overrightarrow{R}} R \otimes \partial^{B^\circ} \qquad \partial^{B^\circ} \otimes R \subseteq_{\overleftarrow{R}} R \otimes \partial^{A^\circ}$$

These two inclusions encode algebraically the usual diagrammatic properties defining a *bisimulation* between two labelled transition systems, thus (very compactly!) justifying the slogan [41] that the forward and backward transformations witness R as a proof-relevant bisimulation between the model spaces A, B, with updates ∂^A, ∂^B defining the transitions between model states. We write:

$$\mathcal{R} =_{\text{def}} (R, \overrightarrow{R}, \overleftarrow{R}) : \mathcal{A} \nrightarrow_R \mathcal{B}$$

where $\mathcal{A} =_{\text{def}} (A, \partial^A)$ and $\mathcal{B} =_{\text{def}} (B, \partial^B)$.

Why bicategories? The above structure yields a further bicategory $\mathcal{B}isim$, with 0-cells given by the model spaces $\mathcal{A}$, 1-cells given by BX as proof-relevant bisimulations $\mathcal{R} : \mathcal{A} \nrightarrow_R \mathcal{B}$, and 2-cells given by proof-relevant *equivalences*[3] $R \subseteq_p R' \subseteq_q R$ between the underlying consistency relations R, R'.

There is a (forgetful) homomorphism of bicategories between $\mathcal{B}isim$ and $\mathcal{R}el$: on 0-cells, it forgets the update structure; on 1-cells, it maps a BX $\mathcal{R}$ to its underlying consistency relation R; and on 2-cells, an equivalence (p, q) maps to p.

This homomorphism depends on a definition of composition $\otimes$ between BX (that is, between bisimulations) that is new, as far as we are aware, but which generalises existing definitions of lens composition:

$$\mathcal{R} \otimes \mathcal{S} =_{\text{def}} (R \otimes S, \overrightarrow{R} \otimes \overrightarrow{S}, \overleftarrow{R} \otimes \overleftarrow{S}) : \mathcal{A} \nrightarrow_{R \otimes S} \mathcal{C}$$

with $(\overrightarrow{R} \otimes \overrightarrow{S})\ (a', \delta_a, (b, r, s)) = (c', (b', r', s'), \delta_c)$ where the actions of $\overrightarrow{R}, \overrightarrow{S}$ on the relevant iterated Σ-types are given by $(b', r', \delta_b) =_{\text{def}} \overrightarrow{R}\ (a', \delta_a, r)$ and $(c', s', \delta_c) =_{\text{def}} \overrightarrow{S}\ (b', \delta_b, s)$. The corresponding definitions for $\overleftarrow{R} \otimes \overleftarrow{S}$ are similar, but omitted for brevity.

Now, one might ask, why all this machinery? At least from a structural perspective, we now believe we have a satisfactory *compositional* account of BX. One criticism of Meertens' original framework for consistency maintenance is that his restorers, and their underlying consistency relations, do not naturally compose. The purpose of the above constructions is to build in enough structure, at the model *plus* update *plus* witness level, to ensure that composition is not only well-defined, but also that the projection from $\mathcal{B}isim$ down to mere relations $\mathcal{R}el$ preserves structure. That is, in essence, that the composition of consistency relations should itself be the consistency relation of the composite BX.

Additional properties Familiar BX properties correspond to additional definitions of structure on model spaces, and to strong, intensional constraints on the interaction between consistency restoration and such structure. We take as a good sign of the robustness of our definitions that such additional structure gives rise to *full* sub-bicategories of $\mathcal{B}isim$; that is, the definitions of composition, *etc.* remain unchanged, but we can further prove that they preserve the additional structure.

[3] The reader troubled by the apparent lack of generality implied by equivalences between consistency relations may wonder whether a richer class of 2-cells might fit with this analysis. Certainly, the choice of equivalences is *sufficient* to consider all the constructions and coherence conditions necessary for the definition of a bicategory. Moreover, the need for consistency restoration functions to go 'back-and-forth', restoring the *same* consistency relation R (at least up to extensional equivalence), seems to make our choice also *necessary*.

