Jeremy Gibbons
Perdita Stevens (Eds.)

Tutorial

LNCS 9715

Bidirectional Transformations

International Summer School
Oxford, UK, July 25–29, 2016
Tutorial Lectures

Springer

EXTRAS ONLINE

Jeremy Gibbons · Perdita Stevens (Eds.)

Bidirectional Transformations

International Summer School
Oxford, UK, July 25–29, 2016
Tutorial Lectures

Springer

Editors
Jeremy Gibbons
University of Oxford
Oxford
UK

Perdita Stevens
University of Edinburgh
Edinburgh
UK

ISSN 0302-9743 ISSN 1611-3349 (electronic)
Lecture Notes in Computer Science
ISBN 978-3-319-79107-4 ISBN 978-3-319-79108-1 (eBook)
https://doi.org/10.1007/978-3-319-79108-1

Library of Congress Control Number: 2018937394

LNCS Sublibrary: SL2 – Programming and Software Engineering

Printed on acid-free paper

This Springer imprint is published by the registered company Springer International Publishing AG
part of Springer Nature
The registered company address is: Gewerbestrasse 11, 6330 Cham, Switzerland

Preface

"Bidirectional transformations" (BX) are a means of maintaining consistency between multiple information sources: When one source is edited, the others may need updating to restore consistency. BX have applications in databases, user interface design, model-driven development, and many other domains.

This volume represents the lecture notes from the Summer School on Bidirectional Transformations, held at Lady Margaret Hall in Oxford during July 25–26, 2016. The school was one of the final activities on the project "A Theory of Least Change for Bidirectional Transformations", running at the University of Oxford and the University of Edinburgh from 2013 to 2017 and funded by the UK Engineering and Physical Sciences Research Council (grant numbers EP/K020919/1 and EP/K020218/1).

http://www.cs.ox.ac.uk/projects/tlcbx/
http://groups.inf.ed.ac.uk/bx/

The summer school was aimed at graduate students and researchers in BX and related areas. It played host to lectures from five external experts in BX, book-ended by some additional lectures from the TLCBX project team on the results obtained during the project. Lecture notes on all six topics are included here:

– The chapter "Introduction to Bidirectional Transformations" by the TLCBX team (Faris Abou-Saleh, James Cheney, Jeremy Gibbons, James McKinna, and Perdita Stevens) sets the scene, introducing general notions of BX (including the view-update problem, the relational perspective via consistency restoration, varieties of lens, and triple graph grammars), in order to establish a common foundation for and highlight some differences in approach between the chapters that follow. It also briefly presents some of the results from the project: on effectful lenses and the entangled state monad; on complement structures as witnesses to consistency, including the use of dependent types (with witnesses more specifically to proofs of consistency); and on the least change principle.
– "Triple Graph Grammars" (TGGs) provide a rule-based means of specifying a consistency relation over two graph languages. TGG rules are direction agnostic, describing the simultaneous creation of pairs of consistent graphs in both languages. Correspondences between elements in the different languages are thereby represented explicitly as a third "correspondence graph". Many useful tools can be derived automatically from a TGG including an instance generator, consistent forward and backward transformations, and incrementally working synchronizers, which are able to realize forward and backward change propagation without incurring unnecessary information loss. Anthony Anjorin's chapter "An Introduction to Triple Graph Grammars as an Implementation of the Delta-Lens Framework" introduces TGGs as a pragmatic implementation of the "symmetric delta lens" framework proposed by Diskin et al.

- Martin Hofmann's chapter "Modular Edit Lenses" presents symmetric edit lenses, as developed in a series of papers by David Wagner, Benjamin Pierce, and himself. Symmetric lenses form a general framework for the modular construction of bidirectional synchronizers, and generalize the popular lens framework of Foster and Pierce to a symmetric setting. The chapter describes both the state-based and the edit-based (or delta-based) versions, and concludes with an extended illustration involving tree-structured data. The main focus is on edit lenses, and the categorical combinators which allow for their modular construction. The chapter serves as a reading guide to the original series of papers.
- "Putback-based" bidirectional programming is an approach that allows the programmer to specify a bidirectional transformation by writing only the "putback" component; the unique corresponding forwards transformation is derived from this for free. A key distinguishing feature of putback-based bidirectional programming is its full control over the bidirectional behaviour, which is important for specifying one's intentions for a bidirectional transformation without any ambiguity. The chapter "Principles and Practice of Bidirectional Programming in BiGUL" by Zhenjiang Hu and Hsiang-Shang Ko introduces the authors' putback-based bidirectional programming language BiGUL. They explain the principles of the language, and show how to develop various kinds of bidirectional transformation in it.
- Richard Paige expounds the view that bidirectional transformations are artifacts that can (and probably should) be engineered, following a suitable lifecycle. In his chapter "Engineering Bidirectional Transformations," he considers the different phases of such a BX engineering lifecycle, explores ways in which the requirements, architectures, designs and implementations of BX can be specified, and discusses what support can be used to help verify or validate such artifacts.

In addition, Mike Johnson lectured on "Mathematical Foundations of Bidirectional Transformations", but for personal reasons was not able to supply a chapter for this volume. His lectures were based on his papers with Bob Rosebrugh, including "Algebras and Update Strategies" (doi 10.3217/jucs-016-05-0729), "Fibrations and Universal View Updatability" (doi 10.1016/j.tcs.2007.06.004), and a series of papers in the annual BX Workshop, starting with "Lens Put–Put Laws: Monotonic and Mixed" (in Volume 49 of *Electronic Communications of the EASST*).

The chapters were reviewed by the lecturers, each chapter getting one review from one of the other invited lecturers and one from a member of the TLCBX team. In addition, some participants in the summer school—Jonathan DiLorenzo, Valdemar Graciano Neto, and Seyyed Shah—each reviewed a chapter.

We would like to express our thanks to EPSRC for funding the project, including the summer school itself; and to Karen Barnes in the Department of Computer Science at Oxford and Gemma Sheppard at Lady Margaret Hall for their help in running the school. We also wish to thank the participants at the school, whose enthusiasm made worthwhile the effort of organizing the event; special thanks go to the participants who volunteered to review one of these chapters. And of course, it couldn't have happened without the generous contributions of time and energy from the invited lecturers.

Finally, we thank Sigrid Rosen and Steffen Jost of Ludwig-Maximilians-Universität München for helping to obtain files and permissions relating to Martin Hofmann's

chapter. Martin died while walking on the mountain Nikko Shirane in Japan, in January 2018 while we were finalizing this volume. BX was only one of many fields of computer science where Martin made important contributions; it is sad that he did not survive to make even more. Far sadder, though, is the loss of someone whose kindness and positivity improved any occasion, scientific or social, where he was present. We will miss him, and our hearts go out to his wife Annette and his three children. We dedicate this volume to his memory.

February 2018

Jeremy Gibbons
Perdita Stevens

Contents

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, Oxford, UK
{faris.abou-saleh,jeremy.gibbons}@cs.ox.ac.uk

[2] School of Informatics, University of Edinburgh, Edinburgh, UK
{james.cheney,james.mckinna,perdita.stevens}@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*

© Springer International Publishing AG, part of Springer Nature 2018
J. Gibbons and P. Stevens (Eds.): Bidirectional Transformations, LNCS 9715, pp. 1–28, 2018.
https://doi.org/10.1007/978-3-319-79108-1_1

in databases. We say more about this motivating scenario in Sect. 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 Sect. 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 Sect. 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 Sect. 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 Sect. 2, we describe a number of motivating scenarios for BX. In Sect. 3, we sketch some of the main approaches that have been used to model and implement BX. Finally, in Sect. 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

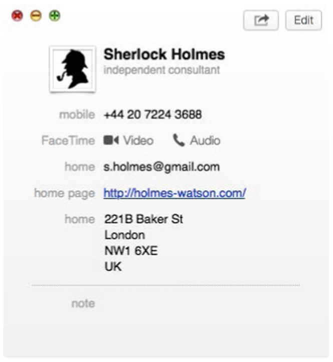
A very simple degenerate class of BX arises between two data sources when each maintains a faithful record of the whole overall state. The overall system may then be thought of as mere data conversion between different formats. Crucially, when one side is updated, the other may simply be discarded and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

Consider the example of an address book application, illustrated in Fig. 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)



Sherlock Holmes
independent consultant
+44 20 7224 3688
s.holmes@gmail.com
http://holmes-watson.com/
221B Baker St
London
NW1 6XE
UK

(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 Fig. 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 and Klar [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 Sect. 3.4.)

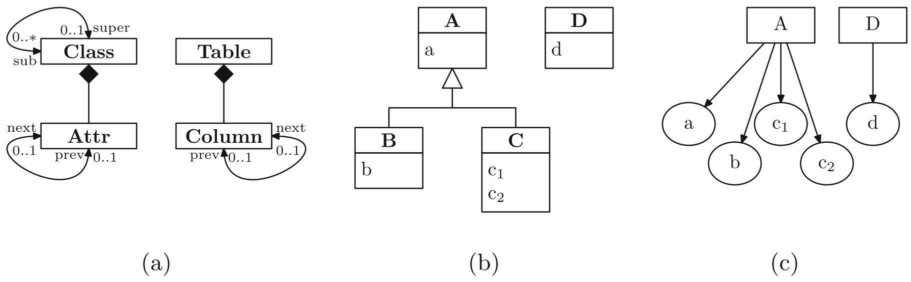


Fig. 3. (a) Two metamodels, (b) a class model, and (c) a table model

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

$$
\begin{aligned}
M &= \{Name \times Dates \times Nationality\} \\
N &= [Name \times Nationality]
\end{aligned}
$$

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:

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

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

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 $(R, \overrightarrow{R}, \overleftarrow{R}) : M \Leftrightarrow N$. 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:

$$\begin{aligned} &\forall m', n.\ R\ (m', \overrightarrow{R}(m', n))\\ &\forall m, n'.\ R\ (\overleftarrow{R}(m, n'), n') \end{aligned}$$

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

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

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

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

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

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

$$\begin{array}{ll}M &= \{\mathit{Name} \times \mathit{Dates} \times \mathit{Nationality}\} \\ N &= [\mathit{Name} \times \mathit{Nationality}]\end{array}$$

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 Sect. 2.4, and m', n' the corresponding states with Copland missing:

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

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 [9,12,13,26,30,31] by Pierce *et al.* on *lenses*. An asymmetric lens $(\mathit{get}, \mathit{put}) : S \rightleftharpoons V$ from source S to view V consists of two functions

$$\begin{aligned}&get : S \rightarrow V\\&put : S \times V \rightarrow S\end{aligned}$$

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 $create : V \rightarrow S$ 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

$$\begin{array}{lll l}\forall s, v. & put\ (s, get\ s) & = s & (\mathsf{GetPut})\\ \forall s, v. & get\ (put\ (s, v)) & = v & (\mathsf{PutGet})\end{array}$$

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 Sect. 3.1. The consistency relationship R being maintained is

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

Forwards consistency restoration $\overrightarrow{R} : S \times V \rightarrow V$ is trivial, $\overrightarrow{R}(s, v) = get\ s$, because the source completely determines the view; backwards consistency restoration $\overleftarrow{R} = put : S \times V \rightarrow S$ 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 $S \mathrel{\ooalign{$\Rightarrow$\cr\hidewidth$=$\hidewidth}} V$ 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 Sect. 2.4, with both state spaces being lists:

$$\begin{aligned} M &= [\mathit{Name} \times \mathit{Dates} \times \mathit{Nationality}] \\ N &= [\mathit{Name} \times \mathit{Nationality}] \end{aligned}$$

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 $(\mathit{putr}, \mathit{putl}) : A \nleftrightarrow_C B$ between A and B with complements C consists of a pair of functions

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

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

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

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

$$\begin{aligned} &\forall a, a', b, c, c'. \ \mathit{putr}\ (a, c) = (b, c') \Rightarrow \mathit{putr}\ (a', c') = \mathit{putr}\ (a', c) && (\mathsf{PutPutR}) \\ &\forall a, b, b', c, c'. \ \mathit{putl}\ (b, c) = (a, c') \Rightarrow \mathit{putl}\ (b', c') = \mathit{putl}\ (b', c) && (\mathsf{PutPutL}) \end{aligned}$$

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

An asymmetric lens $S \nleftrightarrow V$ 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 $C = [\mathit{Name} \times \mathit{Dates} \times \mathit{Nationality}]$ 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 et al. [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 Fig. 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 et al. have been pioneering a line of work [33–36] to provide a categorical unification and generalization of the ordered and delta-based approaches.

$$\begin{tikzcd}
a \arrow[r, leftrightarrow, "c"] \arrow[d, mapsto, "\delta_A"'] & b \arrow[d, mapsto, "\delta_B"] \\
a' \arrow[r, leftrightarrow, "c'"'] & b'
\end{tikzcd}$$

Fig. 4. Delta-based consistency

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 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 $G : \mathcal{A} \rightarrow \mathcal{B}$, which has objects (A, β) where A is an object of $\mathcal{A}$ and $\beta : G(A) \rightarrow B$ 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 $(G, P) : \mathcal{A} \rightleftarrows \mathcal{B}$ is then a pair in which $G : \mathcal{A} \rightarrow \mathcal{B}$ is a functor, and $P : |G/\mathcal{B}| \rightarrow |\mathcal{A}^2|$ is a function, taking a pair $(A, \beta : G(A) \rightarrow B)$ to a transition $\alpha : A \rightarrow A'$. One should think of the pair $(A, \beta : G(A) \rightarrow B)$ 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 $\beta : G(A) \rightarrow B$ and $\beta' : G(A') \rightarrow B'$, 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 $\beta : G(A) \rightarrow B$ 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 $\mathcal{C} \ntriangleleft \mathcal{A}$ and $\mathcal{C} \ntriangleleft \mathcal{B}$ 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* [25,54] 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 Sect. 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 Fig. 5. Here, the two models are as shown in Fig. 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 Fig. 6, which simply consists of the union of the two metamodels from Fig. 3(a) together with the two correspondence associations.

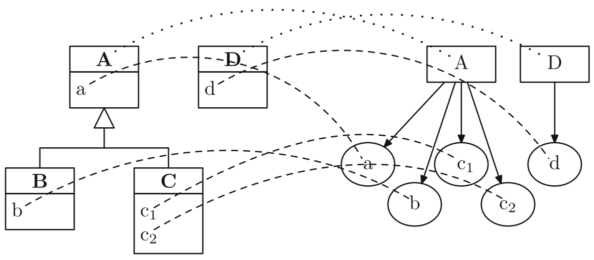


Fig. 5. Two models, with correspondences

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

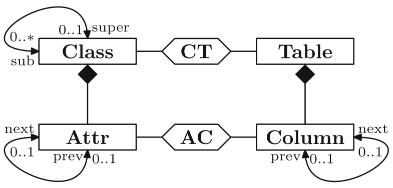


Fig. 6. Two metamodels, with correspondences

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 Fig. 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,

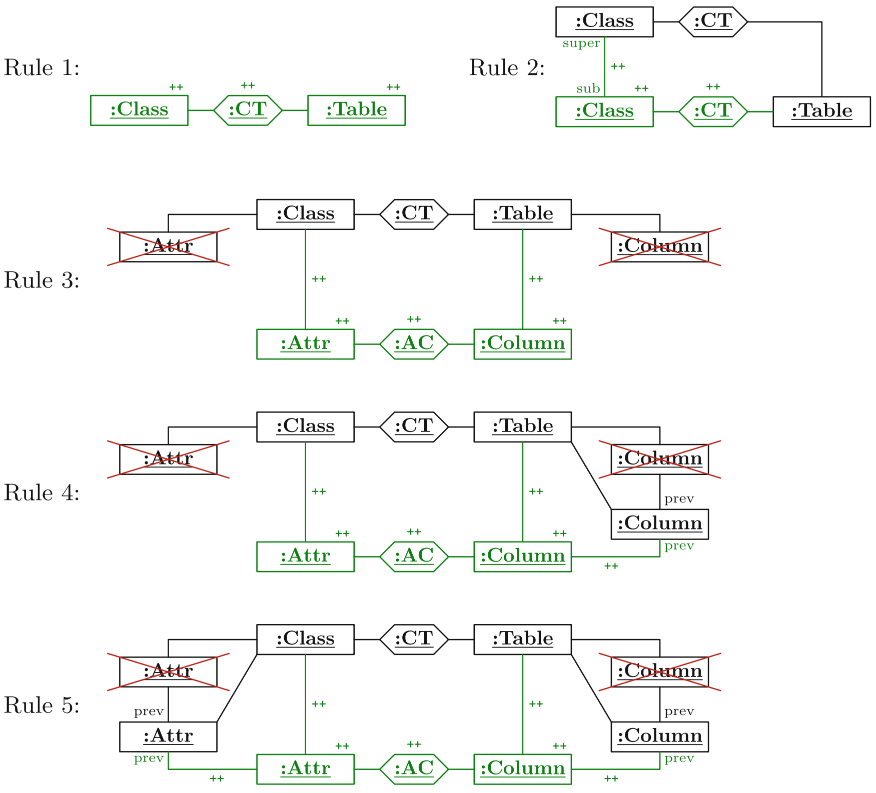


Fig. 7. A triple-graph grammar (Color figure online)

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 [11,63].

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

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

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:

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

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.)

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

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

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

$$\begin{array}{lll} \mathit{get} \gg\!= \mathit{set} & = \mathit{return}\ () & (\mathsf{GetSet}) \\ \mathit{set}\ s_1 \gg \mathit{set}\ s_2 & = \mathit{set}\ s_2 & (\mathsf{SetSet}) \end{array}$$

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:

$$\begin{array}{ll} & \mathit{get}_A \gg\!= \lambda a \rightarrow \mathit{get}_B \gg\!= \lambda b \rightarrow k\ a\ b \\ = & \mathit{get}_B \gg\!= \lambda b \rightarrow \mathit{get}_A \gg\!= \lambda a \rightarrow k\ a\ b \quad (\mathsf{GetComm}) \end{array}$$

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.

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

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.

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 $(\mathit{get}_A, \mathit{set}_A)$ and $(\mathit{get}_B, \mathit{set}_B)$ 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 $\mathit{bx} : A \rightleftharpoons_M B$, and think of bx as a record with four fields $\mathit{get}_A\ \mathit{bx}, \mathit{set}_A\ \mathit{bx}, \mathit{get}_B\ \mathit{bx}, \mathit{set}_B\ \mathit{bx}$. 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.

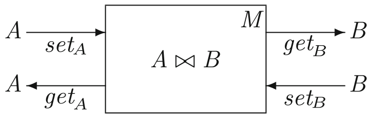


Fig. 8. Monadic BX on sources A, B over monad M

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 $(get, put) : S \rightleftharpoons V$, we can define operations as follows:

$$\begin{aligned}
&get_V &&:: State\ S\ V\\
&get_V &&= get_S \ggg\!= \lambda s \rightarrow return\ (get\ s)\\
&set_V &&:: V \rightarrow State\ S\ ()\\
&set_V\ v &&= get_S \ggg\!= \lambda s \rightarrow set\ (put\ (s, v))
\end{aligned}$$

where get_S:: $State\ S\ S$ and $putS$::$S \rightarrow State\ S\ ()$ 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 $putr :: (A, C) \rightarrow (B, C)$ and $putl :: (B, C) \rightarrow (A, C)$ operations as operations in the state monad, where the state is the complement type C, as follows:

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

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 $State\ S$ where $S = \{(A, B, C) \mid putr\ (A, C) = (B, C)\}$ (which is equal to $\{(A, B, C) \mid putl\ (B, C) = (A, C)\}$ thanks to the symmetric lens laws). Then the get and set operations are as follows:

$$\begin{array}{ll} get_A & :: State\ S\ A \\ get_A & = State\ (\lambda(a,b,c) \to (a,(a,b,c))) \\ set_A & :: A \to State\ C\ () \\ set_A\ a' & = State\ (\lambda(a,b,c) \to \mathbf{let}\ (b',c') = putr\ (a,c)\ \mathbf{in}\ ((),(a',b',c'))) \end{array}$$

and symmetrically for get_B, 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 *State S*, but given two monadic BX of types $A \rightleftharpoons_M B$ and $B \rightleftharpoons_N C$, 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

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

StateT is a standard construction called the *state monad transformer* [38]. Intuitively, $StateT\ M\ S$ 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:

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

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 $bx_1 : A \rightleftharpoons_{M_1} B$ and $bx_2 : B \rightleftharpoons_{M_2} C$ 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:

$$\begin{aligned}
&\mathit{get}_A \quad = \mathit{get} \ggg \lambda(s_1, s_2) \to \mathit{return}\ (\mathit{read}_A\ s_1)\\
&\mathit{set}_A\ a = \mathit{get} \ggg \lambda(s_1, s_2) \to\\
&\qquad\qquad \mathit{runStateT}\ (\mathit{set}_A\ \mathit{bx}_1\ a)\ s_1 \ggg \lambda((), s_1') \to\\
&\qquad\qquad \mathit{runStateT}\ (\mathit{set}_B\ \mathit{bx}_2\ (\mathit{read}_B\ \mathit{bx}_1\ s_1'))\ s_2 \ggg \lambda((), s_2') \to\\
&\qquad\qquad \mathit{set}\ (s_1', s_2')
\end{aligned}$$

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 $\mathit{set}_A\ \mathit{bx}_1$ to the first component, then applies $\mathit{set}_B\ \mathit{bx}_2$ 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 $\mathit{read}_B\ \mathit{bx}_1\ s_1 = \mathit{read}_B\ \mathit{bx}_2\ s_2$. So the composition is actually defined as an monadic BX over $\mathit{StateT}\ (S_1 \bowtie S_2)$, where $(S_1 \bowtie S_2 = \{(s_1, s_2) \mid \mathit{read}_B\ \mathit{bx}_1\ s_1 = \mathit{read}_B\ \mathit{bx}_2\ s_2\}$. 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 Sect. 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 Sect. 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 artificial 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 Sect. 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 [39,49].

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 $(R \otimes S)_{a\,c} =_{\mathrm{def}} \Sigma_{b\,:\,B}\, R_{a\,b} \times S_{b\,c}$; and with 2-cells the *proof-relevant* inclusions $R \subseteq_p S =_{\mathrm{def}} p \,:\, \Pi_{a\,:\,A, b\,:\,B}\, R_{a\,b} \Rightarrow S_{a\,b}$.

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:

$$\mathcal{R} =_{\mathrm{def}} (R, \overrightarrow{R}, \overleftarrow{R}) : \mathcal{A} \not\leftarrow_R \mathcal{B}$$

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

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 $\mathcal{R} : \mathcal{A} \not\leftarrow_R \mathcal{B}$, and 2-cells given by proof-relevant *equivalences*[1] $R \subseteq_p R' \subseteq_q R$ between the underlying consistency relations R, R'.

[1] 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*.

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.

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 [29,31] 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 $R_{a\,b} =_{\mathrm{def}} \{c \in C \mid (a, c, b) \in K\}$. 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 [6,62], 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. Abou-Saleh, F., Cheney, J., Gibbons, J., McKinna, J., Stevens, P.: Notions of bidirectional computation and entangled state monads. In: Hinze, R., Voigtländer, J. (eds.) MPC 2015. LNCS, vol. 9129, pp. 187–214. Springer, Cham (2015). https://doi.org/10.1007/978-3-319-19797-5_9
2. Abou-Saleh, F., Cheney, J., Gibbons, J., McKinna, J., Stevens, P.: Reflections on monadic lenses. In: Lindley, S., McBride, C., Trinder, P., Sannella, D. (eds.) A List of Successes That Can Change the World. LNCS, vol. 9600, pp. 1–31. Springer, Cham (2016). https://doi.org/10.1007/978-3-319-30936-1_1
3. Abou-Saleh, F., McKinna, J., Gibbons, J.: Coalgebraic aspects of bidirectional computation. In: BX 2015. CEUR-WS, vol. 1396, pp. 15–30 (2015)
4. Abou-Saleh, F., McKinna, J., Gibbons, J.: Coalgebraic aspects of bidirectional computation. J. Object Technol. (2017, in press)

5. Ambler, S.W.: Mapping objects to relational databases: O/R mapping in detail (1998). http://www.agiledata.org/essays/mappingObjects.html
6. Angiuli, C., Morehouse, E., Licata, D.R., Harper, R.: Homotopical patch theory. In: International Conference on Functional Programming, pp. 243–256. ACM (2014)
7. Anjorin, A.: Class diagrams to database schemas v0.1. BX Repository. http://bx-community.wikidot.com/examples:classdiagramstodatabaseschemas. Accessed Jan 2017
8. Bancilhon, F., Spyratos, N.: Update semantics of relational views. ACM Trans. Database Syst. **6**(4), 557–575 (1981)
9. Barbosa, D.M.J., Cretin, J., Foster, N., Greenberg, M., Pierce, B.C.: Matching lenses: alignment and view update. In: Hudak, P., Weirich, S. (eds.) International Conference on Functional Programming, pp. 193–204. ACM, New York (2010)
10. Bénabou, J., Davis, R., Dold, A., Isbell, J., MacLane, S., Oberst, U., Roos, J.-E.: Reports of the Midwest Category Seminar. LNM, vol. 47, pp. 1–77. Springer, Heidelberg (1967). https://doi.org/10.1007/BFb0074298
11. Benton, N., Hughes, J., Moggi, E.: Monads and effects. In: Barthe, G., Dybjer, P., Pinto, L., Saraiva, J. (eds.) APPSEM 2000. LNCS, vol. 2395, pp. 42–122. Springer, Heidelberg (2002). https://doi.org/10.1007/3-540-45699-6_2
12. Bohannon, A., Foster, J.N., Pierce, B.C., Pilkiewicz, A., Schmitt, A.: Boomerang: resourceful lenses for string data. In: Principles of Programming Languages (2008)
13. Bohannon, A., Pierce, B.C., Vaughan, J.A.: Relational lenses: a language for updatable views. In: Principles of Database Systems, pp. 338–347. ACM (2006)
14. Boyer, J.M.: W3C XForms, October 2009. https://www.w3.org/TR/xforms/
15. Brady, E.: Type-Driven Development with Idris. Manning Publications, Shelter Island (2017)
16. Cai, Y., Giarrusso, P.G., Rendel, T., Ostermann, K.: A theory of changes for higher-order languages: incrementalizing λ-calculi by static differentiation. In: Programming Language Design and Implementation, pp. 145–155. ACM (2014)
17. Cheney, J., Gibbons, J., McKinna, J., Stevens, P.: On principles of least change and least surprise for bidirectional transformations. J. Object Technol. (2017, to appear)
18. Cheney, J., McKinna, J., Stevens, P., Gibbons, J.: Towards a repository of BX examples. In: BX Workshop, March 2014
19. Cicchetti, A., Di Ruscio, D., Eramo, R., Pierantonio, A.: JTL: a bidirectional and change propagating transformation language. In: Malloy, B., Staab, S., van den Brand, M. (eds.) SLE 2010. LNCS, vol. 6563, pp. 183–202. Springer, Heidelberg (2011). https://doi.org/10.1007/978-3-642-19440-5_11
20. Cooper, E., Lindley, S., Wadler, P., Yallop, J.: The essence of form abstraction. In: Ramalingam, G. (ed.) APLAS 2008. LNCS, vol. 5356, pp. 205–220. Springer, Heidelberg (2008). https://doi.org/10.1007/978-3-540-89330-1_15
21. Czarnecki, K., Foster, J.N., Hu, Z., Lämmel, R., Schürr, A., Terwilliger, J.F.: Bidirectional transformations: a cross-discipline perspective. In: Paige, R.F. (ed.) ICMT 2009. LNCS, vol. 5563, pp. 260–283. Springer, Heidelberg (2009). https://doi.org/10.1007/978-3-642-02408-5_19
22. Date, C.J.: View Updating and Relational Theory. O'Reilly, Newton (2012)
23. Diskin, Z., Xiong, Y., Czarnecki, K.: From state- to delta-based bidirectional model transformations: the asymmetric case. J. Object Technol. **10**(6), 1–25 (2011)
24. Diskin, Z., Xiong, Y., Czarnecki, K., Ehrig, H., Hermann, F., Orejas, F.: From state- to delta-based bidirectional model transformations: the symmetric case. In: Whittle, J., Clark, T., Kühne, T. (eds.) MODELS 2011. LNCS, vol. 6981, pp. 304–318. Springer, Heidelberg (2011). https://doi.org/10.1007/978-3-642-24485-8_22

25. Ehrig, H., Engels, G., Kreowski, H.-J., Rozenberg, G. (eds.): Handbook of Graph Grammars and Computing by Graph Transformation. Applications, Languages and Tools, vol. 2. World Scientific, Singapore (1999)
26. Foster, J.N., Greenwald, M.B., Moore, J.T., Pierce, B.C., Schmitt, A.: combinators for bidirectional tree transformations: a linguistic approach to the view update problem. In: Principles of Programming Languages, pp. 233–246. ACM (2005)
27. Foster, N., Matsuda, K., Voigtländer, J.: Three complementary approaches to bidirectional programming. In: Gibbons, J. (ed.) Generic and Indexed Programming. LNCS, vol. 7470, pp. 1–46. Springer, Heidelberg (2012). https://doi.org/10.1007/978-3-642-32202-0_1
28. Hegner, S.J.: An order-based theory of updates for closed database views. Ann. Math. Artif. Intell. **40**(1–2), 63–125 (2004)
29. Hofmann, M.: Modular edit lenses. In: Gibbons, J., Stevens, P. (eds.) Summer School on Bidirectional Transformations. LNCS, vol. 9715, pp. 73–99. Springer, Cham (2018)
30. Hofmann, M., Pierce, B.C., Wagner, D.: Symmetric lenses. In: Ball, T., Sagiv, M. (eds.) Principles of Programming Languages, pp. 371–384. ACM, New York (2011)
31. Hofmann, M., Pierce, B.C., Wagner, D.: Edit lenses. In: Field, J., Hicks, M. (eds.) Principles of Programming Languages, pp. 495–508. ACM, New York (2012)
32. Hu, Z., Schürr, A., Stevens, P., Terwilliger, J.F.: Bidirectional transformations "bx" (Dagstuhl Seminar 11031). Dagstuhl Rep. **1**(1), 42–67 (2011)
33. Johnson, M., Rosebrugh, R.D.: Lens put-put laws: monotonic and mixed. In: BX Workshop (2012). ECEASST **49**
34. Johnson, M., Rosebrugh, R.D.: Spans of lenses. In: Terwilliger, J., Hidaka, S. (eds.) BX Workshop. CEUR Workshop Proceedings, vol. 1133, pp. 112–118 (2014). CEUR-WS.org
35. Johnson, M., Rosebrugh, R.D.: Unifying set-based, delta-based and edit-based lenses. In: Anjorin, A., Gibbons, J. (eds.) BX Workshop. CEUR Workshop Proceedings, vol. 1571, pp. 1–13 (2016). CEUR-WS.org
36. Johnson, M., Rosebrugh, R.D., Wood, R.J.: Lenses, fibrations and universal translations. Math. Struct. Comput. Sci. **22**(1), 25–42 (2012)
37. Peyton Jones, S.L., Wadler, P.: Imperative functional programming. In: Principles of Programming Languages, pp. 71–84 (1993)
38. Liang, S., Hudak, P., Jones, M.: Monad transformers and modular interpreters. In: Principles of Programming Languages, pp. 333–343 (1995)
39. Luo, Z.: Computation and Reasoning: A Type Theory for Computer Science. International Series of Monographs on Computer Science. Oxford University Press, Oxford (1994)
40. Macedo, N., Cunha, A.: Least-change bidirectional model transformation with QVT-R and ATL. Softw. Syst. Model. **15**(3), 783–810 (2016)
41. McKinna, J.: Bidirectional transformations are proof-relevant bisimulations. Extended Abstract Presented at ICFP Workshop TyDe, Nara, Japan (2016). https://www.youtube.com/watch?v=33RYwcIQ7UM
42. McKinna, J.: 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. McKinna, J.: Complements witness consistency. In: Bx Workshop, ETAPS (2016). http://ceur-ws.org/Vol-1571/paper_10.pdf
44. Meertens, L.: Designing constraint maintainers for user interaction. CWI, Amsterdam, June 1998. http://www.kestrel.edu/home/people/meertens/pub/dcm.ps

45. Meertens, L.: Designing constraint maintainers for user interaction. In: Mu, S.-C. (ed.) Third Workshop on Programmable Structured Documents, pp. 1–3. PSD Laboratory, Tokyo University (2005)
46. Microsoft. Windows Presentation Foundation (2006). https://msdn.microsoft.com/en-us/library/ms754130.aspx
47. Moggi, E.: Notions of computation and monads. Inf. Comput. **93**(1), 55–92 (1991)
48. Neward, T.: The Vietnam of computer science, June 2006. http://blogs.tedneward.com/post/the-vietnam-of-computer-science/
49. Nordström, B., Petersson, K., Smith, J.: Programming in Martin-Löf's Type Theory: An Introduction. International Series of Monographs on Computer Science, vol. 7. Oxford University Press, Oxford (1990). https://www.cse.chalmers.se/research/group/logic/book/book.pdf
50. Norell, U.: Dependently typed programming in agda. In: Koopman, P., Plasmeijer, R., Swierstra, D. (eds.) AFP 2008. LNCS, vol. 5832, pp. 230–266. Springer, Heidelberg (2009). https://doi.org/10.1007/978-3-642-04652-0_5
51. Pacheco, H., Hu, Z., Fischer, S.: Monadic combinators for "putback" style bidirectional programming. In: Partial Evaluation and Program Manipulation, pp. 39–50. ACM (2014)
52. Plotkin, G., Power, J.: Notions of computation determine monads. In: Nielsen, M., Engberg, U. (eds.) FoSSaCS 2002. LNCS, vol. 2303, pp. 342–356. Springer, Heidelberg (2002). https://doi.org/10.1007/3-540-45931-6_24
53. Pratt, T.W.: Pair grammars, graph languages and string-to-graph translations. J. Comput. Syst. Sci. **5**(6), 560–595 (1971)
54. Rozenberg, G. (ed.): Handbook of Graph Grammars and Computing by Graph Transformation. Foundations, vol. 1. World Scientific, Singapore (1997)
55. Schürr, A.: Specification of graph translators with triple graph grammars. In: Mayr, E.W., Schmidt, G., Tinhofer, G. (eds.) WG 1994. LNCS, vol. 903, pp. 151–163. Springer, Heidelberg (1995). https://doi.org/10.1007/3-540-59071-4_45
56. Schürr, A., Klar, F.: 15 years of triple graph grammars. In: Ehrig, H., Heckel, R., Rozenberg, G., Taentzer, G. (eds.) ICGT 2008. LNCS, vol. 5214, pp. 411–425. Springer, Heidelberg (2008). https://doi.org/10.1007/978-3-540-87405-8_28
57. Shkaravska, O.: Side-effect monad, its equational theory and applications (2005). http://www.ioc.ee/~tarmo/tsem05/shkaravska1512-slides.pdf
58. Stevens, P.: ModelTests v0.1 in Bx Examples Repository. http://bx-community.wikidot.com/examples:modeltests. Accessed 6 Feb 2017
59. Stevens, P.: Bidirectional model transformations in QVT: semantic issues and open questions. Softw. Syst. Model. **9**(1), 7–20 (2010)
60. Stevens, P.: Observations relating to the equivalences induced on model sets by bidirectional transformations. In: BX Workshop (2012). ECEASST **49**
61. Stevens, P., McKinna, J., Cheney, J.: Composers v0.1. BX Repository. http://bx-community.wikidot.com/examples:composers. Accessed Jan 2017
62. Swierstra, W., Löh, A.: The semantics of version control. In: Onward!, pp. 43–54 (2014)
63. Wadler, P.: Monads for functional programming. In: Jeuring, J., Meijer, E. (eds.) AFP 1995. LNCS, vol. 925, pp. 24–52. Springer, Heidelberg (1995). https://doi.org/10.1007/3-540-59451-5_2

An Introduction to Triple Graph Grammars as an Implementation of the Delta-Lens Framework

Anthony Anjorin(✉)

University of Paderborn, Paderborn, Germany
anthony.anjorin@upb.de

Abstract. Triple Graph Grammars (TGGs) provide a rule-based means of specifying a consistency relation over two graph languages, with correspondences between elements in the two different languages represented explicitly as a third "traceability" graph. Many useful tools can be derived automatically from a TGG including incrementally working synchronisers, which are able to realise forward and backward change propagation without incurring unnecessary information loss. TGGs are typically introduced based on the algebraic graph transformation framework, which is not particularly accessible to many members of the bidirectional transformation (bx) community, who are often more familiar with some variant of the lens framework as a theoretical foundation for bx.

This chapter, therefore, provides a self-contained, relatively gentle and tutorial-like introduction to TGGs as a pragmatic implementation of the symmetric delta-lens (sd-lens) framework proposed by Diskin et al., thereby mapping abstract and general terms used in the sd-lens framework such as *models* and *deltas*, to concrete realisations in the TGG framework such as *typed graphs* and *spans of typed graphs*.

Keywords: Bidirectional transformation · Triple Graph Grammars
Symmetric delta-lenses

1 Introduction and Motivation

Triple Graph Grammars (TGGs) were first suggested by Schürr [31] as a means of specifying *incremental synchronisers*, i.e., derived operational programs that are able to propagate changes made in one artefact to corresponding changes in another existing artefact. TGGs follow a "consistency first" approach, meaning that a user is expected to supply a direction-agnostic definition of a consistency relation, from which everything else required for consistency management is automatically derived. The style of consistency specification with TGGs generalises the idea from Pratt [28] of coupling a string grammar and a graph grammar as a *pair grammar*, which can be used to derive string-to-graph translations. TGGs are essentially "pair grammars" but (i) are specified uniformly over graph grammars, and (ii) additionally comprise a third, explicit correspondence graph grammar to capture the pairing of a source and target graph grammar in more detail. Graph grammars are a generalisation of string grammars and have been studied intensively in the graph transformation community for over 40 years. The interested reader is referred to Ehrig et al. [15] for a comprehensive introduction and overview of the algebraic graph transformation approach.

The *bidirectional transformation* (bx) community was formed with the goal of fostering collaboration and the exchange of ideas between the various, diverse groups working on all aspects of consistency management [11]. Domains represented in bx include databases (view update problem, schema evolution), software engineering (model synchronisation, conformance checking), and programming language development (bidirectionalisation, coupled transformations).

The desire for a common conceptual bx framework has led to a substantial amount of work on multiple variants of the so-called *lens* framework. The interested reader is referred to Johnson and Rosebrugh [22] for a unifying overview of various types of lenses. A lens is essentially a pair of forward and backward synchronisers, together with a formal characterisation of properties that the pair must possess in order to be "well-behaved", i.e., exhibit desirable synchronisation behavior (depending on the application scenario and domain) such as termination, confluence, non-determinism, and compliance with round trip laws.

The standard introduction to and formalization of TGGs precedes the notion of lenses and is typically presented by and for the graph transformation community. While the fundamental work of Ehrig et al. [14] and Schürr [31] is certainly elegant and in many ways a natural fit for TGGs, at least two pain points can be identified: Firstly, the bx community appears to have converged more towards lenses (with all variants) as a common and shared formal framework. This means that laws are often formulated in a round trip manner, with a focus more on the synchronisers (the programs or functions performing consistency restoration) and less on the underlying notion of consistency. This makes it difficult to somehow "transform" the round trip laws to corresponding properties in the TGG framework. Secondly, most papers on TGGs fix the central notions of a graph and a graph arrow (morphism) and thread this through all definitions, results, and constructions. While this typically results in a constructive theory that can almost be directly implemented, it also requires assumptions that are intentionally avoided in the lens framework. As a consequence, although the TGG framework could be instantiated for strings, lists, or trees, this is neither obvious nor trivial.

To address these challenges and more, there has already been fruitful collaboration between research groups working on lenses and TGGs. The joint work of Diskin, Hermann, and others has led to the *symmetric delta lens* (sd-lens) framework [13], and a novel presentation of TGGs as an implementation of the sd-lens framework [17,18].

This chapter attempts to complement the work of Hermann et al. [17,18] with a detailed and simplified unification of TGGs and lenses. In the spirit of a tutorial, the focus of this chapter is more on intuition and understanding, and less on challenging technical details that are partly presented in a rather compact fashion by Hermann et al. [17,18]. The main contribution is to present TGGs more as a general platform, both formal and practical, for realising *various* formal bx frameworks, and less as a strict implementation of the sd-lens framework. Working out all details for the sd-lens framework requires numerous restrictions to TGGs, which are discussed in full detail by Hermann et al. [17,18]. This might actually indicate that some extensions of the lens framework are still required to achieve a perfect match for TGGs. An example for this is non-determinism – natural for graph transformation and thus TGGs – but excluded in the current sd-lens framework.

This chapter aims to provide an introduction to TGGs that is especially accessible to the bx community, taking inspiration from the lenses framework to provide an alternative perspective on TGGs that fits better to the other chapters of this book. The primary target audience are thus readers with a (basic) background in bx (provided by the introductory chapter of this book) but neither in graph transformation nor TGGs. TGGs are often applied in a Model-Driven Engineering (MDE) context where the central concept of a *model* fits quite well to that of a typed, attributed graph. The simple but illustrative running example used consequently throughout the chapter will be taken loosely from the MDE domain and introduced using basic elements from the UML class and object diagram visual notation. Very basic category theory will be used throughout the chapter as a uniform, underlying organisational structure, which is also able to unify and clarify connections between concepts from both the sd-lens framework and TGGs. While the reader does not require prior knowledge of category theory to read and understand this chapter, some patience is necessary to get used to and appreciate the predominantly visual notation and style, as this might be unusual for the reader.

The rest of the chapter is structured as follows: Sect. 2 handles the *data* to be synchronised and operated on. To clarify similarities and differences between the sd-lens and TGG framework, the discussion is presented on two levels: (i) on the more abstract level of the sd-lens framework, and (ii) on the level of TGGs, representing an executable instantiation of the sd-lens framework.

In Sect. 3, TGGs are introduced as a pragmatic specification language for symmetric delta lenses. Section 4 discusses formal properties and guarantees, focusing on guaranteeing *least change* as a current challenge for TGG-based synchronisation. Section 5 concludes the chapter with a brief overview of related work and an outlook on future work.

2 Model Spaces and Triple Spaces

As depicted in Fig. 1 using basic UML class diagram notation, we shall start by reviewing definitions from Diskin et al. [13] for a *triple space* connecting a source and target *model space* with correspondence links (or just *corrs*). Model spaces consist of *models* and *deltas*, representing all possible states of the artefacts to be synchronised and all possible updates on these states, respectively.

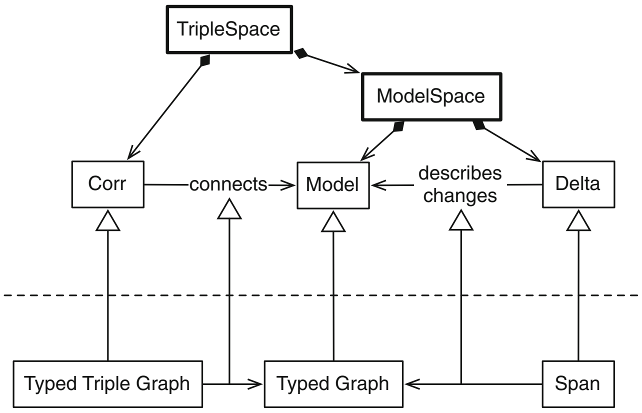


Fig. 1. Overview of important concepts and relations

Following Hermann et al. [18], we shall then refine these abstract concepts in a TGG context (depicted in Fig. 1 below the dashed horizontal line) by introducing *typed graphs* as models, *spans* of typed graphs as deltas, and *typed triple graphs* as corrs. These concepts are concrete enough to be almost directly mapped to an implementation framework, which is exactly what is done in most TGG-based tools [21,26]. To avoid getting lost and ending up not seeing the various different graphs for the arrows, we shall use very basic concepts from category theory as a uniform, underlying organisational structure. The most basic of such concepts is that of a *category*, consisting of things (objects) and connections (arrows). We expect to be able to compose arrows in a manner that is independent of the order in which this is done (associative composition), and to have a unique "idle" arrow for every object (an identity arrow). This basic structure is formalised by the following definition (taken from Awodey [8] and Ehrig et al. [15]):

Definition 1 (Category). *A* category $\mathbf{C} = (Ob, Arr, \,;\, , id)$ *consists of:*

- *a class Ob of* objects*,*
- *for each pair of objects $A, B \in Ob$, a class $Arr_{(A,B)}$ of* arrows*, where $f \in Arr_{(A,B)}$ is denoted by $f : A \to B$,*
- *for all objects $A, B, C \in Ob$, a binary operation (for composing arrows): $; : Arr_{(A,B)} \times Arr_{(B,C)} \to Arr_{(A,C)}$,*
- *for each object $A \in Ob$, an identity arrow $id_A : A \to A$,*

such that the following conditions hold:

- Associativity: $\forall A, B, C, D \in Ob.\ \forall f : A \to B, g : B \to C, h : C \to D.\ f \,;(g\,;h) = (f\,;g)\,;h.$
- Identity: $\forall A, B \in Ob.\ \forall f : A \to B.\ (id_A\,;f = f) \wedge (f\,;id_B = f).$

A useful and familiar category is **Sets**, consisting of sets as objects and total functions as arrows. Apart from serving as a concrete example for Definition 1, we shall also build-up our more complex structures based on **Sets**.