For example, hippocracticness in this setting corresponds to model spaces having the structure of *reflexive graphs*, and to consistency restoration preserving such structure on-the-nose. That is, we have distinguished 'identity' updates $\iota_a : \partial^A_{a\,a}$ for each model space, such that $\overrightarrow{R}\ (a, \iota_a, r) = (b, \iota_b, r)$, and similarly for $\overleftarrow{R}$. Moreover, given $\mathcal{R}$ and $\mathcal{S}$ both hippocratic, then we can show that so too is $\mathcal{R} \otimes \mathcal{S}$.

Similarly, we may consider *history ignorance* for BX, by considering additional structure of *composition* of updates, and its strict preservation under consistency restoration. Such BX, too, are closed under $\otimes$. The combination of hippocraticness and history ignorance, together with proofs of the corresponding equational laws, thus amounts to considering model spaces as fully-fledged *categories*. So history ignorance, in this generalised setting, might more neutrally be described as the property of being *compositional for updates*.

Dependently-typed programming languages such as Agda [50] or Idris [15] offer a natural home for such proof-relevant constructions, with dependent types as strong, machine-checkable, correctness specifications: we can, for example, give a type-theoretic characterisation of the *alignment* problem in the BX literature [9, 24], exhibiting its type as that of a (heuristic) *search* problem: to (forward) *align* $a' : A$ with $b : B$ is to compute an inhabitant of $\Sigma_{a:A}\, \partial^A_{a\,a'} \times R_{a\,b}$.

Discussion Our generalisation shares some of the same underlying machinery as Diskin *et al.*'s *symmetric delta lenses* [24], where model spaces are given as categories, but much of the bicategorical structure of $\mathcal{B}isim$, and relationships to other settings which we describe here, is new.

In particular, Hofmann, Pierce and Wagner's symmetric edit lenses [31, 29] are an instance of our framework: their 'stateful monoid homomorphisms', where defined, can be precisely captured by the more refined dependent types of our consistency restoration functions $\overrightarrow{R}$, $\overleftarrow{R}$; while their 'consistency relation' K exactly corresponds to a consistency relation in our terms given by the dependent type family $R_{a\,b} =_{\text{def}} \{c \in C \mid (a, c, b) \in K\}$. Indeed, by insisting on edit *monoid* structure, and consistency restoration as a 'stateful monoid homomorphism', Hofmann *et al.* build in precisely the additional properties of hippocraticness and compositionality for edits sketched above.

The reflexive graph structure necessary for hippocraticness has close connections to that explored by Cai *et al.* in the theory of static differentiation of functions [16].

The sketch above of our type-theoretical and (bi-)categorical approach is necessarily brief; we defer a longer treatment in full detail to further publications. We nevertheless hope that the reader might glimpse, at least, a unifying mathematical basis for (all?) existing BX formalisms, as well as a starting point for comparison with existing work in the related area of version control systems and patch theory [62, 6], where type-theoretic ideas have also proved fruitful. We further conjecture that interpretations of our constructions in other (enriched) categorical settings may shed light on (geo-)metric accounts of consistency restoration, in terms of a 'differential geometry of consistency restoration' [42].