Definition 2 (Sets and Total Functions). ***Sets*** $= (Ob, Arr, ;, id)$ *consists of:*

- *sets* Ob, *total functions* Arr,
- *for* $A, B, C \in Ob$, $f : A \to B$, $g : B \to C$, $(f ; g) : A \to C$ *is defined as* $\forall x \in A.\ (f ; g)(x) := g(f(x))$,
- *for* $A \in Ob_{Sets}$, $id_A : A \to A$ *is defined as* $\forall x \in A.\ id_A(x) := x$.

Fact 1 (Sets is a Category). ***Sets*** *according to Definition 2 is a category according to Definition 1.*

Proof (Sketch). Follows directly from associativity of function composition and the definition of *id* for **Sets** (see Awodey [8] for further details).

Now that we have introduced the basic structure of a category, we are ready to define the first central concept for this chapter, a *model space*. A model space is essentially a *small* category, i.e., a category in which the class of all objects and class of all arrows are sets. While this is not absolutely relevant for this chapter, the interested reader is invited to look up Russell's paradox and find out why the "set of all sets" cannot be a set and must be a "proper class".

Thinking in terms of data or artefacts (to be later synchronised with other artefacts), a model space captures (i) all possible states (let us refer to these as *models*) of the artefact we wish to admit, and (ii) how we can transition from one state of an artefact to another state by applying suitable updates (let us refer to these transitions as *deltas*). In addition to the basic categorical structure from Definition 1, with objects as models and arrows as deltas, we also expect to be able to invert every delta. These expectations are formalised by the following definition:

Definition 3 (Model Space). *A* model space $\mathcal{M} = (M, \Delta, ;, id, inv)$ *is a category with a set* M *of objects referred to as* models, *a set* Δ *of arrows referred to as* deltas,[1] *and a total function* $inv : \Delta \to \Delta$, *assigning to every delta* $\delta : m \to m'$ *an inverse delta* $inv(\delta) : m' \to m$, *such that for every identity arrow* $id_m : m \to m \in \Delta$ $[\ inv(id_m) = id_m\ ]$ *and for every arrow* $\delta \in \Delta$ $[\ inv(inv(\delta)) = \delta\ ]$.

Example. As a running example used consequently throughout this chapter, we shall investigate a (greatly simplified and thus purposely unrealistic) synchronisation task taken from the domain of medical software (loosely inspired by Weber et al. [35]). Depicted in Fig. 2, the example is centred around a patient dashboard application ❶. To address different goals, different abstractions (models) of this system have been established ❷. These models live in their respective model spaces (depicted as coloured circles in Fig. 2) describing how they can be updated. To the left of Fig. 2, a doctor ❸ is interested in patient information, prescribed medication, and the physician currently assigned to each patient. To the right of Fig. 2, an expert in charge of logistics ❹ ensures that the hospital has adequate and uninterrupted supply of medication, and is only interested in the "dosage plan" of the hospital, i.e., a collection of all current prescriptions.

[1] Subscripts on Δ are omitted if all arrows are meant.

To simplify logistics and allow for optimisation, doctors prescribe medication only via so-called “generic” names, e.g., Aspirin. The actual brand, e.g., Ascriptin, can then be decided by the logistic expert based on the current price situation or what is already in stock. This means that the model spaces have both shared and private data: the hospital does not have to (and probably does not *want* to!) share information about patients and doctors ❺; analogously the logistic company wants to be able to change the mapping of generic to brand medication names in order to optimise the supply process ❻. Both experts, however, must synchronise on their shared data, comprising the current prescriptions ❼. The challenge here is to propagate changes made in one artefact to the other artefact automatically, ensuring that shared data is kept consistent, and that no data is lost when round tripping.

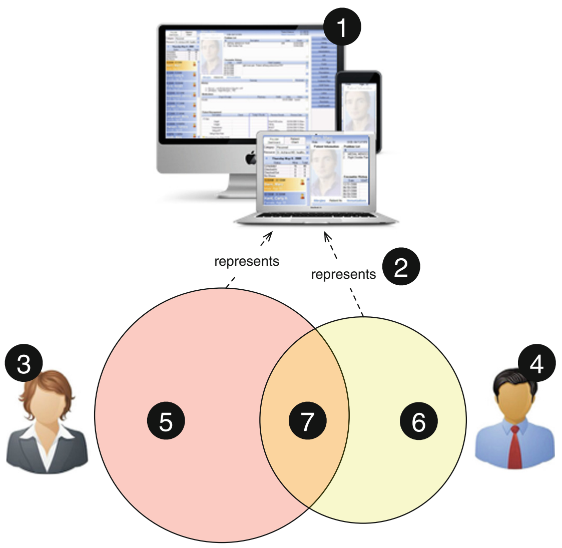


Fig. 2. Our running example from the domain of medical software (Color figure online)

The concept of a model space is abstract in the sense that almost no assumptions are made about what models and deltas actually constitute. Depending on the problem domain and/or technological space, a concrete framework or tool will of course “instantiate” these concepts appropriately. Models can be lists, trees, tables, functions, collections of constraints, or graphs. In an MDE context, in which most TGG tools are used today, a *model* is often a typed, attributed graph. To keep all definitions simple, we shall omit details of attribution and other advanced typing features such as node/edge inheritance, composition/aggregation, multiplicities and other constraints used to further shape the model space. The interested reader is referred to Biermann et al. [9] and Ehrig et al. [15] for further details.

Depicted in Fig. 3, graphs are shaped from sets (edges and nodes) and set arrows (connecting edges to their source and target nodes). This induces a categorical structure with graph arrows as pairs of set arrows, which are structure preserving, i.e., respect the way edges are connected to nodes in a graph by mapping source and target nodes together with edges. Note how one can "zoom" out from **Sets** to **Graphs** by changing the underlying category of the diagram.[2] To the right of Fig. 3, the diagram in **Graphs** depicts graph objects and graph arrows as atomic entities with hidden internal structure. This is a useful and powerful abstraction mechanism, which will become even more important when dealing uniformly with more complex "objects" and "arrows".

$$\begin{tikzcd}
E \arrow[r, bend left, "src"] \arrow[r, bend right, "trg"'] \arrow[d, "f_E"'] & V \arrow[d, "f_V"'] \\
E' \arrow[r, bend left, "src'"] \arrow[r, bend right, "trg'"'] & V'
\end{tikzcd}$$

Sets

$$\begin{tikzcd}
G \arrow[d, "f"'] \\
G'
\end{tikzcd}$$

Graphs

Fig. 3. Graphs and graph arrows

Typing structure can be introduced by choosing a graph as a *type graph*, and then demanding that every well-typed graph have a graph arrow *type* that maps its nodes and edges to nodes and edges in the type graph. This induces again a categorical structure with these "type arrows" as objects and type preserving graph arrows, i.e., nodes/edges are mapped to nodes/edges of the same type, as arrows. This is depicted in Fig. 4, demonstrating again the "zooming out" transition from a diagram in **Graphs** to a diagram in **TGraphs**.

The following definition formalises the preceding discussion, introducing typed graphs (with their corresponding categorical structure) as a concrete instantiation of the abstract concept of a model. Note that the definition is concrete enough to be implemented directly in a standard programming language (and this is basically what is done for TGG-based tools).

[2] The formal diagrams used in this chapter are placed in a frame with a label in the right-bottom corner denoting how to interpret the objects and arrows in the diagram. For example, a diagram with label **Sets** means that objects in the diagram are sets, and arrows in the diagram are total functions. All other diagrams are informal and require a diagram-specific legend or extra explanation.

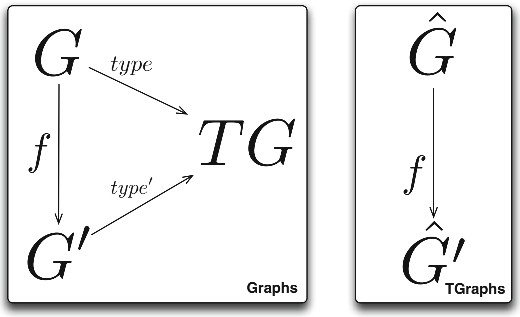


Fig. 4. Typed graphs and typed graph arrows

Definition 4 (Models as Typed Graphs). *A* graph G *consists of: (i) A set E of edges. (ii) A set V of nodes (vertices). (iii) Total functions $src : E \to V$ and $trg : E \to V$, assigning every edge a source and target node, respectively.*
A graph arrow *f from graph $G = (E, V, src, trg)$ to graph $G' = (E', V', src', trg')$, denoted as $f : G \to G'$, consists of a pair of total functions $f_E : E \to E'$ and $f_V : V \to V'$ such that $f_E \,;src' = src \,; f_V$ and $trg \,; f_V = f_E \,; trg'$.*
A type graph *is a distinguished graph TG. A* typed graph *$\hat{G} = (G, type)$ consists of a graph G, and a graph arrow $type : G \to TG$.*
A typed graph arrow *$f : \hat{G} \to \hat{G}'$ is a graph arrow $f : G \to G'$ such that $type = f \,; type'$, where $\hat{G} = (G, type), \hat{G}' = (G', type')$.*

Example. A source type graph TG_S for our running example is depicted in Fig. 5 together with a source typed graph G_S. Let us refer to this source "domain" in the rest of the chapter as *MediSoft*. For the "internals" of the graphs, a simplified UML-like syntax for class and object diagrams is used. Nodes are represented as rectangles, edges as arrows. The internal mapping of $type_S$ is indicated by denoting the types of nodes in G_S as `id:Type`, where `Type` is a node in TG_S. Edges in G_S are mapped to edges in TG_S with the same label (identifiers for edges are only used if this is absolutely necessary).

According to TG_S, a hospital has doctors, patients, and different pharmaceuticals such as Aspirin and Ibuprofen. Doctors can be assigned to patients, and patients can be prescribed pharmaceuticals. The concrete hospital `h` in G_S has only one patient `p`, one doctor `d`, who is assigned to `p`, and one pharmaceutical `a` (aspirin). Note that `p` has *not* been prescribed `a`.

The following fact ensures that graphs and typed graphs are indeed categories with objects, arrows, composition, and identity as discussed informally in the preceding sections.

Fact 2 (Graphs and TGraphs are Categories). ***Graphs*** $= (Ob, Arr, \,;, id)$ *consisting of:*

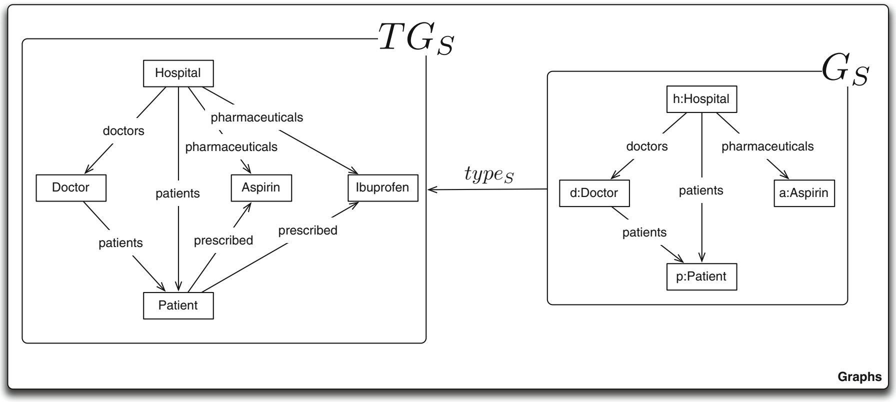


Fig. 5. Source type graph and source typed graph

- *graphs Ob, graph arrows Arr,*
- *for $G, G', G'' \in Ob$, $f = G \to G'$, $g = G' \to G'' \in Arr$, $;(f, g)$ is defined as $f\,;g := (f_V\,;g_V, f_E\,;g_E)$,*
- *for $G = (V, E, src, trg) \in Ob$, $id_G : G \to G$ is defined as $id_G := (id_V, id_E)$.*

*and **TGraphs** $= (Ob, Arr, ;, id)$ consisting of:*

- *typed graphs Ob, typed graph arrows Arr,*
- *$;$ and id taken from **Graphs**.*

are categories according to Definition 1.

Proof (Sketch). **Graphs** and **TGraphs** can be constructed as "comma categories" of **Sets**. This not only ensures that they are indeed again categories but also that they inherit certain properties from **Sets**, which will be useful later in the chapter. The reader is referred to Ehrig et al. [15] for further details.

After introducing typed graphs as models, let us now turn to deltas describing the results of applying a series of updates to a typed graph. To represent a delta $\delta : G \to G'$ between typed graphs G and G', a so-called *span* of typed graphs can be used to record which elements are to be deleted and created. The basic idea is to establish an additional typed graph $\overline{G}$ consisting of all elements that are retained (unchanged) by the delta. Two typed graph arrows can then be used to embed $\overline{G}$ in G and G', representing deletion and creation, respectively. This is formalised by the following definition:

Definition 5 (Deltas as Spans of Typed Graphs). *A* span of typed graphs *$G \xleftarrow{\delta^-} \overline{G} \xrightarrow{\delta^+} G'$ consists of three typed graphs $G, \overline{G}, G'$ and two typed graph arrows $\delta^- : \overline{G} \to G$ and $\delta^+ : \overline{G} \to G'$, where δ^- and δ^+ are monomorphisms (injective functions on nodes and edges).*

Such a span can be interpreted as a delta $\delta : G \rightarrow G'$ (according to Definition 3) by regarding G as the source of the delta, and G' as the target of the delta. Edges (and nodes) in $E_G(, V_G)$ that are not in the co-domain of $\delta^-_E(, \delta^-_V)$ are said to be "deleted" by the delta, while edges (and nodes) in $E_{G'}(, V_{G'})$ that are not in the co-domain of $\delta^+_E(, \delta^+_V)$ are said to be "created" by the delta δ.

Example. A source delta is depicted in Fig. 6 for our running example, representing the change of a prescription from `a` (Aspirin) to `i` (Ibuprofen) for `p` (a patient in the hospital). This is represented formally by the depicted span, where the `prescribed` edge from `p` to `a` is in G_S but not in $\overline{G}_S$ (and is therefore "deleted" by the delta), while the `prescribed` edge from `p` to `i` and `i` itself as a new pharmaceutical in the hospital are in G'_S but not in $\overline{G}_S$ (and are therefore "created" by the delta).

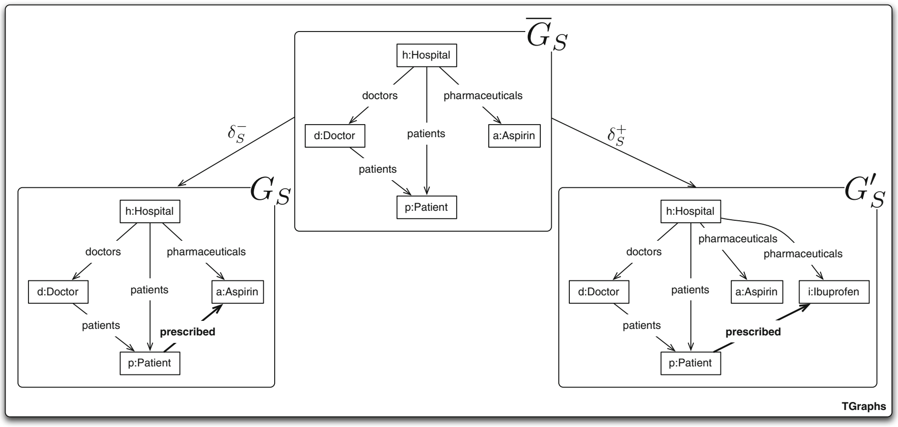


Fig. 6. A source delta as a span of typed graphs

To establish model spaces of typed graphs as models and spans of typed graphs as deltas, we still need to define what composition of deltas means for spans. This can be done categorically (i.e., in terms of arrows and objects and not of their internal structure) via generalised notions of union (and intersection) of objects. Taking the union (intersection) of two objects B and C *without* inspecting their contents only makes sense if we additionally know which parts of B and C are to be regarded as being the same. This extra information is provided by two arrows embedding the same object A in B and C (embedding B and C in the same object A), so that the union (intersection) $\{D, b, d\}$ can be defined with respect to these two arrows. Note that the concepts of a generalised union, called a *pushout*,[3] and a generalised intersection, called a *pullback*,[4] are dual to each other, i.e., are the same up to "flipping all arrows" as can be seen in

[3] As d and b can be seen as being derived by "pushing out" a and c along each other.
[4] As d and b can be seen as being derived by "pulling back" a and c along each other.

Fig. 7. This is, however, not directly relevant for this chapter and is not treated further. The interested reader is referred to Awodey [8] for all details.

$$\begin{tikzcd}
A \arrow[r, "a"] \arrow[d, "c"'] \arrow[dr, phantom, "PO"] & B \arrow[d, dashed, "b"] \arrow[ddr, dashed, bend left, "b'"] & \\
C \arrow[r, dashed, "d"'] \arrow[drr, dashed, bend right, "d'"'] & D \arrow[dr, dashed, "x"] & \\
 & & D'
\end{tikzcd}
\qquad
\begin{tikzcd}
A \arrow[dr, phantom, "PB"] & B \arrow[l, "a"'] & \\
C \arrow[u, "c"] & D \arrow[l, dashed, "d"] \arrow[u, dashed, "b"'] & \\
 & & D' \arrow[ul, dashed, "x"'] \arrow[ull, dashed, bend left, "d'"] \arrow[uul, dashed, bend right, "b'"']
\end{tikzcd}$$

Fig. 7. Pushouts and pullbacks

The following definition formalises pushouts (and pullbacks) in a declarative manner, i.e., *the* pushout (pullback) is characterised precisely via a so-called universal property (Condition (2) in Definition 6), stating that there exists no "better" candidate (that also fulfils Condition (1) in Definition 6). Note that the definition says nothing about how to define, let alone construct, pushouts (and pullbacks) for concrete categories such as **Sets**, **Graphs**, or **TGraphs**.

Definition 6 (Pushouts and Pullbacks). *Let* $\mathbf{C} = (Ob, Arr, \,;\, , id)$ *be a category. Given arrows* $a \in Arr : A \to B$ *and* $c \in Arr : A \to C$*, a* pushout (D, d, b) *over* a *and* c *is defined by:*

- *a* pushout object $D \in Ob$*, and*
- *morphisms* $d : C \to D, b : B \to D$*,*

where:

1. *the "pushout square" commutes, i.e.,* $a; b = c; d$*, and*
2. *the following universal property is fulfilled:*
   $\forall\, D' \in Ob \;\forall\, d' : C \to D' \;\forall\, b' : B \to D'$
   $(a; b' = c; d') \Rightarrow (\exists!\, x : D \to D' \,[(b; x = b') \wedge (d; x = d')])$.

The category $\mathbf{C}$ *is said to* have pushouts *if pushout* (D, d, b) *always exists.*

Given arrows $a \in Arr : B \to A$ *and* $c \in Arr : C \to A$*, a* pullback (D, d, b) *over* a *and* c *is defined by:*

- *a* pullback object $D \in Ob$*, and*
- *morphisms* $d : D \to C, b : D \to B$*,*

where:

1. *the "pullback square" commutes, i.e.,* $b; a = d; c$*, and*
2. *the following universal property is fulfilled:*
   $\forall\, D' \in Ob \;\forall\, d' : D' \to C \;\forall\, b' : D' \to B$
   $(b'; a = d'; c) \Rightarrow (\exists!\, x : D' \to D \,[(x; b = b') \wedge (x; d = d')])$.

The category $\mathbf{C}$ *is said to* have pullbacks *if pullback* (D, d, b) *always exists.*

The following fact guarantees that we can work with pushouts and pullbacks in **Graphs** and **TGraphs**, as is required in the rest of the chapter.

Fact 3 (Pushouts and Pullbacks in Sets, Graphs and TGraphs). *The categories **Sets**, **Graphs**, and **TGraphs** have pushouts and pullbacks.*

Proof (Sketch). The interested reader is referred to, e.g., Ehrig et al. [15] for a construction of pushouts and pullbacks in **Sets**, **Graphs**, and **TGraphs**.

Example. To provide some intuition for the universal properties in Definition 6, Fig. 8 depicts a concrete example of a pullback in **Sets**. All functions a, b, c, and d map the same letters to themselves in source and target sets, e.g., $c(p) = p$. Using the same schema and labels for all sets and functions as in Definition 6, $\{D, d, b\}$ is the pullback of $a : B \to A$ and $c : C \to A$. By using the "same" letters in all sets, this example shows why pullbacks in **Sets** can be intuitively understood as generalised intersections, i.e., $\{p\} = \{a, p\} \cap \{p, i\}$.

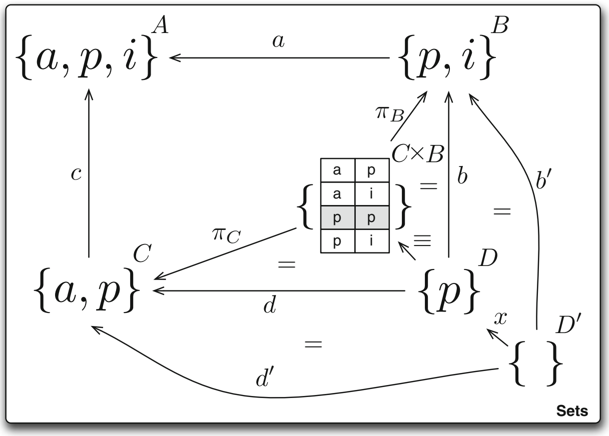


Fig. 8. Pullback construction and intuition for universal property in **Sets**

Pullbacks in **Sets** can be constructed by determining the cartesian product $C \times B$ with projections to each set π_C and π_B, and then taking all entries for which the inner square commutes: $\pi_C; c = \pi_B; a$. This is the case here for only the entry (p, p) (highlighted in grey), which is taken to form the final result D (choosing one of the ps as a representative). The interested reader is referred to Ehrig et al. [15] for a proof that this construction actually does result in a pullback. To demonstrate the universal property of the pullback, the empty set can be taken as D'; The universal properties guarantees the existence of $x : D' \to D$ (with all required conditions). Conditions (1) and (2) for pullbacks in Definition 6 characterise *the* pullback uniquely and can be used in our example to rule out both $C \times B$ and $\{\}$ as pullback candidates: $C \times B$ is not the pullback because it violates Condition (1), i.e., $\pi_C; c \neq \pi_B; a$; $\{\}$ is not the pullback because it violates Condition (2) as it is, e.g., impossible to construct a total function from D to D' (swapping their roles in Condition (2), the universal property for pullbacks).

We are now ready to define composition of deltas for spans of typed graphs. As depicted in Fig. 9, the pullback (think intersection) of the retained typed graphs G_2, G_4 is constructed as G_6 with respect to the overlapping in G_3. The composed delta can now be formed by composed arrows $\delta_5 \,;\, \delta_1$ and $\delta_6 \,;\, \delta_4$.

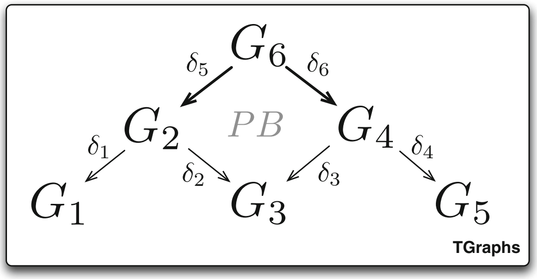

$$\begin{tikzcd}
 & & G_6 \arrow[dl, "\delta_5"'] \arrow[dr, "\delta_6"] & & \\
 & G_2 \arrow[dl, "\delta_1"'] \arrow[dr, "\delta_2"'] & PB & G_4 \arrow[dl, "\delta_3"] \arrow[dr, "\delta_4"] & \\
G_1 & & G_3 & & G_5
\end{tikzcd}$$

Fig. 9. Composition of spans of typed graphs

The following fact applies this idea for composing spans to finally establish typed graphs and spans of typed graphs as a model space according to Definition 3.

Fact 4 (Model Space of Typed Graphs and Spans). *A set M of typed graphs over the same type graph TG, together with a set Δ of spans of typed graphs from M, form a model space $(M, \Delta, \,;\, , id, inv)$ with:*

- $\forall\, G \in M, id_G := G \xleftarrow{id} G \xrightarrow{id} G \in \Delta_{(G,G)}$, *with id taken from* ***TGraphs****,*
- *delta composition defined as follows:*
  $\forall \delta = G_1 \xleftarrow{\delta_1} G_2 \xrightarrow{\delta_2} G_3, \delta' = G_3 \xleftarrow{\delta_3} G_4 \xrightarrow{\delta_4} G_5 \in \Delta.$
  $\delta \,;\, \delta' := G_1 \xleftarrow{\delta_5 \,;\, \delta_1} G_6 \xrightarrow{\delta_6 \,;\, \delta_4} G_5 \in \Delta$, *where* $(\delta_5, \delta_6, G_6)$ *is the pullback of* δ_2, δ_3*,*
- $\forall\, \delta = G \xleftarrow{\delta^-} \overline{G} \xrightarrow{\delta^+} G' \in \Delta_{(G,G')}, inv(\delta) = G' \xleftarrow{\delta^+} \overline{G} \xrightarrow{\delta^-} G \in \Delta_{(G',G)}.$

Proof (Sketch). Associativity of **;** can be shown using pullback composition and decomposition [15], identity follows from identity in **TGraphs**, and the conditions for *inv* follow directly from the definition of *inv* and *id* for spans.

Example. To provide a concrete example for the rather abstract Definition 6, Fig. 10 depicts an example of two spans of typed graphs together with their composition according to Fact 4. To avoid diagram clutter, all internal mappings of typed graph arrows are indicated by using the same node label, i.e., the source node with label a is mapped to the target node with label a. Note, however, that labels have no further semantics and are *not* part of our typed graphs according to Definition 4. To ease comparison with the previous example in **Sets** depicted in Fig. 8, the same node labels are reused. Intuitively, pullback construction in **Graphs** is basically just component-wise pullback construction in **Sets** for edges and nodes. To the left of Fig. 10, the first delta removes a prescription of a (Aspirin) to the patient p and adds a new pharmaceutical i (Ibuprofen) to the model. The second delta to the right of Fig. 10 then removes a from the model and prescribes the previously created i to the patient p. The composed delta, formed by constructing a pullback square as indicated in Fig. 10, combines both deltas by deleting a together with the `prescribed` edge directly in one deletion step, and similarly, creating i and the new `prescribed` edge in a creation step.

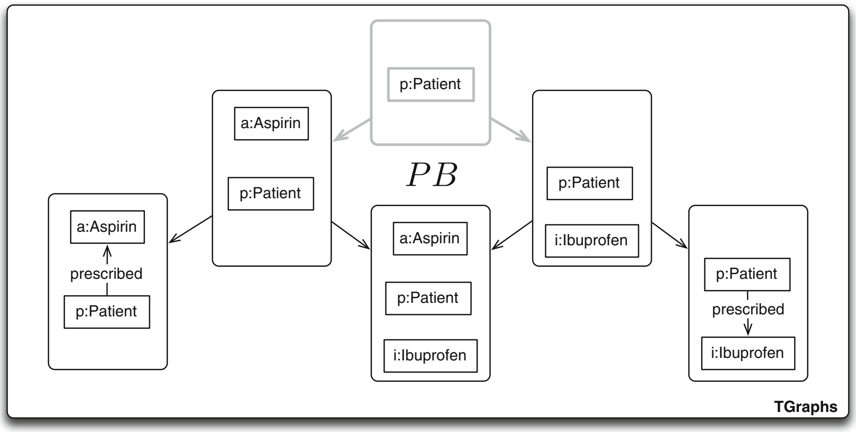


Fig. 10. Composition of deltas from our running example

After defining model spaces of typed graphs and spans of typed graphs, we are now ready to handle the next abstract concept from Diskin et al. [13], that of a *triple space*. A triple space essentially defines triples over two model spaces, let us call these the "source" and "target" model spaces, by connecting source and target models with correspondence links. Recall that our goal is to develop a synchronisation framework, and such triples capture the state of a pair of related source and target models. This is formalised by the following definition.

Definition 7 (Triple Space). *A* triple space $\mathcal{M} \xleftrightarrow{\mathcal{R}} \mathcal{N}$ *is a graph* $(\Delta_{MN}, M \cup N, src, trg)$ *with a set* $M \cup N$ *of nodes (*M *and* N *are the sets of models from the model spaces* $\mathcal{M}$ *and* $\mathcal{N}$*, respectively), set* Δ_{MN} *of edges (referred to as correspondence relations or just "corrs"), and total functions* $src : \Delta_{MN} \rightarrow M$ *and* $trg : \Delta_{MN} \rightarrow N$*.*
We write $m \leftarrow \delta_{MN} \rightarrow n$ *if* $m \in M, n \in N, \delta_{MN} \in \Delta_{MN}$ *and* $src(\delta_{MN}) = m, trg(\delta_{MN}) = n$*.*

Analogously to model spaces of typed graphs, we shall now refine triple spaces for typed graphs. Note that triple spaces according to Diskin et al. [13] are not completely uniform, with corrs – in contrast to source and target models – not necessarily being themselves models taken from a "correspondence" model space.

The standard way of defining triple spaces for TGGs, however, is typically to take a uniform approach, introducing a category of typed *triple graphs* and *triple arrows*. This leads to a richer structure than is required by Diskin et al. [13], but still yields a valid implementation. For the sake of a direct comparison with Diskin et al. [13], we shall ignore this additional structure (deltas on corrs) in this chapter. The following definition formalises typed triple graphs.

Definition 8 (Typed Triple Graphs). *A* triple graph *$G = G_S \overset{\sigma}{\leftarrow} G_C \overset{\tau}{\rightarrow} G_T$ consists of graphs G_S, G_C, G_T and graph arrows $\sigma : G_C \rightarrow G_S$ and $\tau : G_C \rightarrow G_T$.*
A triple arrow *$f = (f_S, f_C, f_T) : G \rightarrow G'$ (depicted in Fig. 11), where $G = G_S \overset{\sigma}{\leftarrow} G_C \overset{\tau}{\rightarrow} G_T$ and $G' = G'_S \overset{\sigma'}{\leftarrow} G'_C \overset{\tau'}{\rightarrow} G'_T$ are triple graphs, is a triple of graph arrows $f_S : G_S \rightarrow G'_S$, $f_C : G_C \rightarrow G'_C$, $f_T : G_T \rightarrow G'_T$, such that $\sigma ; f_S = f_C ; \sigma'$ and $\tau ; f_T = f_C ; \tau'$.*
A type triple graph *is a distinguished triple graph TG.*
A typed triple graph *$\hat{G} = (G, type)$ consists of a triple graph G, and a triple graph arrow $type : G \rightarrow TG$.*
A typed triple graph arrow *$f : \hat{G} \rightarrow \hat{G}'$ is a triple graph arrow $f : G \rightarrow G'$ with $type = f ; type'$, where $\hat{G} = (G, type), \hat{G}' = (G', type')$ are typed triple graphs.*

$$\begin{tikzcd}
G_S \arrow[d, "f_S"'] \arrow[dr, phantom, "="] & G_C \arrow[l, "\sigma"'] \arrow[r, "\tau"] \arrow[d, "f_C"'] \arrow[dr, phantom, "="] & G_T \arrow[d, "f_T"'] \\
G'_S & G'_C \arrow[l, "\sigma'"'] \arrow[r, "\tau'"] & G'_T
\end{tikzcd}$$

Graphs

Fig. 11. A triple arrow is a structure preserving triple of graph arrows.

The following fact ensures that typed triple graphs as objects and typed triple graph arrows as arrows do indeed form a category with pushouts and pullbacks:

Fact 5 (Triples is a Category with pushouts and pullbacks). ***Triples*** *$= (Ob, Arr, \,;\,, id)$ consisting of:*

- *typed triple graphs Ob, typed triple graph arrows Arr,*
- *for $G, G', G'' \in Ob$, $f = G \rightarrow G'$, $g = G' \rightarrow G'' \in Arr$, $;(f, g)$ is defined as $f ; g := (f_S ; g_S, f_C ; g_C, f_T ; g_T)$,*
- *for $G = G_S \leftarrow G_C \rightarrow G_T \in Ob$, $id_G : G \rightarrow G$ is defined as $id_G := (id_S, id_C, id_T)$.*

is a category with pushouts and pullbacks.

Proof (Sketch). Analogous arguments as in Fact 2, i.e., **Triples** can be constructed from **TGraphs** in a standard manner (as a functor category) that preserves numerous properties including the basic categorical structure, pushouts, and pullbacks.

The following fact states that a triple space consisting of two model spaces of typed triple graphs and a set of triples of typed graphs forms a triple space according to Definition 7:

Fact 6 (Triple Space of Typed Triple Graphs). *Let $\mathcal{M}$ and $\mathcal{N}$ be model spaces of typed graphs and spans (according to Fact 4) typed over a* source *type graph* TG_S*, and a* target *type graph* TG_T*, respectively. The set* Δ_{MN} *of all typed triple graphs typed over* $TG = TG_S \overset{\sigma_{TG}}{\leftarrow} TG_C \overset{\tau_{TG}}{\rightarrow} TG_T$ *forms a triple space* $(\Delta_{MN}, M \cup N, src, trg)$ *according to Definition 7, with:*
$\forall\, G = G_S \overset{\sigma}{\leftarrow} G_C \overset{\tau}{\rightarrow} G_T \in \Delta_{MN} : src(G) = G_S, trg(G) = G_T.$

Proof (Sketch). Follows directly from Fact 4 and Definition 7.

Example. As an example of a typed triple graph $(G, type)$, Fig. 12 depicts a diagram in **Graphs** showing the underlying triple structure. The internal mappings of $type_S, type_C, type_T$ are implicitly given by indicating node types in the typed graphs, e.g., `h:Hospital` of type `Hospital`. Note that the graph arrows $\sigma_{TG}, \tau_{TG}, \sigma, \tau$ are in **Graphs** and not necessarily in **TGraphs**, i.e., are not type preserving; their internal mappings are indicated with thin dotted arrows. The diagram introduces the type graph TG_T for the target domain, which we shall

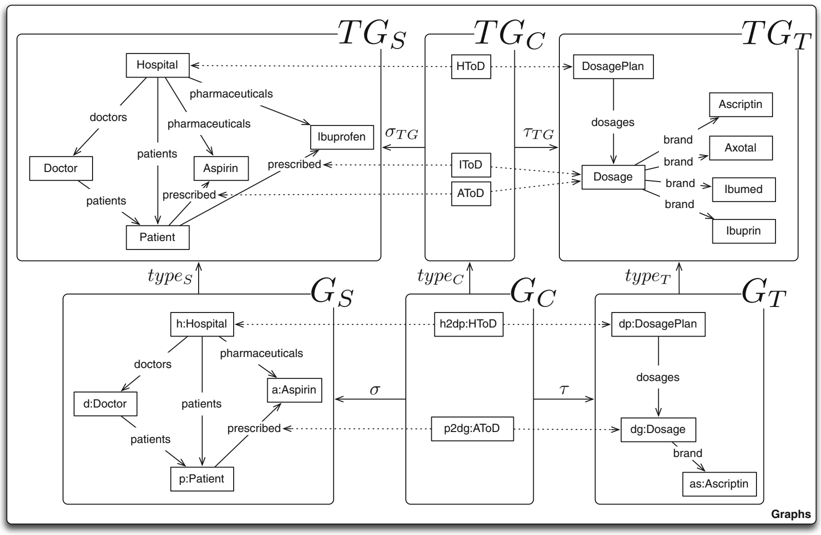


Fig. 12. A typed triple graph from our example

refer to in the rest of the chapter as *MediSupply*. It consists of a dosage plan with dosages, each of which can be marked with a certain brand, e.g., Ascriptin, or Axotal. The correspondence type graph TG_C states which source and target types can be connected in triples: Hospitals correspond to dosage plans, while prescribed edges, representing prescriptions via generic names, correspond to dosages, fixing the actual brand for the prescription. The typed triple graph $G_S \leftarrow G_C \rightarrow G_T$ represents a concrete and well-typed example triple.

3 Specifying Symmetric Delta Lenses with TGGs

In this section, we shall first cover the basics of the *symmetric delta lens* (sd-lens) framework of Diskin et al. [13], and then discuss in a step-wise fashion how sd-lenses can be specified with a TGG. Although there are numerous *operational* scenarios that are required for consistency management in general, the sd-lens framework focuses on *forward* and *backward* change propagation. This is best explained and motivated with a concrete example.

Example. A forward propagation, denoted as `fpg`, takes a corr and a source delta as input, and outputs a target delta and a new corr. This is depicted schematically to the left of Fig. 13. In contrast to input artefacts (c and δ_M in the example), output artefacts are denoted with dashed lines (δ_N and c').

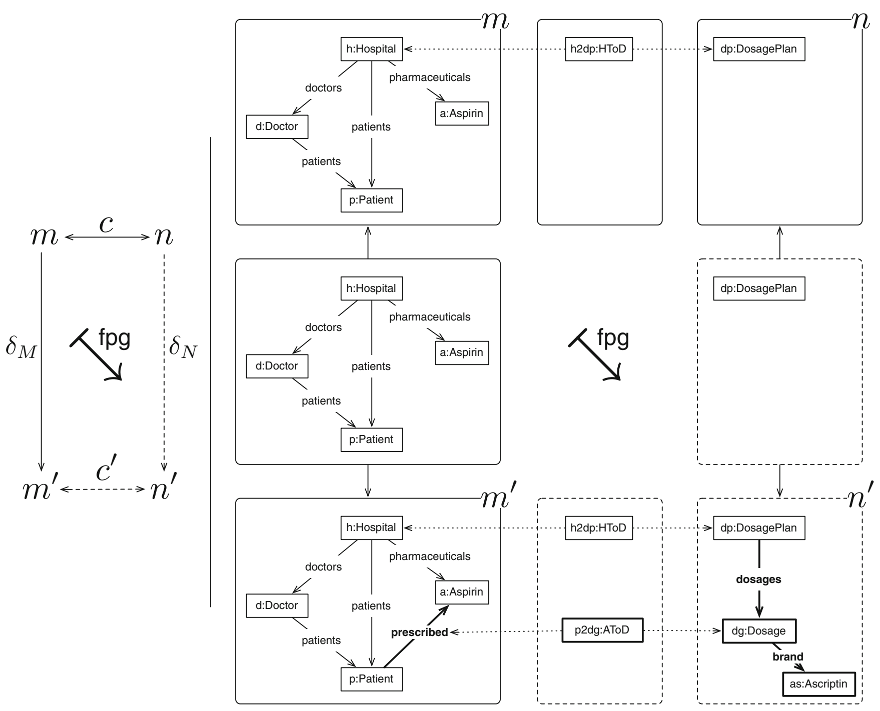


Fig. 13. A forward propagation square

A concrete example based on typed triple graphs and spans of typed graphs is depicted to the right of Fig. 13. Following the same schema as the abstract square to the left of Fig. 13, the example shows how a source delta, representing creating a prescription of Aspirin, is forward propagated to the target domain, where the corresponding target delta represents creating a dosage with Ascriptin as brand. The corr is also updated to reflect the changes.

As indicated by the example, only corrs and deltas that "make sense" are accepted as input and produced as output; intuitively, `fpg` and `bpg` take two adjacent sides of a square and complete them to a full square. Consequently, a corr c that has nothing to do with a source delta δ_M is not valid input, i.e., the source m of the corr must be the same as the source of δ_M. These synchronisation domains (and co-domains) are formalised in the following definition as suitable pullbacks.

Definition 9 (Synchronisation Domains and Co-Domains). *Given a model space $\mathcal{M} = (M, \Delta_M, \,;\, , id, inv)$, source and target functions $src : \Delta_M \to M$ and $trg : \Delta_M \to M$, can be defined for deltas as follows:*
$\forall\, \delta : m \to m' \in \Delta_M,\ src(\delta) := m, trg(\delta) := m'$.

Given another model space $\mathcal{N} = (N, \Delta_N, \,;\, , id, inv)$, let $\mathcal{M} \xleftrightarrow{\mathcal{R}} \mathcal{N}$ be a triple space over $\mathcal{M}$ and $\mathcal{N}$. The synchronisation domains and co-domains $\ulcorner, \urcorner, \llcorner, \lrcorner$ are then defined as the pullbacks of the squares in Fig. 14 (with dashed arrows belonging to the pullback).

$$\begin{tikzcd}
\ulcorner \arrow[r, dashed] \arrow[d, dashed] \arrow[dr, phantom, "PB"] & \Delta_{MN} \arrow[d, "src"] & \Delta_{MN} \arrow[d, "trg"'] \arrow[dr, phantom, "PB"] & \urcorner \arrow[l, dashed] \arrow[d, dashed] \\
\Delta_M \arrow[r, "src"] & M & N & \Delta_N \arrow[l, "src"'] \\
\Delta_{MN} \arrow[r, "src"] \arrow[dr, phantom, "PB"] & M & N \arrow[dr, phantom, "PB"] & \Delta_{MN} \arrow[l, "trg"'] \\
\llcorner \arrow[u, dashed] \arrow[r, dashed] & \Delta_M \arrow[u, "trg"'] & \Delta_N \arrow[u, "trg"] & \lrcorner \arrow[u, dashed] \arrow[l, dashed]
\end{tikzcd}$$

Fig. 14. Synchronisation domains and co-domains

We are now ready to define forward and backward propagation as functions with these reasonable synchronisation domains and co-domains. As formalised by the following definition, a symmetric delta lens is just a pair (`fpg`, `bpg`) of such forward and backward propagation functions.

Definition 10 (Symmetric Delta Lens). *A* symmetric delta lens $\lambda : \mathcal{M} \overset{\mathcal{R}}{\Longleftrightarrow} \mathcal{N}$ *over a triple space* $\mathcal{M} \overset{\mathcal{R}}{\longleftrightarrow} \mathcal{N}$ *is a pair* (fpg, bpg) *of forward and backward propagation functions:*
$fpg : \ulcorner\!\!\longrightarrow \lrcorner, (c, \delta_M) \mapsto (c', \delta_N)$ *such that* $src(\delta_N) = trg(c) \wedge trg(\delta_M) = src(c')$.
$bpg : \llcorner \longleftarrow\!\! \urcorner, (c, \delta_N) \mapsto (c', \delta_M)$ *such that* $src(\delta_M) = src(c) \wedge trg(\delta_N) = trg(c')$.

A primary motivation for developing a synchronisation framework such as the sd-lens framework is to be able to identify and precisely characterise *desirable behaviour* for synchronisers (in this case forward and backward propagation functions). We shall cover some formal properties in Sect. 4, and discuss how these properties can be guaranteed for TGG-based synchronisers.

In the rest of this section, we shall focus for the moment on the basic idea of the TGG approach. Our goal is to develop an intuition for how a TGG represents a rather direct specification of an sd-lens. Without any additional restrictions, the specified sd-lens might not be "well-behaved" in any sense but this is not necessarily a disadvantage. It just means that the TGG framework provides a flexible playground for studying trade-offs between assumptions, corresponding imposed limitations, and guaranteed desirable formal properties.

3.1 Enumerate All Squares

The first step on the way to a TGG-based specification of an sd-lens is to take an *example-based* approach. Imagine how a binary function $sum : int \times int \to int$ could theoretically be specified with a list of "examples" $\{(1,1) \mapsto 2, (1,2) \mapsto 3, \ldots\}$ describing the expected output value for given input pairs. Although this can only be achieved for a finite (and probably rather small) domain, there are some approaches [23] dealing with how to "learn" a program or transformation from a small set of examples. Applying this idea to sd-lenses, examples in this sense would correspond to a set of `fpg` and `bpg` squares, describing how to complete concrete squares given a specific corr and vertical delta. An example for an `fpg` and a `bpg` square is depicted in Figs. 15 and 16, respectively, following the same schema as Fig. 13.

In the former, a hospital `h` with one doctor `d`, one patient `p`, and one pharmaceutical `a` (Aspirin) prescribed for the patient `p`, corresponds to a dosage plan `dp` for the hospital, with a dosage `dg` and its assigned brand `as` (Ascriptin).

Given such a triple, a vertical source delta deleting the prescription and pharmaceutical, and creating a new prescription and pharmaceutical (this time Ibuprofen), is propagated to a corresponding vertical target delta.

As should be expected, the dosage corresponding to the deleted prescription is also deleted, and a new dosage `dg'` is created for the new prescription. A bit surprising is perhaps, that the brand `as` (Ascriptin) is *not* deleted and instead retained in the MediSupply model. The rationale here is that such a "clean up" operation should only be performed explicitly by whoever is in charge of the MediSupply model. Even though there is no dosage currently referencing `as` in the final MediSupply model, this might change in the near future.