References

1. Faris Abou-Saleh, James Cheney, Jeremy Gibbons, James McKinna, and Perdita Stevens. Notions of bidirectional computation and entangled state monads. In *Mathematics of Program Construction*, number 9129 in Lecture Notes in Computer Science, pages 187–214. Springer-Verlag, 2015.
2. Faris Abou-Saleh, James Cheney, Jeremy Gibbons, James McKinna, and Perdita Stevens. Reflections on monadic lenses. In *A List of Successes That Can Change the World — Essays Dedicated to Philip Wadler on the Occasion of His 60th Birthday*, number 9600 in Lecture Notes in Computer Science, pages 1–31, 2016.
3. Faris Abou-Saleh, James McKinna, and Jeremy Gibbons. Coalgebraic aspects of bidirectional computation. In *BX 2015*, volume 1396 of *CEUR-WS*, pages 15–30, 2015.
4. Faris Abou-Saleh, James McKinna, and Jeremy Gibbons. Coalgebraic aspects of bidirectional computation. *Journal of Object Technology*, 2017. In press.
5. Scott W. Ambler. Mapping objects to relational databases: O/R Mapping in detail. `http://www.agiledata.org/essays/mappingObjects.html`, 1998.
6. Carlo Angiuli, Edward Morehouse, Daniel R. Licata, and Robert Harper. Homotopical patch theory. In *International Conference on Functional Programming*, pages 243–256. ACM, 2014.
7. Anthony Anjorin. Class diagrams to database schemas v0.1. *BX Repository*, last retrieved January 2017. `http://bx-community.wikidot.com/examples:classdiagramstodatabaseschemas`.
8. François Bancilhon and Nicolas Spyratos. Update semantics of relational views. *ACM Transactions on Database Systems*, 6(4):557–575, 1981.
9. Davi M. J. Barbosa, Julien Cretin, Nate Foster, Michael Greenberg, and Benjamin C. Pierce. Matching lenses: Alignment and view update. In Paul Hudak and Stephanie Weirich, editors, *International Conference on Functional Programming*, pages 193–204. ACM, 2010.
10. Jean Bénabou. *Reports of the Midwest Category Seminar*, volume 47 of *Lecture Notes in Mathematics*, chapter Introduction to bicategories, pages 1–77. Springer-Verlag, 1967.
11. Nick Benton, John Hughes, and Eugenio Moggi. Monads and effects. In Gilles Barthe, Peter Dybjer, Luis Pinto, and João Saraiva, editors, *Applied Semantics*, volume 2395 of *Lecture Notes in Computer Science*, pages 42–122. Springer-Verlag, 2000.
12. Aaron Bohannon, J. Nathan Foster, Benjamin C. Pierce, Alexandre Pilkiewicz, and Alan Schmitt. Boomerang: Resourceful lenses for string data. In *Principles of Programming Languages*, 2008.
13. Aaron Bohannon, Benjamin C. Pierce, and Jeffrey A. Vaughan. Relational lenses: A language for updatable views. In *Principles of Database Systems*, pages 338–347. ACM, 2006.
14. John M. Boyer. W3C XForms. `https://www.w3.org/TR/xforms/`, October 2009.
15. Edwin Brady. *Type-Driven Development with Idris*. Manning Publications, 2017.
16. Yufei Cai, Paolo G. Giarrusso, Tillmann Rendel, and Klaus Ostermann. A theory of changes for higher-order languages: Incrementalizing λ-calculi by static differentiation. In *Programming Language Design and Implementation*, pages 145–155. ACM, 2014.
17. James Cheney, Jeremy Gibbons, James McKinna, and Perdita Stevens. On principles of least change and least surprise for bidirectional transformations. *Journal of Object Technology*, 2017. To appear.