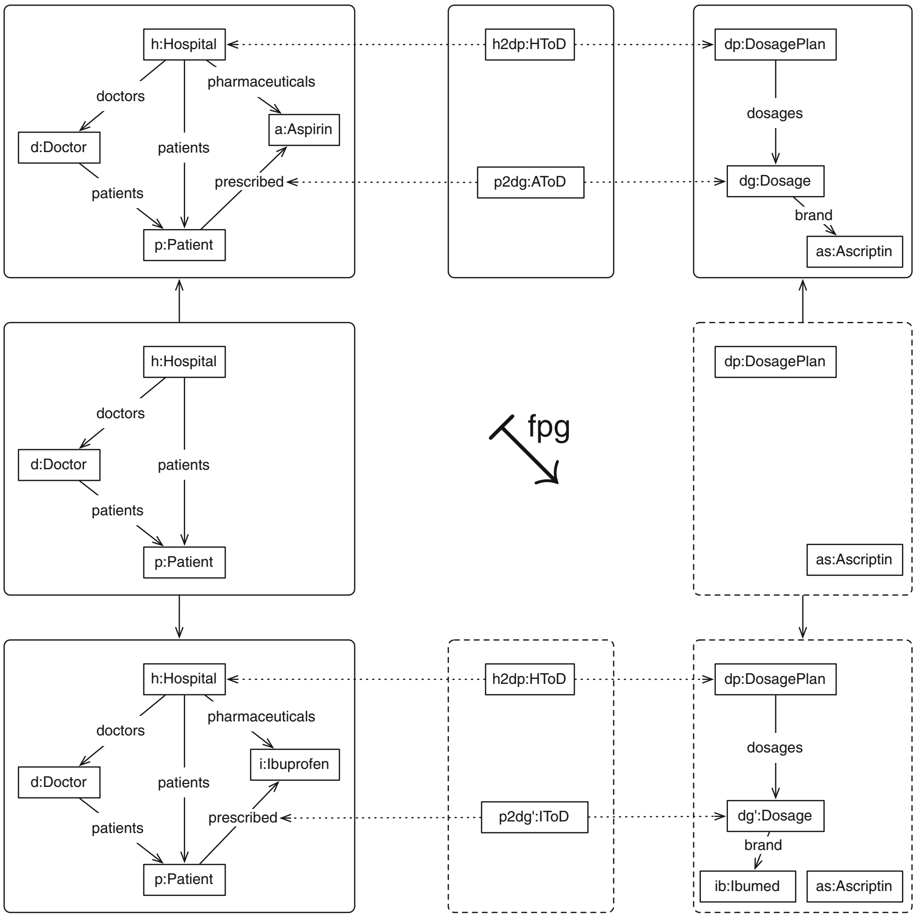


Fig. 15. An `fpg` square

In this concrete example square, however, a new brand `ib` (Ibumed) must of course be created for the new prescription in the MediSoft model.

The situation is analogous but reversed for the `bpg` square in Fig. 16: here a dosage `dg` is deleted together with the referenced brand `as` (Ascriptin), and a new dosage `dg'` is created with a new brand `ib` (Ibumed). In the MediSoft model, however, no corresponding "clean up" is performed, i.e., the pharmaceutical `a` (Aspirin) is *not* deleted, for the same arguments as for the `fpg` square.

The point here is to understand that one could, theoretically, specify the pair (`fpg`, `bpg`) by drawing a lot of (to be precise almost always infinitely many) concrete squares. As this is not very practical, the following ideas are used to progressively reduce the number of "squares" required to specify an sd-lens, culminating in a TGG, a *finite* set of "rules" able to generate an infinite number of concrete `fpg` and `bpg` squares, and thus an sd-lens.

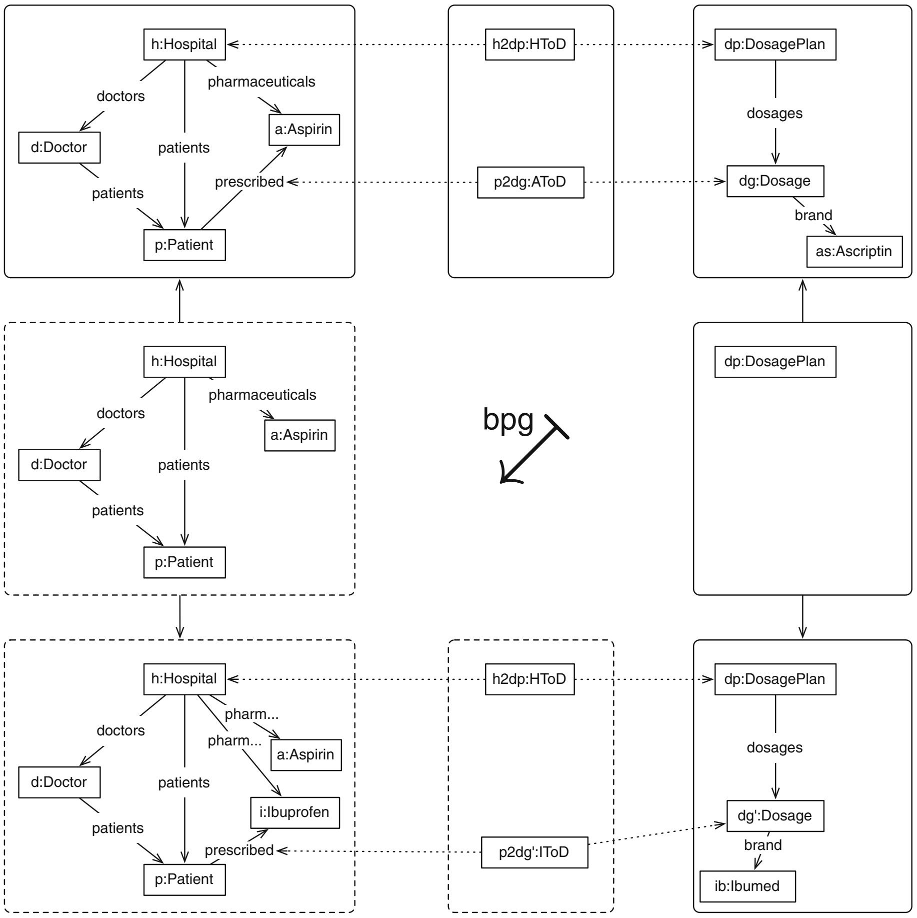


Fig. 16. A `bpg` square

3.2 Enumerate Simultaneous fpg/bpg Squares

The first idea towards reducing the number of required squares is to "fuse" `fpg` and `bpg` squares together into a single *simultaneous* square. Such a square is depicted in Fig. 17, also introducing a series of new notational elements to enable compact diagrams.

Firstly, vertical deltas are depicted in a single graph with colours instead of as an explicit span of graphs: black elements (edges and nodes) are retained, green elements are created, and red elements are deleted by the delta.

Secondly, triples are flattened into a single diagram without explicitly demarcating the individual graphs in the triple. This is indicated by aligning all elements of the source graph to the left, all elements of the target graph to the right, and by using hexagons to clearly distinguish elements in the correspondence graph from all other source and target nodes. To make clear that the links between correspondence elements and source/target elements are *not* edges but rather mappings of the source/target graph arrows between the correspondence graph and source/target graphs, these links are dotted instead of solid.

Finally, and most importantly, the "square" (think of blowing up the diagram into a square without colours) is now to be interpreted as being simultaneous, i.e., as describing at the same time both an `fpg` and a `bpg` square as in the previous subsection. In this concrete square, deleting a prescription and creating a new prescription together with a new pharmaceutical (Ibuprofen to be exact) corresponds to deleting the corresponding dosage and creating a new one together with the new brand (Ibumed to be exact). Recall that this is not a "pattern" of some kind – it is a concrete example and only describes the forward/backward propagation of exactly this triple and source/target vertical deltas.

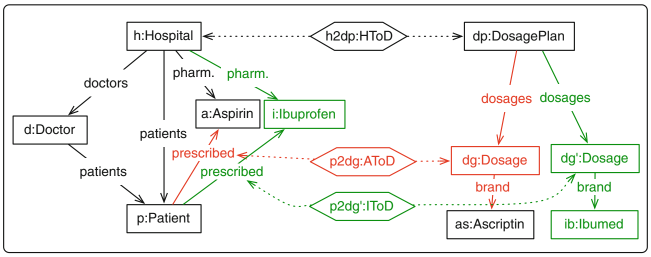


Fig. 17. A simultaneous `fpg`/`bpg` square

The attentive reader should have noticed that we have taken a shortcut. Figure 17 does *not* exactly decompose into the two squares in Figs. 15 and 16. The cleaning up operation in the input domain is not enforced, i.e., after deleting a prescription/dosage, the leftover pharmaceutical/brand is not deleted but left in the source/target model. The rationale here is that these parts of the squares do not match up nicely, cannot be combined, and have to be specified in extra, simpler, but non-simultaneous squares.

For this concrete example, the required squares are depicted in Fig. 18. These are *not* simultaneous squares, i.e., the square to the left ❶ describes an `fpg` square that propagates deleting `a` (Aspirin) by simply doing nothing in the target domain. It does *not* describe a `bpg` square that propagates doing nothing in the target domain to inexplicably deleting `a`. This would indeed be rather surprising. Together with the square to the right ❷, both squares describe the separate "clean up" action in each domain, which can be executed independently by the respective MediSoft and MediSupply clients, and which have no effect on the other domain. Let us refer to such squares in the following as *ignore squares*, as one vertical delta is always the *id* delta.

Although we have almost halved[5] the squares that have to be enumerated to specify an sd-lens, there are still too many for any realistic example. Before

[5] Remember that not all squares can be combined.

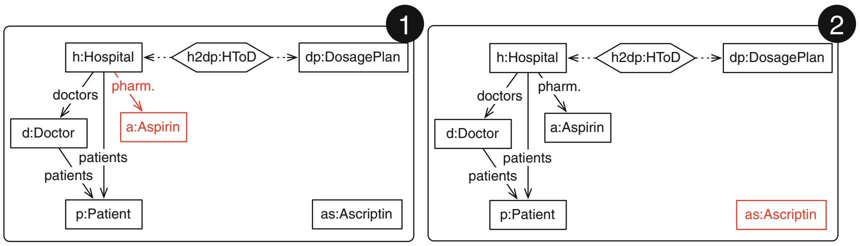


Fig. 18. Not all squares have to be simultaneous

we finally resort to specifying a square "generator" instead, the next simple but important idea, or better, convention, is to describe only monotonic (purely creating or deleting but not both) squares.

3.3 Enumerate Monotonic, Simultaneous fpg/bpg Squares

Let us say we decide to describe only creating, simultaneous (and ignore) squares. By convention, we now agree that all "dual" deleting squares are also derived by inverting the vertical deltas. For our example, this means that we only think of and pair monotonic source and target deltas resulting in the two simultaneous squares depicted in Fig. 19. Comparing this carefully with Fig. 17 reveals that we have again taken another shortcut. What we can now describe is not precisely what Fig. 17 specifies. Using the squares in Fig. 19 we can indeed derive the square in Fig. 17 by composition of squares, but we can also do other things not allowed by Fig. 17. As we have separated creation and deletion, there is nothing forcing us to replace a deleted prescription with a new one. On the positive side, we have described quite a few additional squares, including, e.g., deleting the prescription of `i` (Ibuprofen) and replacing it with a prescription of `a` (Aspirin).

Our ignore squares also have to be rephrased in terms of creation as depicted in Fig. 20. The corresponding squares in Fig. 18 are also indirectly specified as the duals of their creating counterparts.

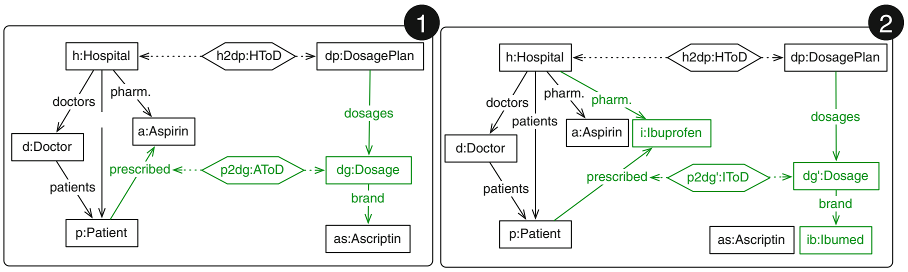


Fig. 19. Monotonic, simultaneous squares

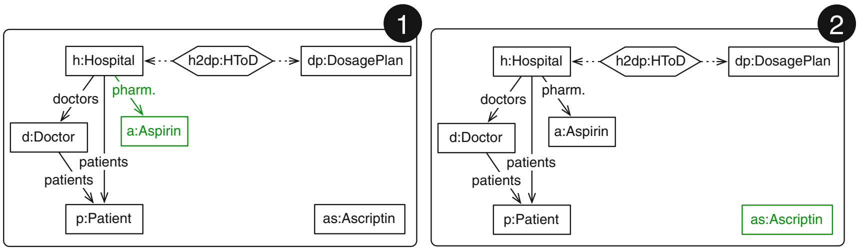


Fig. 20. Monotonic, but non-simultaneous squares

Restricting enumerated squares to purely creating deltas is without doubt a true restriction, arguably even a rather strong one. In practice, however, such dual squares appear to be quite reasonable and can be viewed as a welcome limitation, enforcing not only a compact, but also a simple specification of expected synchronisation behaviour. As we shall see in the next subsection, having purely monotonic *rules* also greatly simplifies the required theory for describing rule application. Finally, monotonic rules also have numerous technical advantages. For instance, deciding if a given graph belongs to a graph language or not is much easier if the rules generating the graph language are monotonic.

3.4 Specify Monotonic, Simultaneous Triple Rules

The most important idea is to transition from enumerating (infinitely many) concrete squares to generating them with a finite set of *rules*. This is a well-known technique used for *generative* language definition, for example, via *string grammars* for context-free tree languages, or *graph grammars*, generalising the technique to describing graph languages.

To generate all our (simultaneous) squares, so-called *triple rules* are used to represent the pre- and postcondition for forward/backward propagation as graph patterns. In its simplest form, a triple rule is just an arrow in the category **Triples** of triple graphs and triple graph arrows:

Definition 11 (Triple Rule). *A* triple rule $r : L \rightarrow R$ *is an arrow in the category* ***Triples*** *(cf. Definition 8).*

Example. So what is the difference between the simultaneous, monotonic squares depicted in Fig. 19 and the set of triple rules depicted in Fig. 21? Compare, for example, the square to the left of Fig. 19 and the top-left triple rule in Fig. 21 (both marked as ❶). The important difference is that the triple rule consists of *variables*, and not concrete nodes and edges.

For our example, this means that the triple rule contains *a* hospital and *a* patient, not a particular hospital and patient as in the square. Typically, the context (the black elements) specified in a rule is also minimal in the sense that unnecessary elements are not included. In this case, we do not care about the doctor assigned to the patient and it is thus omitted in the rule. The other triple rules in Fig. 21 handle different situations: The triple rule to the top-right ❷ handles the case where Axotal is chosen as a brand for the dosage of Aspirin. Analogously, the triple rule to the left ❸ handles Ibuprofen/Ibumed while the triple rule to the right ❹ handles Ibuprofen/Ibupren. Finally, the last triple rule at the bottom ❺ is a so called *axiom* as it has no precondition and can always be applied to create a new hospital and dosage plan.

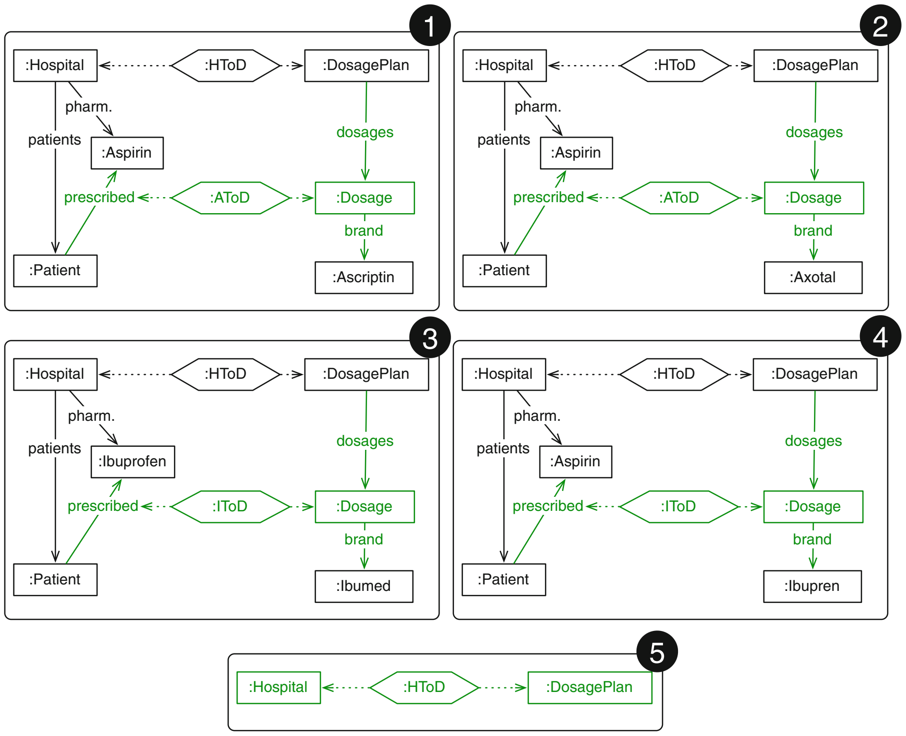


Fig. 21. Finite set of triple rules representing infinitely many squares

Infinitely many concrete squares can be produced from a triple rule by assigning all its variables to actual nodes and edges in a given host graph. In this way, the graph "pattern" representing the precondition (all black elements) of the rule can be "matched" in a larger context (given by the host graph), where the postcondition (all black and green elements) can be subsequently enforced by applying the rule. In our case, correct pattern matching is precisely defined by the requirements for structure preservation in the category **Triples**, i.e., exactly the conditions that make a triple of graph arrows a *triple arrow*. The process of rule application is formalised using the pushout construction in **Triples**.

Definition 12 (Triple Rule Application). *A triple rule $r : L \to R$ can be applied to a host triple graph G at a match arrow $m : L \to G$ to yield a resulting triple graph G' by constructing a pushout $\{G', m' : R \to G', r' : G \to G'\}$ of $\{m, r\}$. This is referred to as a* direct derivation*, denoted by $G \overset{r@m}{\Longrightarrow} G'$ or just $G \overset{r}{\Rightarrow} G'$. A* derivation *$G \overset{*}{\Rightarrow} G'$ is either a direct derivation $G \overset{r}{\Rightarrow} G'$ or a sequence $G \overset{r_1}{\Longrightarrow} G_1 \overset{r_2}{\Longrightarrow} \dots \overset{r_n}{\Longrightarrow} G'$ with $n > 1$.*

We defined pushouts (think generalised union over a common subobject) in Definition 6, and we know that the category **Triples** has pushouts due to Fact 5. This tells us, however, nothing about how pushouts are actually constructed in **Triples**. The interested reader is referred to, e.g., Ehrig et al. [15] for a detailed construction for pushouts in **Sets**, **Graphs**, and **TGraphs**, and, e.g., Schürr [31] for pushouts in **Triples**. For this chapter, the intuition that pushouts in **Triples** are basically constructed in a component-wise manner in **Graphs** suffices.

Example. To support this intuition, a concrete example is depicted in Fig. 22 showing how triple rule $r : L \to R$ (❶ in Fig. 21) is applied to a host graph G to yield the resulting G'. Not that the nodes in the match of L in G via m are depicted with a grey background. The host graph G contains a concrete hospital `h`, with a doctor `d`, and another patient `q`. Rule application is accomplished by constructing the pushout $\{G', r', m'\}$ of $\{r, m\}$ in **Triples**, i.e., in some sense

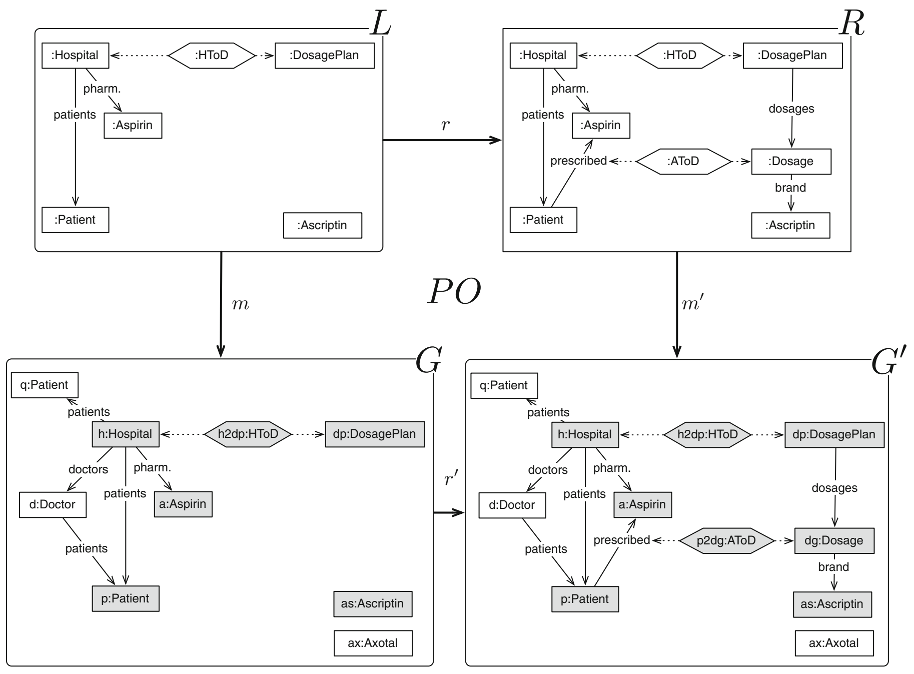


Fig. 22. Monotonic triple rule application via pushouts in **Triples**

merging R and G together by gluing them along all elements that originated from L. Note that the final, concrete square derived by this rule application can be reconstructed from $r' : G \rightarrow G'$ by placing $G = G_S \leftarrow G_C \rightarrow G_T$ over $G' = G'_S \leftarrow G'_C \rightarrow G'_T$ and extending r' to vertical deltas of the form $G_S \leftarrow G_S \rightarrow G'_S$ and $G_T \leftarrow G_T \rightarrow G'_T$. In essence, the precondition pattern L was matched in the concrete context provided by the host graph G, and then extended according to the postcondition pattern R.

After formalising rule application, we now introduce the concept of a *triple graph grammar*, consisting of a type triple graph and a finite set of triple rules:

Definition 13 (Triple Graph Grammar). *A* triple graph grammar $TGG = (TG, \mathcal{R})$ *consists of a type triple graph* $TG = TG_S \leftarrow TG_C \rightarrow TG_T$*, and a finite set* $\mathcal{R}$ *of triple rules.*
Let $\mathcal{L}(TG)$ *denote the set of all triple graphs of type* TG*.*
The triple graph language generated by the triple graph grammar TGG *is given by* $\mathcal{L}(TGG) := \{G \in \mathcal{L}(TG) \mid \exists\ \emptyset \overset{*}{\Longrightarrow} G\}$*, where* $\emptyset$ *is the empty triple graph (the initial object in* ***Triples****), and the derivation* $\emptyset \overset{*}{\Longrightarrow} G$ *consists only of direct derivations with triple rules from the triple graph grammar* TGG*.*
The projection to the source domain is given by:
$Proj_S(\mathcal{L}(TGG)) := \{G_S \in \mathcal{L}(TG_S) \mid \exists G_S \leftarrow G_C \rightarrow G_T \in \mathcal{L}(TGG)\}$.
Projections to the correspondence and target domains are defined analogously.

3.5 Automatically Derive “Boring” Ignore Rules

A final pragmatic, and useful convention, is to automatically and systematically derive a set of “boring” ignore rules that in a sense complete a specified TGG.[6] As a final definition in this section, let us formally distinguish triple rules used to generate squares that are only meant to be interpreted as either `fpg` or `bpg` squares but not as both, i.e., non-simultaneous, monotonic squares such as those depicted in Fig. 20:

Definition 14 (Ignore Rule). *A* source (target) ignore rule *is a triple rule* $r : L \rightarrow R$*,* $r = (r_S, r_C, r_T)$*, with* $r_T = id_{L_T}$ $(r_S = id_{L_S})$*. Source and target ignore rules are collectively referred to as* ignore rules*.*

Example. This idea is best explained using our concrete example. Consider the TGG consisting of 5 triple rules depicted in Fig. 21, and the underlying triple space formed by taking the type triple graph depicted in Fig. 12, i.e., taking all typed source and target graphs for the source and target model spaces, respectively, together with all possible deltas consisting of the atomic operations: (i) creating a node or an edge, and (ii) deleting a node or an edge.

Although our TGG already describes quite a number of `fpg` and `bpg` squares with these 5 rules, there are still many possible inputs to `fpg`/`bpg` for which it says nothing. What should happen if a new doctor is added to a hospital? Or

[6] This is analogous to notions of a “conservative copy” in the context of model migration, e.g., as exemplified in the Epsilon Flock tool [30].

a new patient? If TGG-based tools would always complain about such "undefined" adjacent sides of squares, specifying synchronisers with TGGs would be a tedious, big-bang effort as it is often not feasible to restrict changes to only what is currently supported. In MDE application scenarios, it turns out to be better to simply ignore as many undefined changes as possible. This allows users to deploy and iteratively test TGG-based synchronisers in realistic application scenarios even after just specifying a handful of triple rules. To do this formally, the following convention has proven to be a suitable heuristic for deriving "missing" rules automatically:

> *For every type* **t** *that is not created in any rule, an ignore rule* $\mathbf{r_t}$ *is derived that creates* **t** *with minimal context, i.e., with no context for nodes, and source and target nodes as context for edges.*

Applying this to our example results in the additional ignore rules depicted in Figs. 23 and 24, basically formalising in each domain that these "private" changes do not have an effect on the other domain.

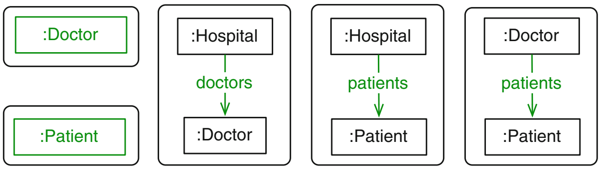


Fig. 23. Automatically derived source ignore rules

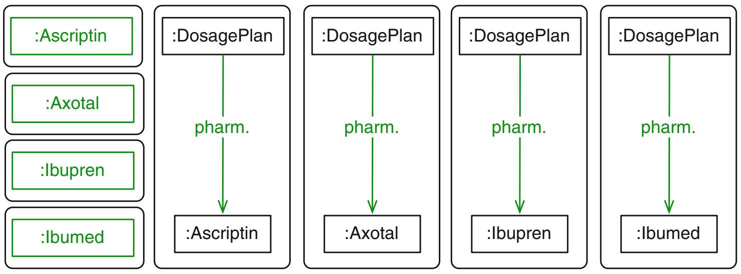


Fig. 24. Automatically derived target ignore rules

Do not be irritated by the compact concrete syntax used in the diagrams: in sum, 13 complete ignore rules are depicted. The top-left triple rule in Fig. 23, for instance, is a source ignore rule: $(r_S, r_C, r_T) : \emptyset \leftarrow \emptyset \rightarrow \emptyset \quad \longrightarrow \quad R_S \leftarrow \emptyset \rightarrow \emptyset$, where $\emptyset$ is the empty graph and R_S is a typed source graph consisting of only one node (`:Doctor`).

3.6 Advanced TGG Language Features

To be able to work productively with TGGs, a series of further language features are necessary in practice. As discussing all these features in detail would go far beyond the scope of this chapter, the following gives only a brief overview of the most important advanced features with pointers for further reading.

In an MDE context, the source and target model spaces are typically specified declaratively by providing a "metamodel", which is basically a type graph according to Definition 4 together with a set of domain constraints used to further restrict the set of valid models in the domain. In our example, such constraints could ensure, e.g., that there are no two pharmaceuticals in the source domain, and, analogously, no two brands in the target domain with the same name, or that every patient is assigned at least one (or perhaps exactly one) doctor, etc. With details depending on the specific modelling framework, some kind of class diagram dialect is often used with which (simple) constraints can be specified via language features such as multiplicities and composite/aggregation annotations for associations. For more complex domain constraints an additional language is sometimes used, for example, the Object Constraint Language (OCL). Constraints are handled in the graph transformation framework via *graph constraints*, which can be converted to suitable *application conditions* for graph transformation rules. The interested reader is referred to Radke et al. [29] for existing results on converting metamodels (class diagrams and an "essential" subset of OCL) to equivalent type graphs with graph constraints. Handling domain constraints in a TGG-context is discussed, e.g., by Anjorin et al. [6]. Enriching a TGG with application conditions is a useful and powerful means of controlling and shaping the generated language, and does not have to be directly connected to avoiding constraint violation. Most TGG-tools, however, do not yet support arbitrary application conditions [21,26].

Although attribute handling was avoided in this chapter for presentation purposes, models typically have multiple attribute values in practice, which have to be accounted for in deltas, constraints, and rules. For our example, this would include the unique chemical formula of pharmaceuticals, names and address information of patients and doctors, the cost of brands, etc. The interested reader is referred to Ehrig et al. [15] for details on *attributed* typed graphs, and to Anjorin et al. [7] for a discussion of how attribute conditions can be handled in a TGG context.

Consider the five triple rules in Fig. 21. Without any support for modularity and reuse, large parts of these rules would have to be duplicated for every pharmaceutical. While this might not appear problematic for such a small example, in practice TGGs consisting of hundreds of rules would become challenging to maintain and refactor. There exist various approaches to enable compact and modular rule specification: on the level of class diagrams, *inheritance* can be used to introduce common and often abstract super-types. For our example, it would be natural to introduce an abstract type Pharmaceutical and make Aspirin and Ibuprofen "inherit" from this type. In many cases, (parts of) triple rules can be unified by demanding abstract instead of concrete types as context. The interested reader is referred to Ehrig et al. [15] for a formalisation and integration of inheritance into the graph transformation framework. On the level of graph transformation rules, there exist approaches such as that suggested by Anjorin et al. [5] that allow triple rules to be split into and combined flexibly from "rule fragments". For our example, commonalities of the triple rules in Fig. 21 could be extracted into a single "abstract" triple rule, which other rules could "refine" and extend as required. Another approach is to unify rules by adding variability annotations [34]. For our example, the triple rules with labels 1 and 2 in Fig. 21 could be thus unified in a single triple rule by annotating the differences (the node `:Ascriptin` vs. `:Axotal`) with labels indicating the variants in which these elements are to be present.

Although all TGG rules are depicted visually in this chapter, for an improved editing and refactoring experience, crucial when extracting "super-rules" and generally during maintenance, the TGG tool eMoflon[7] now supports a simple *textual concrete syntax* together with a read-only visualisation.

In general, addressing realistic application scenarios with TGGs, such as an application to transforming satellite procedures by Hermann et al. [19], often requires some pre- and post-processing with normal graph transformations, e.g., to establish auxiliary derived elements in models in order to simplify (or enable) the required TGG specification.

4 Formal Properties

In an ongoing attempt to formally capture what "good" behaviour means in the context of model synchronisation, a set of laws has been proposed for the sd-lens framework [13]. These laws can not only be used to check whether a manually programmed pair of `fpg` and `bpg` functions is in this sense "well-behaved", but also to guide and drive the automatic derivation of an sd-lens from some compact specification such as a TGG. Indeed, the interpretation of triple (ignore) rules as (non-) simultaneous "square generators", as suggested in this chapter, can be justified with the basic laws of the sd-lens framework.

The typical lens laws are round trip properties, formulated over `fpg` and `bpg` without any assumptions about their internal structures. This treatment of `fpg` and `bpg` as black-box functions is natural for approaches based on functional programming, from which the original ideas for lenses originate [16]. The well-known lens laws have been transferred to the sd-lens framework by Diskin et al. [13] resulting in (i) *stability*, formalising the expectation that idle deltas be mapped to idle deltas,[8] (ii) *undoability*, demanding well-behavedness of a vertical composition of squares, and (iii) *invertibility*, demanding well-behavedness of a horizontal composition of squares. Based on Definition 6, it is possible to formulate these laws in a TGG setting. Hermann et al. do exactly this [18], providing proofs for stability and undoability by formalising exactly how `fpg` and `bpg` functions are derived from a TGG that must comply with certain additional restrictions.

[7] www.emoflon.org.

[8] Hence the suggestion to view source ignore rules as generating only `fpg` squares.

The goal of this chapter in general, and of this section in particular, is humble but clear: to provide an introduction to and overview of the TGG approach that is particularly accessible to bxers more accustomed to the various lens frameworks and their style of round trip "functional" laws. This means that instead of covering the familiar round trip laws in a TGG context (as done by Hermann et al. [18], to which the interested reader is referred), we shall instead cover basic TGG laws rephrased to fit better in the sd-lens framework.

The TGG laws *correctness* and *completeness*, proposed by Schürr and Klar [32], are in constrast more related to the consistency relation induced by a specified TGG, namely that a triple is consistent if and only if it is a member of the language of triples generated by the TGG. Taken from Diskin et al. [13], a notion of consistency can be defined as follows on the level of the sd-lens framework:

Definition 15 (Consistency in the SD-Lens and TGG Framework). *Given a triple space* $\mathcal{M} \overset{\mathcal{R}}{\longleftrightarrow} \mathcal{N} = (\Delta_{MN}, M \cup N, src, trg)$ *over model spaces* $\mathcal{M}$ *and* $\mathcal{N}$*, let* $\mathcal{C} \subseteq \Delta_{MN}$ *be a chosen set of* consistent *corrs in* Δ_{MN}*.*
A triple $m \leftarrow \delta_{MN} \rightarrow n$ *is* consistent *if* δ_{MN} *is consistent, i.e.,* $\delta_{MN} \in \mathcal{C}$*.*
A pair of source and target models $(m, n) \in M \times N$ *is said to be* consistent *with respect to* $\mathcal{C}$*, denoted by* $m \sim_{\mathcal{C}} n$*, iff* $\exists\, \delta_{MN} \in \mathcal{C}.\ m \leftarrow \delta_{MN} \rightarrow n$*.*
A source model $m \in M$ *is* consistent *with respect to* $\mathcal{C}$ *iff* $\exists\, n \in N.\ m \sim_{\mathcal{C}} n$*.*

In the TGG framework, the set $\mathcal{C}$ *of consistent corrs is given by:*
$\{G_C \mid \exists\, G_S \leftarrow G_C \rightarrow G_T \in \mathcal{L}(TGG)\}$*, i.e., the language generated by a given triple graph grammar* TGG *over a triple space of typed triple graphs.*

As the TGG laws say very little about how exactly `fpg` and `bpg` should be derived from a TGG, the intuition of squares suggested in this chapter is, therefore, only "best practice" and should be regarded as just one of many possible ways of specifying, reading, and working with TGG rules. It is, however, based on and compatible with the work of Diskin et al. [13] and Hermann et al. [17], and is hopefully thus accessible to bxers familiar with lenses and in particular with the sd-lens framework. It also reflects, to the best of our knowledge, the basic strategy employed in current TGG tools [21,26].

A problem with the square generation intuition, however, is that `fpg` and `bpg` do not necessarily end up being functions, let alone total functions, in practice. The TGG framework is naturally non-deterministic, not only during rule application which is typical for the graph transformation approach, but also concerning the choice of applicable rules in each step of a derivation. The standard TGG laws do not help here (or viewed positively, do not impose any restrictions). There are many ways of handling non-determinism (if it is unwanted), including incorporating uncertainty or variability directly in the involved model spaces [12], or using *update policies* at design time (e.g., via preferences) or at runtime (e.g., via user interaction) to choose between multiple matches of multiple applicable rules whenever necessary.

Yet another way of handling non-determinism is via *monadic* lenses, as discussed in the introductory chapter of this book. For the rest of this chapter, we shall assume that an appropriate update policy is used to restrain TGG-based `fpg` and `bpg` operations to be (not necessarily total) functions.

4.1 Problem Setting in an MDE Context

The TGG framework has been influenced and shaped by application scenarios mostly in an MDE context. This concerns how model and triple spaces are specified in practice, expectations concerning deltas that can result at runtime, and the corresponding synchronisation (co-)domains of `fpg` and `bpg`. It is important to understand the "MDE-based" problem setting, as the TGG laws and corresponding current challenges only make sense in this context.

The first point is that TGG-based synchronisers are typically partial and can fail in practice, even on "meaningful" input from the synchronisation domain as defined in Definition 9. One reason is scalability, i.e., certain heuristics or strategies are used instead of a complete search, substantially increasing efficiency but also imposing extra (sometimes very technical) restrictions on the class of TGGs and on the supported synchronisation domain. Another reason is a mismatch between how model spaces are specified in an MDE context, namely declaratively using a metamodel (essentially a type graph for this chapter), and how a TGG is specified over these model spaces, namely in a generative, rule-based manner. Finally, in a typical MDE context there are often no restrictions on how a model can be edited, i.e., most constraints can be violated. The expectation here is that a synchroniser should still behave elegantly for invalid input, e.g., detecting and rejecting such invalid input with a helpful message and a precise characterisation of the exact problem.

This situation[9] is depicted schematically in Fig. 25 (to be explained in more detail in a moment): the source model space $\mathcal{M} = (M, \Delta_M, id_M, inv_M)$ is specified by providing TG_S. The set of all source models is taken as $M = \mathcal{L}(TG_S)$, while Δ_M is formed by generating all possible deltas on all these models from the basic set of four[10] possible updates: (i) adding a single edge, (ii) removing a single edge, (iii) adding a single node, and (iv) removing a single, isolated node.

If `fpg` is partial, i.e., can fail for certain adjacent square sides as input, it is of course useful to precisely characterise the subset of $\ulcorner$ on which `fpg` is total. This is stated by the *completeness* law, which essentially demands that a TGG-based `fpg` be defined for every pair of consistent triple and *consistent* source delta in $\ulcorner$. But what exactly is a consistent source delta? Intuitively, a consistent source delta forms an input pair of adjacent sides of an `fpg` square that might not necessarily be generated by the TGG in question, but can at least be "approximated" with a generated `fpg` square. This is the case for *any* source delta between two consistent source models, as formalised by the following definition.

[9] The diagram is more complex for constraints; the interested reader is referred to Anjorin et al. [4].

[10] Most practical implementations also support attribute manipulation; some even support "moving" a subgraph in a containing graph.

Definition 16 (Consistent Source Delta). *Given a triple space* $\mathcal{M} \overset{\mathcal{R}}{\longleftrightarrow} \mathcal{N} = (\Delta_{MN}, M \cup N, src, trg)$ *over source and target model spaces* $\mathcal{M}$ *and* $\mathcal{N}$*, respectively, and a set of consistent corrs* $\mathcal{C} \subseteq \Delta_{MN}$*, a source delta* $\delta_M : m \rightarrow m'$ *is* consistent *iff both* m *and* m' *are consistent. Consistent target deltas are defined analogously.*

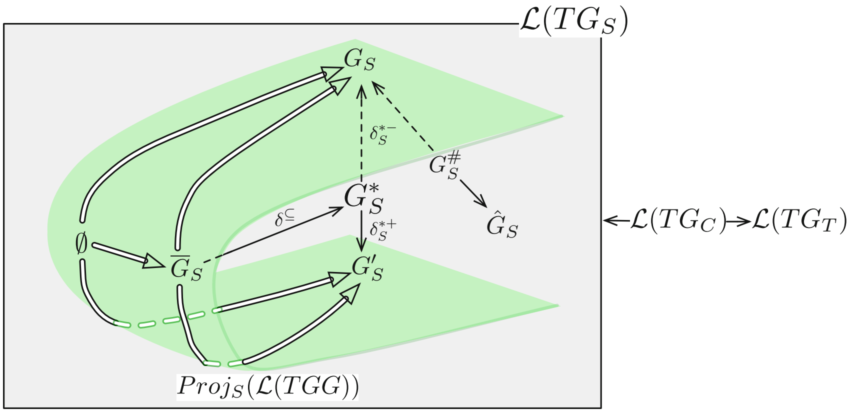


Fig. 25. TGG-based "consistency surface" and (in)consistent input deltas

As can be seen to the left of Fig. 25,[11] the set of consistent source models is given by $Proj_S(\mathcal{L}(TGG))$ and can be depicted as a consistency "surface" (the curved green surface in the diagram). With $\emptyset$ representing the empty graph, consistent source models G_S, G'_S, and $\overline{G}_S$ are all reachable via (projections to the source component of) derivations of rules of the TGG (depicted as double lined arrows $\Longrightarrow$ in the diagram). These derivations imply that deltas $\delta : \emptyset \rightarrow G_S, \delta' : \emptyset \rightarrow G'_S, \overline{\delta} : \emptyset \rightarrow \overline{G_S}$ exist and can be described by a series of `fpg` squares (rule applications). Propagating such source deltas (and their inverses) is more or less straight-forward, as the TGG rules specify directly what is to be done. In an MDE context, however, where source models are typically updated using editors that are completely independent of some underlying TGG-based synchroniser, the expectation is to be able to forward propagate *any* consistent source delta.

In Fig. 25, $\delta^* : G_S \rightarrow G'_S$, depicted as a span $G_S \overset{\delta_S^{*-}}{\leftarrow} G_S^* \overset{\delta_S^{*+}}{\rightarrow} G'_S$ in the diagram, represents a source delta that is consistent, but does not necessarily correspond to any TGG derivation. This is indicated by placing G_S^* in $\mathcal{L}(TG_S)$ but not on the "consistency surface" $Proj_S(\mathcal{L}(TGG))$. It is arguably unreasonable to expect a TGG-based `fpg` to be defined for such consistent deltas – how

[11] This visualisation, as an attempt to impart to a geometric intuition for consistency, was suggested by James McKinna at the 2016 summer school on bidirectional transformations: https://www.cs.ox.ac.uk/projects/tlcbx/ssbx/.

should we know what to do? But such a flexible "completeness" property is unarguably *useful* in (MDE) practice and is formalised by the following definition.

Definition 17 (Transformation Completeness). *A* symmetric delta lens $\lambda : \mathcal{M} \overset{\mathcal{R}}{\Longleftrightarrow} \mathcal{N}$ *over a triple space* $\mathcal{M} \overset{\mathcal{R}}{\longleftrightarrow} \mathcal{N}$ *is* forward transformation complete *with respect to a set of consistent corrs* $\mathcal{C} \subseteq \Delta_{MN}$*, if its forward propagation function* $fpg : \ulcorner \longrightarrow \lrcorner$ *is total on the set:*

$$\{(c, \delta_M) \in \ulcorner \mid c, \delta_M \textit{ are consistent}\}$$

Backward transformation completeness *is defined analogously.*
An sd-lens is transformation complete *if it is both forward and backward transformation complete.*

A source delta that is *not* consistent is represented in Fig. 25 as $\hat{\delta} : G_S \to \hat{G}_S$, depicted as a span $G_S \leftarrow G_S^{\#} \to \hat{G}_S$ in the diagram (note that $\hat{G}_S$ is not on the consistency surface). According to Definition 17, a transformation complete sd-lens does not have to be able to propagate such a delta and can reject it (fail).

The attentive reader might already have realised that achieving transformation completeness is actually quite simple: As the starting G_S and ending model G'_S for every consistent delta δ^* must themselves be consistent, we know that there exist deltas $\delta : \emptyset \to G_S$ and $\delta' : \emptyset \to G'_S$, which can thus both be described directly with derivations $\emptyset \overset{*}{\Rightarrow} G_S$ and $\emptyset \overset{*}{\Rightarrow} G'_S$ of the TGG. This means that we could simply "rollback" δ and propagate δ', i.e., take $inv(\delta) \,;\, \delta'$ as an approximation of δ^*. As this effectively ignores δ^*, i.e., produces the same approximation *independent* of how G_S is exactly changed to G'_S, it should not be surprising that this "brute force" strategy does not produce good results. As depicted in Fig. 25, most TGG tools attempt to determine an intermediate (if possible maximal) consistent source model $\overline{G}_S$ with deltas $\delta_S^- : \overline{G}_S \to G_S, \delta_S^+ : \overline{G}_S \to G'_S$, providing the approximation $inv(\delta_S^-) \,;\, \delta_S^+$ for δ^*. The intuition is that avoiding the unnecessary rollback of $\overline{\delta} : \emptyset \to \overline{G}_S$ yields a"better" approximation of δ^*.

While TGG tools do have substantial freedom concerning how best to approximate deltas such as δ^*, the following correctness law demands basic compatibility with the underlying TGG, i.e., that the resulting triple be consistent:

Definition 18 (Transformation Correctness). *A* symmetric delta lens $\lambda : \mathcal{M} \overset{\mathcal{R}}{\Longleftrightarrow} \mathcal{N}$ *over a triple space* $\mathcal{M} \overset{\mathcal{R}}{\longleftrightarrow} \mathcal{N}$ *is* forward transformation correct *with respect to a set of consistent corrs* $\mathcal{C} \subseteq \Delta_{MN}$*, if the following holds for its forward propagation function* $fpg : \ulcorner \longrightarrow \lrcorner$*:*

$$fpg \textit{ is defined for } (c, \delta_M) \in \ulcorner \Longrightarrow fpg(c, \delta_M) = (c', \delta_N) \in \lrcorner \textit{ and } c' \textit{ is consistent}$$

Backward transformation correctness *is defined analogously.*
An sd-lens is transformation correct *if it is both forward and backward transformation correct.*

Example. A consistent delta $\delta_T^* : G_T \rightarrow G'_T$ and inconsistent delta $\hat{\delta}_T : G_T \rightarrow \hat{G}_T$ are depicted for our running example as spans in Fig. 26, following the same schema as in Fig. 25. Note, however, that Fig. 26 shows *target* deltas as it easier to construct an instructive example in the target model space.