18. James Cheney, James McKinna, Perdita Stevens, and Jeremy Gibbons. Towards a repository of BX examples. In *BX Workshop*, March 2014.
19. Antonio Cicchetti, Davide Di Ruscio, Romina Eramo, and Alfonso Pierantonio. JTL: A bidirectional and change propagating transformation language. In *SLE 2010*, volume 6563 of *Lecture Notes in Computer Science*, pages 183–202. Springer-Verlag, 2010.
20. Ezra Cooper, Sam Lindley, Philip Wadler, and Jeremy Yallop. The essence of form abstraction. In *Asian Symposium on Programming Languages and Systems*, volume 5356 of *Lecture Notes in Computer Science*, pages 205–220. Springer, 2008.
21. Krzysztof Czarnecki, J. Nathan Foster, Zhenjiang Hu, Ralf Lämmel, Andy Schürr, and James F. Terwilliger. Bidirectional transformations: A cross-discipline perspective. In Richard F. Paige, editor, *Theory and Practice of Model Transformations (ICMT)*, volume 5563 of *Lecture Notes in Computer Science*, pages 260–283. Springer, 2009.
22. Christopher J. Date. *View Updating and Relational Theory.* O'Reilly, 2012.
23. Zinovy Diskin, Yingfei Xiong, and Krzysztof Czarnecki. From state- to delta-based bidirectional model transformations: The asymmetric case. *Journal of Object Technology*, 10:6: 1–25, 2011.
24. Zinovy Diskin, Yingfei Xiong, Krzysztof Czarnecki, Hartmut Ehrig, Frank Hermann, and Fernando Orejas. From state- to delta-based bidirectional model transformations: The symmetric case. In Jon Whittle, Tony Clark, and Thomas Kühne, editors, *Model Driven Engineering Languages and Systems (MoDELS)*, volume 6981 of *Lecture Notes in Computer Science*, pages 304–318. Springer, 2011.
25. Hartmut Ehrig, Gregor Engels, Hans-Jörg Kreowski, and Grzegorz Rozenberg, editors. *Handbook of Graph Grammars and Computing by Graph Transformation*, volume 2: Applications, Languages and Tools. World Scientific, 1999.
26. J. Nathan Foster, Michael B. Greenwald, Jonathan T. Moore, Benjamin C. Pierce, and Alan Schmitt. Combinators for bidirectional tree transformations: A linguistic approach to the View Update Problem. In *Principles of Programming Languages*, pages 233–246. ACM, 2005.
27. Nate Foster, Kazutaka Matsuda, and Janis Voigtländer. Three complementary approaches to bidirectional programming. In Jeremy Gibbons, editor, *Spring School on Generic and Indexed Programming*, volume 7470 of *Lecture Notes in Computer Science*, pages 1–46. Springer, 2010.
28. Stephen J. Hegner. An order-based theory of updates for closed database views. *Annals of Mathematics and Artificial Intelligence*, 40(1-2):63–125, 2004.
29. Martin Hofmann. Modular edit lenses. In Jeremy Gibbons and Perdita Stevens, editors, *Summer School on Bidirectional Transformations*, volume 9715 of *Lecture Notes in Computer Science*, pages 75–101. Springer, 2018. In this volume.
30. Martin Hofmann, Benjamin C. Pierce, and Daniel Wagner. Symmetric lenses. In Thomas Ball and Mooly Sagiv, editors, *Principles of Programming Languages*, pages 371–384. ACM, 2011.
31. Martin Hofmann, Benjamin C. Pierce, and Daniel Wagner. Edit lenses. In John Field and Michael Hicks, editors, *Principles of Programming Languages*, pages 495–508. ACM, 2012.
32. Zhenjiang Hu, Andy Schürr, Perdita Stevens, and James F. Terwilliger. Bidirectional transformations "bx" (Dagstuhl Seminar 11031). *Dagstuhl Reports*, 1(1):42–67, 2011.
33. Michael Johnson and Robert D. Rosebrugh. Lens put–put laws: Monotonic and mixed. In *BX Workshop*, 2012. In ECEASST volume 49.

34. Michael Johnson and Robert D. Rosebrugh. Spans of lenses. In James Terwilliger and Soichiro Hidaka, editors, *BX Workshop*, volume 1133 of *CEUR Workshop Proceedings*, pages 112–118. CEUR-WS.org, 2014.
35. Michael Johnson and Robert D. Rosebrugh. Unifying set-based, delta-based and edit-based lenses. In Anthony Anjorin and Jeremy Gibbons, editors, *BX Workshop*, volume 1571 of *CEUR Workshop Proceedings*, pages 1–13. CEUR-WS.org, 2016.
36. Michael Johnson, Robert D. Rosebrugh, and Richard J. Wood. Lenses, fibrations and universal translations. *Mathematical Structures in Computer Science*, 22(1):25–42, 2012.
37. Simon L. Peyton Jones and Philip Wadler. Imperative functional programming. In *Principles of Programming Languages*, pages 71–84, 1993.
38. Sheng Liang, Paul Hudak, and Mark Jones. Monad transformers and modular interpreters. In *Principles of Programming Languages*, pages 333–343, 1995.
39. Zhaohui Luo. *Computation and Reasoning: A Type Theory for Computer Science*. International Series of Monographs on Computer Science. Oxford University Press, 1994.
40. Nuno Macedo and Alcino Cunha. Least-change bidirectional model transformation with QVT-R and ATL. *Software and System Modeling*, 15(3):783–810, 2016.
41. James McKinna. Bidirectional transformations are proof-relevant bisimulations. Extended Abstract presented at ICFP Workshop TyDe, Nara, Japan, 2016. Video available at: `https://www.youtube.com/watch?v=33RYwcIQ7UM`.
42. James McKinna. Bidirectional transformations with deltas: A dependently typed approach (talk proposal). In *Bx Workshop, ETAPS*, 2016. `http://ceur-ws.org/Vol-1571/paper_11.pdf`.
43. James McKinna. Complements witness consistency. In *Bx Workshop, ETAPS*, 2016. `http://ceur-ws.org/Vol-1571/paper_10.pdf`.
44. Lambert Meertens. Designing constraint maintainers for user interaction. CWI, Amsterdam; available from `http://www.kestrel.edu/home/people/meertens/pub/dcm.ps`, June 1998.
45. Lambert Meertens. Designing constraint maintainers for user interaction. In Shin-Cheng Mu, editor, *Third Workshop on Programmable Structured Documents*, pages 1–3, PSD Laboratory, Tokyo University, 2005.
46. Microsoft. Windows Presentation Foundation. `https://msdn.microsoft.com/en-us/library/ms754130.aspx`, 2006.
47. Eugenio Moggi. Notions of computation and monads. *Information and Computation*, 93(1):55–92, 1991.
48. Ted Neward. The Vietnam of computer science. `http://blogs.tedneward.com/post/the-vietnam-of-computer-science/`, June 2006.
49. Bengt Nordström, Kent Petersson, and Jan Smith. *Programming in Martin-Löf's Type Theory: An Introduction*, volume 7 of *International Series of Monographs on Computer Science*. Oxford University Press, 1990. Available in PDF at: `https://www.cse.chalmers.se/research/group/logic/book/book.pdf`.
50. Ulf Norell. Dependently typed programming in Agda. In Pieter Koopman, Rinus Plasmeijer, and Doaitse Swierstra, editors, *Advanced Functional Programming: 6th International School, AFP 2008, Heijen, The Netherlands, May 2008, Revised Lectures*, pages 230–266. Springer, 2009.
51. Hugo Pacheco, Zhenjiang Hu, and Sebastian Fischer. Monadic combinators for "putback" style bidirectional programming. In *Partial Evaluation and Program Manipulation*, pages 39–50. ACM, 2014.