The initial consistent target graph G_T consists of a dosage plan `dp`, with a single dosage `dg` of `as` (Ascriptin). The delta $\hat{\delta}$ removes the Dosage `dg` from the dosage plan `dp` leading to a target model $\hat{G}$ that is inconsistent, i.e., cannot be generated with the rules of our TGG as the target component of a triple graph. The delta δ_T^* changes the assigned brand of the dosage `dg` from Ascriptin to Axotal. Although the resulting target graph G'_T is consistent (consistent graphs are on the consistency surface and additionally have a bold border and a drop shadow), deleting and creating a single edge connecting a dosage to a brand cannot be directly described as a derivation of our TGG. As the rules always create (and thus delete) a dosage together with both edges connecting it to the dosage plan and a current brand, the best approximation in terms of the TGG is via the consistent intermediate target graph $\overline{G}_T$. The delta $\hat{\delta}_T$ is inconsistent as deleting the edge connecting a dosage to its dosage plan (resulting in the final target graph $\hat{G}_T$) cannot be described with a derivation of the TGG.

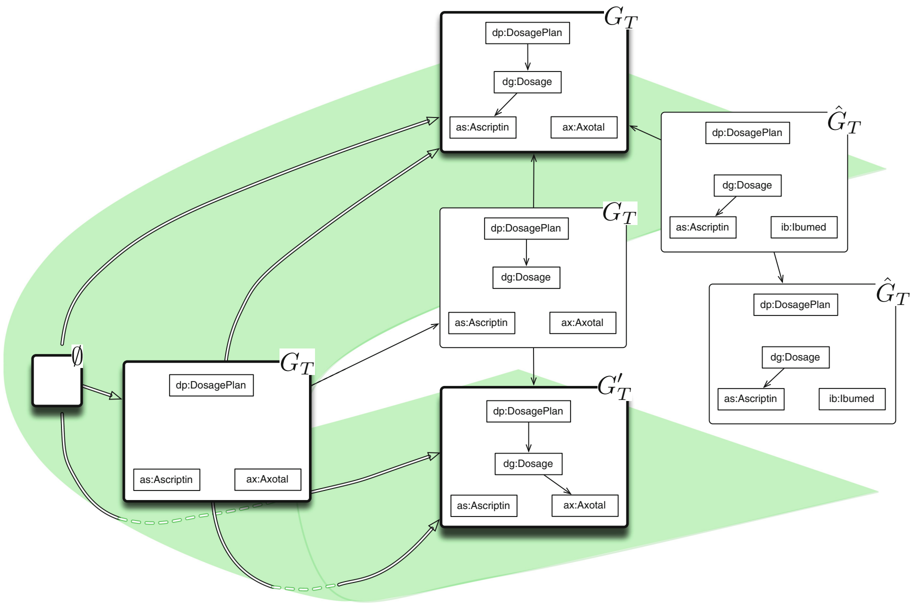


Fig. 26. Consistent and inconsistent target deltas for our running example

4.2 A Correct and Complete TGG Synchronisation Algorithm

This subsection formalises (at least to a certain extent) the suggestion in this chapter of viewing a TGG not only as a generative specification of consistency, but also as a direct specification of `fpg` and `bpg` in terms of generated "squares".

A TGG synchronisation algorithm is a concrete strategy of how to produce an sd-lens, i.e., propagation functions (`fpg`, `bpg`), from a given TGG. The algorithm described in the following, which we shall refer to as a *precedence-driven* TGG synchronisation algorithm, is based on the work of Lauder et al. [24], developed further by Anjorin [1], and is not exactly what is proposed by Hermann et al. [18]. The main difference is that it favours *scalability*, i.e., propagation time depends on the size of the "delta" and not of the models, over a slightly better handling of information loss. Conceptually, however, both algorithms are similar and share similar strengths and weaknesses. The precedence-driven algorithm of Lauder et al. also fits with the "square generation" intuition for TGGs proposed in this chapter, and is implemented as described in the TGG tool eMoflon (with additional details for handling advanced language features).

Let us start by repeating the valid input and expected output for correct and complete TGG-based forward propagation, where consistency is defined with respect to a given triple graph grammar $TGG = (TG, \mathcal{R})$, i.e., $\mathcal{C} = Proj_C(\mathcal{L}(TGG))$:

`Input:` A pair $(G, \delta_S^*) \in \ulcorner$ such that:

1. The triple graph $G = G_S \leftarrow G_C \rightarrow G_T$ is consistent
2. The source delta $\delta_S^* = G_S{\leftarrow}G_S^*{\rightarrow}G'_S$ is consistent

`Output:` A pair $(G', \delta'_T) \in \lrcorner$ computed as $\texttt{fpg}(G, \delta_S^*)$ such that:

1. The target delta is of the form $\delta'_T = G_T \leftarrow \overline{G}_T \rightarrow G'_T$
2. The triple graph $G' = G'_S \leftarrow G'_C \rightarrow G'_T$ is consistent

As depicted schematically in Fig. 27, `fpg` is realised via two auxiliary functions `rollback` and `translate`, i.e., $\texttt{fpg}(G, \delta_S^*) = \texttt{translate}(\texttt{rollback}(G, \delta_S^*))$. Note that correspondence models are depicted as bidirectional arrows, all derived arrows are denoted with a dashed stroke, and that the diagram is compatible with both Figs. 25 and 26. The contract for `rollback` is as follows:

`Input:` A pair $(G, \delta_S^*) \in \ulcorner$ such that:

1. The triple graph $G = G_S \leftarrow G_C \rightarrow G_T$ is consistent
2. The source delta $\delta_S^* = G_S \overset{\delta_S^{*-}}{\leftarrow} G_S^* \overset{\delta_S^{*+}}{\rightarrow} G'_S$ is consistent

`Output:` Triple arrow $\delta^- : \overline{G} \rightarrow G$, and source delta $G_S \overset{\delta_S^-}{\leftarrow} \overline{G}_S \overset{\delta_S^+}{\rightarrow} G'_S$ such that:

1. The triple graph $\overline{G} = \overline{G}_S \leftarrow \overline{G}_C \rightarrow \overline{G}_T$ is consistent
2. $\delta^- : \overline{G} \rightarrow G$ can be derived via a derivation $\overline{G} \overset{*}{\Rightarrow} G$ with rules of the TGG

3. The source delta $G_S \xleftarrow{\delta_S^-} \overline{G}_S \xrightarrow{\delta_S^+} G'_S$ is a reduction of $G_S \xleftarrow{\delta_S^{*-}} G_S^* \xrightarrow{\delta_S^{*+}} G'_S$, i.e., $\exists\ \delta_S^{\subseteq} : \overline{G}_S \to G_S^*$, such that $\delta_S^{\subseteq} \mathbin{;} \delta_S^{*-} = \delta_S^-$, $\delta_S^{\subseteq} \mathbin{;} \delta_S^{*+} = \delta_S^+$ and $\delta_S^{\subseteq}$ is monomorphic.
4. Finally, $\delta_S^+ : \overline{G}_S \to G'_S$ can be extended to a triple graph arrow $\delta^+ : \overline{G} \to G'$ using the rules of the TGG, i.e., $\exists\, \overline{G} \overset{*}{\Rightarrow} G'$

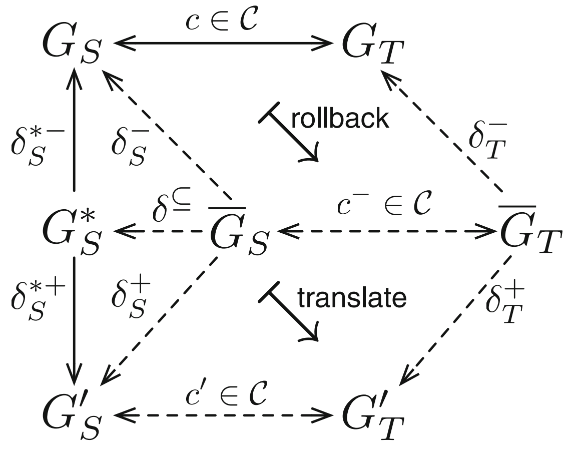
The function `rollback` essentially determines a consistent triple graph $\overline{G}$ by deleting elements in the input triple graph G. In addition, Point (4) of its contract guarantees that $\overline{G}$ is also a suitable starting point for the next and final step `translate` with the following contract:

`Input:` A triple graph $\overline{G}$ and a graph arrow $\delta_S^+ : \overline{G}_S \to G'_S$ such that:

1. The triple graph $\overline{G} = \overline{G}_S \leftarrow \overline{G}_C \to \overline{G}_T$ is consistent
2. The graph arrow $\delta_S^+ : \overline{G}_S \to G'_S$ can be extended to a triple graph arrow $\delta^+ = (\delta_S^+, \delta_C^+, \delta_T^+) : \overline{G} \to G'$ using the rules of the TGG, i.e., $\exists\, \overline{G} \overset{*}{\Rightarrow} G'$

`Output:` A triple graph G' and a graph arrow $\delta_T^+ : \overline{G}_T \to G'_T$ such that:

1. The triple graph $G' = G'_S \leftarrow G'_C \to G'_T$ is consistent



$$\begin{tikzcd}
G_S \arrow[rr, leftrightarrow, "c \in \mathcal{C}"] & & G_T \\
G_S^* \arrow[u, "\delta_S^{*-}"] \arrow[d, "\delta_S^{*+}"'] & \overline{G}_S \arrow[l, dashed, "\delta^{\subseteq}"'] \arrow[ul, dashed, "\delta_S^-"] \arrow[dl, dashed, "\delta_S^+"] \arrow[r, dashed, leftrightarrow, "c^- \in \mathcal{C}"] & \overline{G}_T \arrow[u, dashed, "\delta_T^-"'] \arrow[d, dashed, "\delta_T^+"] \\
G'_S \arrow[rr, dashed, leftrightarrow, "c' \in \mathcal{C}"] & & G'_T
\end{tikzcd}$$

Fig. 27. Decomposition of `fpg` into two steps: `rollback` and `translate`

Instead of discussing concrete implementations of `rollback` and `translate` in detail (the interested reader is referred to Anjorin [1]), let us conclude this section by discussing (efficient) implementation strategies in general. First of all implementations *do* exist, i.e., `fpg` is complete, as it is (i) always possible to rollback to $\overline{G} = \emptyset$, and (ii) it must be possible to extend consistent G'_S to a consistent triple graph G' from scratch (definition of consistency).

Rolling back to $\emptyset$ for every input is obviously neither the best nor most efficient implementation possible. Hermann et al. [18] suggest to instead determine a maximally consistent subgraph by "re-marking" as many elements as possible from scratch, using the rules of the TGG. While this is certainly qualitatively better than rolling back to $\emptyset$, it scales directly with model size and not with delta size, i.e., a very small change for large models will take as long (or even longer) to propagate as reconstructing all models from scratch. By using auxiliary structures to track dependencies as a so-called precedence relation on elements, Lauder et al. [24] suggest an efficient algorithm that is not as information preserving as that of Hermann et al. [18] but is certainly better than a complete rollback to $\emptyset$. Further simplifying the efficient implementation of `rollback` is ongoing work.

Implementing `translate` is a simple matter of applying TGG rules and backtracking until G' is attained as required (a complete search with backtracking to correct wrong local decisions). Such a naïve implementation does not scale in practice, however, unless an underlying model repository database or similar engine is used for efficient backtracking. Existing strategies implement `translate` by imposing additional restrictions on the structure of the TGG. The simplest example for such a restriction is demanding that the set of translation rules of the TGG be *confluent*, i.e., that applicable rules can be applied in any order as the result of a forward translation is always the same. Even though this can be checked statically at design time via a critical pair analysis [15], this is a rather strong restriction in practice. A suggestion of how to achieve a similar effect with weaker restrictions has been made by Anjorin et al. [4], and this is what is currently implemented in eMoflon. A weakness of the precedence-based approach is, however, that only a rather small class of (negative) application conditions can be supported in TGG rules [6].

Given implementations of `rollback` and `translate`, proving that `fpg` is correct is typically straightforward as we have the rollback derivation $G \overset{*}{\Leftarrow} \overline{G}$ and the translation derivation $\overline{G} \overset{*}{\Rightarrow} G'$, guaranteeing that we remain in $\mathcal{L}(TGG)$. It is, however, often challenging to show that "clever" `rollback` implementations fulfil Point (4) of its contract in all cases and for all TGG language features, particularly (negative) application conditions.

4.3 Hippocraticness and Least Change as Current Challenges

Guaranteeing correct and complete TGG-based propagation functions is already non-trivial, especially if these functions are expected to be efficient enough for practical applications. Once one has a working synchroniser, however, the question of the "quality" of synchronisation results arises. Correctness alone does not guarantee that a synchroniser always produces the "best possible" result.

Concerning consistency restoration, there appears to be some (tenuous) consensus on the fact that changing "as little as possible" to restore consistency is typically what is wanted (see Cheney et al. [10] for a related discussion). The following property suggested by Meertens [27] and Stevens [33] is for a special but relatively clear case: a synchroniser is hippocratic if it can guarantee that it does nothing if doing nothing is correct.

Definition 19 (Hippocraticness). *Given a* symmetric delta lens $\lambda : \mathcal{M} \overset{\mathcal{R}}{\Longleftrightarrow} \mathcal{N}$ *over a triple space* $\mathcal{M} \xleftrightarrow{\mathcal{R}} \mathcal{N}$*, and a set of consistent corrs* $\mathcal{C} \subseteq \Delta_{MN}$*. The forward propagation function* $fpg : \ulcorner \longrightarrow \lrcorner$ *of the lens is* hippocratic *if (depicted diagrammatically to the left of Fig. 28):*

$$\forall (m \xleftrightarrow{c} n, \delta_M : m \rightarrow m') \in \ulcorner .\ \exists\, m' \xleftrightarrow{c' \in \mathcal{C}} n \Longrightarrow fpg(c, \delta_M) = (c', id_N) \in \lrcorner$$

$$\begin{tikzcd}
m \arrow[r, leftrightarrow, "c"] \arrow[d, "\delta_M"'] \arrow[dr, phantom, "\text{fpg}"] & n \arrow[d, dashed, "id_N"] \\
m' \arrow[r, leftrightarrow, dashed, "c' \in \mathcal{C}"] & n
\end{tikzcd}
\qquad
\begin{tikzcd}
m \arrow[r, leftrightarrow, "c"] \arrow[d, "\delta_M"'] \arrow[dr, phantom, "\text{fpg}"] & n \arrow[d, dashed, "\delta_N"] \arrow[ddr, dashed, bend left, "\overline{\delta}_N"] & \\
m' \arrow[r, leftrightarrow, dashed, "c' \in \mathcal{C}"] \arrow[drr, leftrightarrow, dashed, bend right, "\overline{c} \in \mathcal{C}"'] & n' & \\
 & & \overline{n}
\end{tikzcd}$$

Fig. 28. Hippocraticness and least change

In an attempt to generalise hippocraticness, one could demand a partial order $\leq$ on all possible consistent results, representing a preference of which change is considered the "least surprising" (see Cheney et al. [10] for a detailed discussion). In practice, this partial order could perhaps count the number of (possibly weighted) changes, or favour certain types of changes (attribute manipulation) over others (creation and deletion), etc. Change propagation then becomes an optimisation problem, with the requirement of determining the least surprising consistent result:

Definition 20 (Least Change). *Given a* symmetric delta lens $\lambda : \mathcal{M} \overset{\mathcal{R}}{\Longleftrightarrow} \mathcal{N}$ *over a triple space* $\mathcal{M} \xleftrightarrow{\mathcal{R}} \mathcal{N}$*, a set of consistent corrs* $\mathcal{C} \subseteq \Delta_{MN}$*, and a partial order* $\leq\ \subseteq \lrcorner \times \lrcorner$*. The forward propagation function* $fpg : \ulcorner \longrightarrow \lrcorner$ *of the lens adheres to the principle of* least change *if (depicted diagrammatically to the right of Fig. 28):*

$$\forall (m \xleftrightarrow{c} n, \delta_M : m \rightarrow m') \in \ulcorner .\ fpg(c, \delta_M) = (m' \xleftrightarrow{c' \in \mathcal{C}} n', \delta_N : n \rightarrow n') \Longrightarrow \nexists (m' \xleftrightarrow{\overline{c} \in \mathcal{C}} \overline{n}, \overline{\delta}_N : n \rightarrow \overline{n}).\ (\overline{c}, \overline{\delta}_N) \leq (c', \delta_N)$$

Hippocraticness and least change are current challenges for the TGG approach. Promising work in this direction include the idea of deriving additional "repair" rules from a TGG, with which consistency can sometimes be restored before applying `rollback` [20]. A further idea is to delay the deletion of elements by keeping them in a pool and attempting to reuse these elements marked for deletion in `translate`. Finally, Leblebici [25] are working on combining optimisation techniques with TGGs using, e.g., an integer linear programming solver to minimise an objective function and thus choose from multiple consistent solutions. Such an objective function can be defined by a TGG designer to capture a problem-specific notion of least change/surprise.

With an understanding of least change related limitations of TGG-based approaches, the practical best practice guidelines proposed by Anjorin et al. [2] can be followed to improve synchronisation behaviour.

Example. To demonstrate how hippocraticness and least change can be violated by rolling back and translating, two diagrams following the same schema as Fig. 27 are depicted in Figs. 29 and 30. In the first diagram (Fig. 29), the target (input) delta to be backward propagated is a change of the brand of a dosage from Ascriptin to Axotal. The interesting point here is that the source (output) component of the input and output triple are identical, i.e., the result

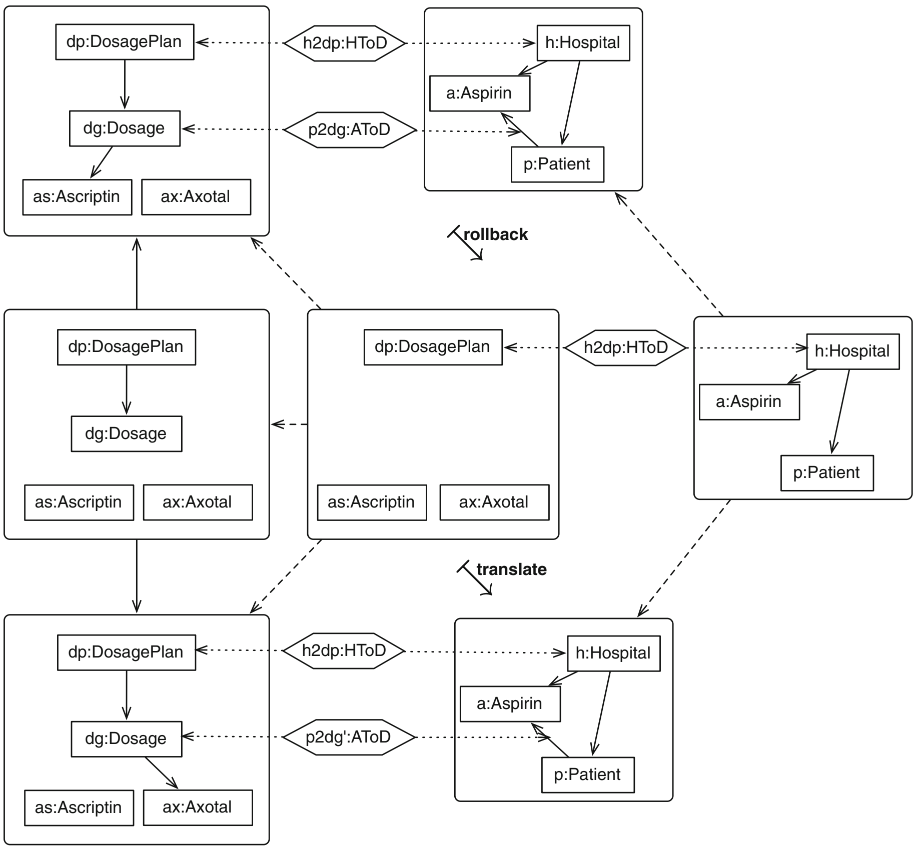


Fig. 29. Violation of hippocraticness

after changing the brand is already consistent and a hippocratic synchroniser need not do anything at all. Guaranteeing this with a TGG-based synchroniser, however, typically requires a costly consistency check that will in general scale with model and not delta size. Often faster (but not hippocratic) would be to `rollback` to a state that can be derived via a derivation with TGG rules, and then `translate` from this point as depicted in Fig. 29. Note how the edge in the source (output) model is unnecessarily deleted and re-created. While this might appear unremarkable for our simple example, such unnecessary changes can have a grievous ripple effect for more complex and realistic examples.

Even if hippocraticness is (artificially) enforced by always applying a (costly) consistency check before running `fpg`, the root of the problem will remain and manifest itself as undesirable synchronisation behaviour (effectively violating a suitable notion of least change). The problem is *not* solely due to a "bad" approximation of a consistent delta as was the case in Fig. 29. One could consider restricting the domain of `fpg` to only deltas that correspond directly to derivations with rules of the underlying TGG, but the example depicted in Fig. 30 shows that this does not solve the problem completely.

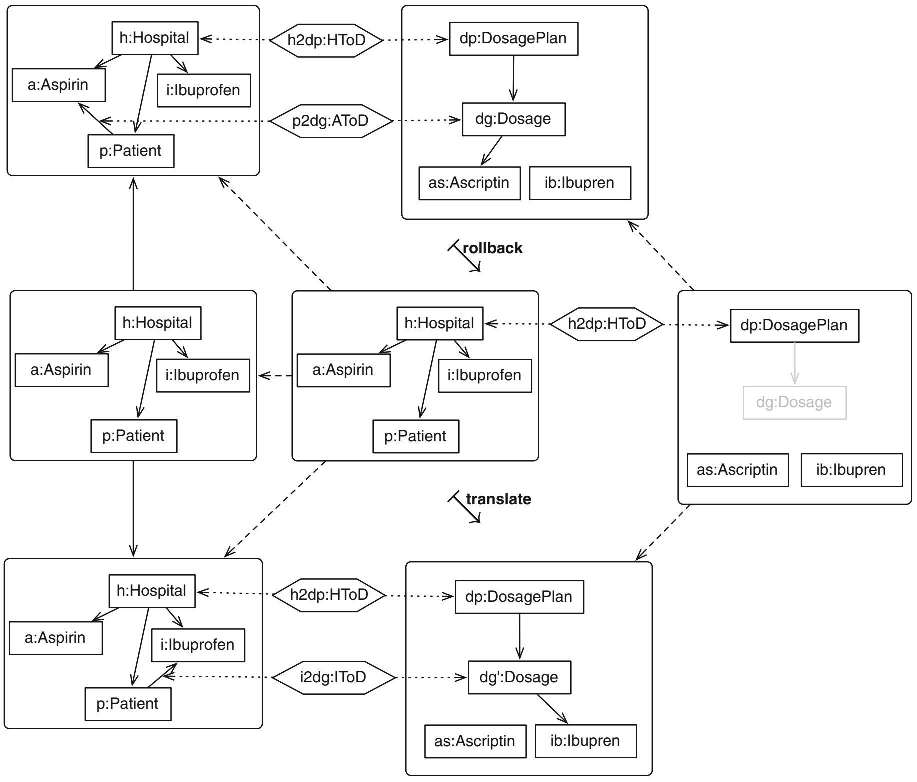


Fig. 30. Violation of least change

In this case, the source (input) delta to be propagated can be perfectly described via derivations with the rules of the TGG. Nonetheless, `rollback` is still too conservative as it enforces a consistent intermediate triple graph, deleting the dosage `dg`, just to have it recreated in `translate`. The "clever" fix of just deleting and creating the edge assigning `dg` its new brand (note the light grey elements) is beyond the TGG synchronisation algorithm presented in this chapter.