52. Gordon D. Plotkin and John Power. Notions of computation determine monads. In *Foundations of Software Science and Computation Structures*, volume 2303 of *Lecture Notes in Computer Science*, pages 342–356. Springer-Verlag, 2002.
53. Terrence W. Pratt. Pair grammars, graph languages and string-to-graph translations. *Journal of Computer and System Sciences*, 5(6):560–595, 1971.
54. Grzegorz Rozenberg, editor. *Handbook of Graph Grammars and Computing by Graph Transformation*, volume 1: Foundations. World Scientific, 1997.
55. Andy Schürr. Specification of graph translators with Triple Graph Grammars. In Ernst W. Mayr, Gunther Schmidt, and Gottfried Tinhofer, editors, *Graph-Theoretic Concepts in Computer Science (WG94)*, volume 903 of *Lecture Notes in Computer Science*, pages 151–163. Springer, 1994.
56. Andy Schürr and Felix Klar. 15 years of Triple Graph Grammars. In Hartmut Ehrig, Reiko Heckel, Grzegorz Rozenberg, and Gabriele Taentzer, editors, *Graph Transformations (ICGT)*, volume 5214 of *Lecture Notes in Computer Science*, pages 411–425. Springer, 2008.
57. Olha Shkaravska. Side-effect monad, its equational theory and applications. Seminar slides available at: `http://www.ioc.ee/~tarmo/tsem05/shkaravska1512-slides.pdf`, 2005.
58. Perdita Stevens. ModelTests v0.1 in Bx Examples Repository. `http://bx-community.wikidot.com/examples:modeltests`. Date retrieved: 6 Feb 2017.
59. Perdita Stevens. Bidirectional model transformations in QVT: Semantic issues and open questions. *Software and System Modeling*, 9(1):7–20, 2010.
60. Perdita Stevens. Observations relating to the equivalences induced on model sets by bidirectional transformations. In *BX Workshop*, 2012. In ECEASST volume 49.
61. Perdita Stevens, James McKinna, and James Cheney. Composers v0.1. *BX Repository*, last retrieved January 2017. `http://bx-community.wikidot.com/examples:composers`.
62. Wouter Swierstra and Andres Löh. The semantics of version control. In *Onward!*, pages 43–54, 2014.
63. Philip Wadler. Monads for functional programming. In Johan Jeuring and Erik Meijer, editors, *Advanced Functional Programming*, volume 925 of *Lecture Notes in Computer Science*, pages 24–52. Springer-Verlag, 1995.