5 Conclusion and Future Perspectives

This chapter complements the work of Hermann, Diskin, and others [13,18], sharing the common goal of bringing the TGG and (sd-)lens frameworks closer together by revisiting the former as an implementation of the latter.

Although substantial work has been accomplished, e.g., by introducing a symmetric and delta-based version of the lens framework which fits better to TGGs, further steps are still required including incorporating non-determinism (see Diskin et al. [12] for first steps), and other consistency restoration functions besides forward and backward propagation. Besides the least change challenge highlighted in Sect. 4.3, there exist numerous suggestions of ways in which the TGG framework could be improved in the near future (see Anjorin et al. [3] for a detailed overview).

References

1. Anjorin, A.: Synchronization of models on different abstraction levels using triple graph grammars. Ph.D. thesis, Technische Universität Darmstadt (2014)
2. Anjorin, A., Leblebici, E., Kluge, R., Schürr, A., Stevens, P.: A systematic approach and guidelines to developing a triple graph grammar. In: Cunha, A., Kindler, E. (eds.) BX 2015, CEUR Workshop Proceedings, vol. 1396, pp. 81–95. CEUR-WS.org (2015)
3. Anjorin, A., Leblebici, E., Schürr, A.: 20 years of triple graph grammars: a roadmap for future research. ECEASST **73**, 1–20 (2016)
4. Anjorin, A., Leblebici, E., Schürr, A., Taentzer, G.: A static analysis of non-confluent triple graph grammars for efficient model transformation. In: Giese, H., König, B. (eds.) ICGT 2014. LNCS, vol. 8571, pp. 130–145. Springer, Cham (2014). https://doi.org/10.1007/978-3-319-09108-2_9
5. Anjorin, A., Saller, K., Lochau, M., Schürr, A.: Modularizing triple graph grammars using rule refinement. In: Gnesi, S., Rensink, A. (eds.) FASE 2014. LNCS, vol. 8411, pp. 340–354. Springer, Heidelberg (2014). https://doi.org/10.1007/978-3-642-54804-8_24
6. Anjorin, A., Schürr, A., Taentzer, G.: Construction of integrity preserving triple graph grammars. In: Ehrig, H., Engels, G., Kreowski, H.-J., Rozenberg, G. (eds.) ICGT 2012. LNCS, vol. 7562, pp. 356–370. Springer, Heidelberg (2012). https://doi.org/10.1007/978-3-642-33654-6_24
7. Anjorin, A., Varró, G., Schürr, A.: Complex attribute manipulation in TGGs with constraint-based programming techniques. In: Hermann, F., Voigtländer, J. (eds.) BX 2012, ECEASST, vol. 49, pp. 1–16. EASST (2012)

8. Awodey, S.: Category Theory. Oxford Logic Guides. Ebsco Publishing, Ipswich (2006)
9. Biermann, E., Ermel, C., Taentzer, G.: Precise semantics of EMF model transformations by graph transformation. In: Czarnecki, K., Ober, I., Bruel, J.-M., Uhl, A., Völter, M. (eds.) MODELS 2008. LNCS, vol. 5301, pp. 53–67. Springer, Heidelberg (2008). https://doi.org/10.1007/978-3-540-87875-9_4
10. Cheney, J., Gibbons, J., McKinna, J., Stevens, P.: Towards a principle of least surprise for bidirectional transformations. In: Cunha, A., Kindler, E. (eds.) BX 2015, CEUR Workshop Proceedings, vol. 1396, pp. 66–80. CEUR-WS.org (2015)
11. Czarnecki, K., Foster, J.N., Hu, Z., Lämmel, R., Schürr, A., Terwilliger, J.F.: Bidirectional transformations: a cross-discipline perspective. In: Paige, R.F. (ed.) ICMT 2009. LNCS, vol. 5563, pp. 260–283. Springer, Heidelberg (2009). https://doi.org/10.1007/978-3-642-02408-5_19
12. Diskin, Z., Eramo, R., Pierantonio, A., Czarnecki, K.: Incorporating uncertainty into bidirectional model transformations and their delta-lens formalization. In: Anjorin, A., Gibbons, J. (eds.) BX 2016, CEUR Workshop Proceedings, vol. 1571, pp. 15–31. CEUR-WS.org (2016)
13. Diskin, Z., Xiong, Y., Czarnecki, K., Ehrig, H., Hermann, F., Orejas, F.: From state- to delta-based bidirectional model transformations: the symmetric case. In: Whittle, J., Clark, T., Kühne, T. (eds.) MODELS 2011. LNCS, vol. 6981, pp. 304–318. Springer, Heidelberg (2011). https://doi.org/10.1007/978-3-642-24485-8_22
14. Ehrig, H., Ehrig, K., Ermel, C., Hermann, F., Taentzer, G.: Information preserving bidirectional model transformations. In: Dwyer, M.B., Lopes, A. (eds.) FASE 2007. LNCS, vol. 4422, pp. 72–86. Springer, Heidelberg (2007). https://doi.org/10.1007/978-3-540-71289-3_7
15. Ehrig, H., Ehrig, K., Prange, U., Taentzer, G.: Fundamentals of Algebraic Graph Transformation. Monographs in Theoretical Computer Science. An EATCS Series. Springer, Heidelberg (2006). https://doi.org/10.1007/3-540-31188-2
16. Foster, J.N., Greenwald, M.B., Moore, J.T., Pierce, B.C., Schmitt, A.: Combinators for bidirectional tree transformations: a linguistic approach to the view-update problem. ACM Trans. Program. Lang. Syst. **29**(3), 17 (2007)
17. Hermann, F., Ehrig, H., Orejas, F., Czarnecki, K., Diskin, Z., Xiong, Y.: Correctness of model synchronization based on triple graph grammars. In: Whittle, J., Clark, T., Kühne, T. (eds.) MODELS 2011. LNCS, vol. 6981, pp. 668–682. Springer, Heidelberg (2011). https://doi.org/10.1007/978-3-642-24485-8_49
18. Hermann, F., Ehrig, H., Orejas, F., Czarnecki, K., Diskin, Z., Xiong, Y., Gottmann, S., Engel, T.: Model synchronization based on triple graph grammars: correctness, completeness and invertibility. SoSym **14**(1), 241–269 (2015)
19. Hermann, F., et al.: Triple graph grammars in the large for translating satellite procedures. In: Di Ruscio, D., Varró, D. (eds.) ICMT 2014. LNCS, vol. 8568, pp. 122–137. Springer, Cham (2014). https://doi.org/10.1007/978-3-319-08789-4_9
20. Hildebrandt, S.: On the performance and conformance of triple graph grammar implementations. Ph.D. thesis, University of Potsdam (2014)
21. Hildebrandt, S., Lambers, L., Giese, H., Rieke, J., Greenyer, J., Schäfer, W., Lauder, M., Anjorin, A., Schürr, A.: A survey of triple graph grammar tools. In: Stevens, P., Terwilliger, J.F. (eds.) BX 2013, ECEASST, vol. 57. EASST (2013)
22. Johnson, M., Rosebrugh, R.D.: Unifying set-based, delta-based and edit-based lenses. In: Anjorin, A., Gibbons, J. (eds.) BX 2016, CEUR Workshop Proceedings, vol. 1571, pp. 1–13. CEUR-WS.org (2016)

23. Kappel, G., Langer, P., Retschitzegger, W., Schwinger, W., Wimmer, M.: Model transformation by-example: a survey of the first wave. In: Düsterhöft, A., Klettke, M., Schewe, K.-D. (eds.) Conceptual Modelling and Its Theoretical Foundations. LNCS, vol. 7260, pp. 197–215. Springer, Heidelberg (2012). https://doi.org/10.1007/978-3-642-28279-9_15
24. Lauder, M., Anjorin, A., Varró, G., Schürr, A.: Efficient model synchronization with precedence triple graph grammars. In: Ehrig, H., Engels, G., Kreowski, H.-J., Rozenberg, G. (eds.) ICGT 2012. LNCS, vol. 7562, pp. 401–415. Springer, Heidelberg (2012). https://doi.org/10.1007/978-3-642-33654-6_27
25. Leblebici, E.: Towards a graph grammar-based approach to inter-model consistency checks with traceability support. In: Anjorin, A., Gibbons, J. (eds.) BX 2016, CEUR Workshop Proceedings, vol. 1571, pp. 35–39. CEUR-WS.org (2016)
26. Leblebici, E., Anjorin, A., Schürr, A., Hildebrandt, S., Rieke, J., Greenyer, J.: A comparison of incremental triple graph grammar tools. In: Hermann, F., Sauer, S. (eds.) GT-VMT 2014, ECEASST, vol. 67. EASST (2014)
27. Meertens, L.: Designing constraint maintainers for user interaction. Technical report (1998)
28. Pratt, T.W.: Pair grammars, graph languages and string-to-graph translations. J. Comput. Syst. Sci. **5**(6), 560–595 (1971)
29. Radke, H., Arendt, T., Becker, J.S., Habel, A., Taentzer, G.: Translating essential OCL invariants to nested graph constraints focusing on set operations. In: Parisi-Presicce, F., Westfechtel, B. (eds.) ICGT 2015. LNCS, vol. 9151, pp. 155–170. Springer, Cham (2015). https://doi.org/10.1007/978-3-319-21145-9_10
30. Rose, L.M., Kolovos, D.S., Paige, R.F., Polack, F.A.C., Poulding, S.: Epsilon flock: a model migration language. SoSyM **13**(2), 735–755 (2014)
31. Schürr, A.: Specification of graph translators with triple graph grammars. In: Mayr, E.W., Schmidt, G., Tinhofer, G. (eds.) WG 1994. LNCS, vol. 903, pp. 151–163. Springer, Heidelberg (1995). https://doi.org/10.1007/3-540-59071-4_45
32. Schürr, A., Klar, F.: 15 years of triple graph grammars. In: Ehrig, H., Heckel, R., Rozenberg, G., Taentzer, G. (eds.) ICGT 2008. LNCS, vol. 5214, pp. 411–425. Springer, Heidelberg (2008). https://doi.org/10.1007/978-3-540-87405-8_28
33. Stevens, P.: Towards an algebraic theory of bidirectional transformations. In: Ehrig, H., Heckel, R., Rozenberg, G., Taentzer, G. (eds.) ICGT 2008. LNCS, vol. 5214, pp. 1–17. Springer, Heidelberg (2008). https://doi.org/10.1007/978-3-540-87405-8_1
34. Strüber, D., Rubin, J., Chechik, M., Taentzer, G.: A variability-based approach to reusable and efficient model transformations. In: Egyed, A., Schaefer, I. (eds.) FASE 2015. LNCS, vol. 9033, pp. 283–298. Springer, Heidelberg (2015). https://doi.org/10.1007/978-3-662-46675-9_19
35. Weber, J.H., Diemert, S., Price, M.: Using graph transformations for formalizing prescriptions and monitoring adherence. In: Parisi-Presicce, F., Westfechtel, B. (eds.) ICGT 2015. LNCS, vol. 9151, pp. 205–220. Springer, Cham (2015). https://doi.org/10.1007/978-3-319-21145-9_13

Modular Edit Lenses

Martin Hofmann†

LMU Munich, Munich, Germany

Abstract. This article is a reading guide to the theory of symmetric edit lenses by Pierce, Wagner, and the author, which form a general framework for the modular construction of bidirectional synchronizers and which generalize the popular lenses framework by Foster and Pierce to a truly symmetric, bidirectional setting.

The article describes both the state-based and the edit-based version, as well as an extended example instantiation involving tree-structured data. The main focus is on edit lenses and the categorical combinators which allow for their modular construction. The article is based on three original research papers [9–11, 22] and summarises these in a concise form but does not contain new scientific material.

1 Introduction

This article is based on a series of three lectures given by the author about his joint work with Benjamin Pierce and David Wagner on the topic of bidirectional synchronisation based on a symmetric version of lenses [7]. In addition to the slides that are still available on the school website this article fills in most of the oral explanations offered. For more detail the reader should consult the papers [9–11, 22] on which the lectures were based. This article does not contain any original material beyond those.

Bidirectional synchronisation between data given in two different representations is a ubiquitous task. Typical examples include different file systems, data on the web or in the cloud, different software models, different data formats.

In each case the first step is to specify the translation or more general correspondence between the two representations to be synchronised. Mutually inverse back-and-forth translations between the two representations, when available, are always the best option, but in many cases are not possible because each representation may contain additional data that is not reflected in the other representation.

In the classical framework of (asymmetric) lenses [7] one considers that one representation (the "source") contains more information than the other (the "view"). Thus, a lens from source set S to view set V comprises two functions

$$\begin{aligned} get &\in S \to V \\ put &\in V \times S \to S \end{aligned}$$

subject to the requirements that

$$get(put(v, s)) = s$$
$$put(get(s), s) = s$$

Thus, $get(s)$ extracts the V-view from $s \in S$ and $put(v, s)$ is the result of modifying the V-part of $s \in S$ with a new view $v \in V$ but leaving as they were the additional parts of s that are not reflected by the V-view.

Much research on those lenses has been devoted to the comfortable construction of useful lenses. Rather than writing *get* and *put* from scratch and establishing the two equational laws one can now define them from basic building blocks with combinators [7] and also derive a suitable *put* from a given *get* [21]. The other direction where *put* must be provided and *get* is derived and the laws are checked has also been explored [16] (see also this volume) and recently a reconciliation between the combinatory approach from [7] and the programming style with variables implicit in [20,21] has been achieved using category-theoretic ideas [17].

In the work described here, the asymmetry between source and view implicit in lenses is removed; a *symmetric lens* between two sets X and Y is defined as a stateful back- and forth transformation: it comprises a set C of states, the *complement*, and functions

$$putr \in X \times C \to Y \times C$$
$$putl \in Y \times C \to X \times C$$

subject to

$$\frac{putr(x, c) = (y, c')}{putl(y, c') = (x, c')} \qquad \text{(PutRL)}$$

$$\frac{putl(y, c) = (x, c')}{putr(x, c') = (y, c')} \qquad \text{(PutLR)}$$

In addition, C must contain a distinguished element $init \in C$.

Thus, to synchronise between repositories X and Y, one uses a shared state with values in C and initialised with *init*. In order to transport a new value $x \in X$ across the lens one invokes $putr(x, c)$ with c being the current state, initially *init*, which yields a pair (y, c') comprising the corresponding Y-value y and the new shared state c'. The shared state may be used to represent components of either X or Y that are not reflected in the other repository.

The article elaborates this idea as follows. The next section presents the theory of symmetric lenses as given in [9]. We show how they can be composed both sequentially and in parallel, how they relate to asymmetric lenses, and how to design lens combinators for recursive data structures. Section 3 then describes an extension of the lens framework with edit operations. Rather than transporting through the lens the complete new state of the repository one only transmits the changes that have occurred since the last synchronisation point. Besides possibly saving bandwidth in a distributed scenario this allows for more precise translations if instead of merely propagating the changes themselves we propagate the edit operation that led to them. This allows us, for example, to distinguish a swap between two entries from erasure followed by insertions of new entries. The allowed editing operations on some data structure or repository thus form an *edit language* and bidirectional synchronisations translate between them. Again, we describe the category-theoretic properties of these *edit lenses*, i.e., the ways in which they can be composed, and also combinators for the construction of lenses to operate on more complex data structures. It turns out that the view of such data structures as recursive types does not lend itself to an extension with edits. Instead, we show how to define meaningful edit lenses on so-called *containers* [1] which are closely related to the combinatory species from [15].

Section 4 describes an application of edit lenses or rather edit languages to XML processing using the automata-theoretic framework of [4,8]. We define a useful set of edits for XML trees and demonstrate how these can be typechecked against document type specifications represented by automata. It is based on the article [11].

We conclude with some suggestions for further research and projects.

2 Symmetric Lenses

We begin by introducing some notation for the symmetric lenses that have already been presented in the introduction.

Definition 1 (Symmetric lens). *A lens ℓ from X to Y (written $\ell \in X \leftrightarrow Y$) has three parts: a set of complements C, a distinguished element init $\in C$, and two functions*

$$\begin{aligned} \mathit{putr} &\in X \times C \to Y \times C \\ \mathit{putl} &\in Y \times C \to X \times C \end{aligned}$$

satisfying the following laws:

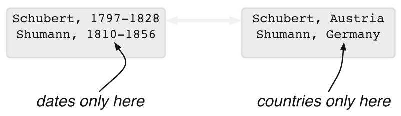
$$\frac{\mathit{putr}(x, c) = (y, c')}{\mathit{putl}(y, c') = (x, c')} \qquad (\textsc{PutRL})$$

$$\frac{\mathit{putl}(y, c) = (x, c')}{\mathit{putr}(x, c') = (y, c')} \qquad (\textsc{PutLR})$$

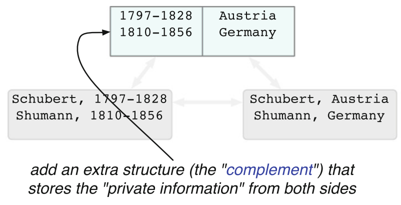
We use record notation writing $\ell.C$ for the complement set of ℓ, etc.

Figure 1 taken from [9] illustrates the operation of a symmetric lens in a concrete example.

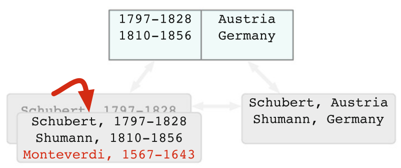
An important feature of symmetric lenses is that they can be sequentially composed as follows.



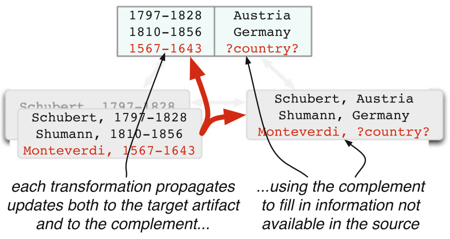
(a) Initial replicas



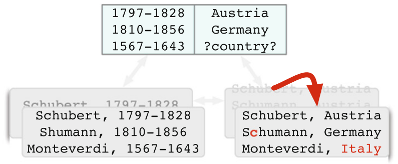
(b) Initial complement



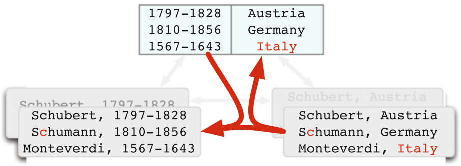
(c) One replica edited


1797-1828 Austria
1810-1856 Germany
1567-1643 ?country?
Schubert, 1797-1828
Shumann, 1810-1856
Monteverdi, 1567-1643
Schubert, Austria
Shumann, Germany
Monteverdi, ?country?
each transformation propagates updates both to the target artifact and to the complement...
...using the complement to fill in information not available in the source

(d) Propagating the edit



(e) Second replica is edited



(f) This change is propagated

Fig. 1. Behavior of a symmetric lens

Definition 2 (Lens composition).

$$\frac{k \in X \leftrightarrow Y \qquad \ell \in Y \leftrightarrow Z}{k;\ell \in X \leftrightarrow Z}$$

$$\begin{array}{lcl}
C & = & k.C \times \ell.C \\
init & = & (k.init, \ell.init) \\
putr(x,(c_k,c_\ell)) & = & \text{let } (y,c'_k) = k.putr(x,c_k) \text{ in} \\
 & & \text{let } (z,c'_\ell) = \ell.putr(y,c_\ell) \text{ in} \\
 & & (z,(c'_k,c'_\ell)) \\
putl(z,(c_k,c_\ell)) & = & \text{let } (y,c'_\ell) = \ell.putl(z,c_\ell) \text{ in} \\
 & & \text{let } (x,c'_k) = k.putl(y,c_k) \text{ in} \\
 & & (x,(c'_k,c'_\ell))
\end{array}$$

Viewing a symmetric lens as a pair of stateful back-and-forth functions this composition amounts to ordinary composition of functions.

It is important to note that earlier attempts at formalising symmetric bidirectional synchronisation had difficulties with composition.

Consider Meertens' constraint maintainers [18] which were later followed up in the context of model-driven design by Stevens [19], Diskin [5], and Xiong et al. [23]. A constraint maintainer between two sets X and Y has a *consistency relation* $R \subseteq X \times Y$ and two *consistency maintainers* $\triangleleft : X \times Y \to X$ and $\triangleright : X \times Y \to Y$ such that $(x \triangleleft y)\ R\ y$ and $x\ R\ (x \triangleright y)$ always hold, and such that $x\ R\ y$ implies $x \triangleleft y = x$ and $x \triangleright y = y$. Now, given such consistency relations $R \subseteq X \times Y$ and $R' \subseteq Y \times Z$ maintained by $\triangleright, \triangleleft$ and $\triangleright', \triangleleft'$, then to construct a maintainer for the composition $R; R'$, we need to, given $x \in X$ and $z \in Z$, come up with a $y \in Y$ that will allow us to use either of the existing maintainer functions, but there is no canonical way of doing that and indeed Meertens gives a concrete counterexample.

2.1 Composition and Equivalence

In order to render lens composition associative and also to iron out other unimportant detail it is useful to regard lenses up to behavioural equivalence:

Definition 3. *Two lenses $k, \ell \in X \leftrightarrow Y$ are equivalent, written $k \equiv \ell$, when there is a relation $R \subset k.S \times \ell.S$ and:*

$$\frac{s_k\ R\ s_\ell \qquad k.putr(a, s_k) = (b_k, s'_k) \qquad \ell.putr(a, s_\ell) = (b_\ell, s'_\ell)}{b_k = b_\ell \wedge s'_k\ R\ s'_\ell}$$

$$\frac{s_k\ R\ s_\ell \qquad k.putl(b, s_k) = (a_k, s'_k) \qquad \ell.putl(b, s_\ell) = (a_\ell, s'_\ell)}{a_k = a_\ell \wedge s'_k\ R\ s'_\ell}$$

$$\ell.init\ R\ k.init$$

It is not hard to see that equivalence of lenses is indeed an equivalence relation and that composition is associative up to $\equiv$. Thus, sets with equivalence classes of symmetric lenses as morphisms form a category.

There is an alternative characterisation of lens equivalence using observations: Given a lens $\ell \in X \leftrightarrow Y$, define a *put object* for ℓ to be a member of $X + Y$. Define a function *apply* taking a lens, an element of that lens' complement set, and a list of put objects, by pushing the list's elements through the lens beginning with the given element. Now, lenses $k, \ell \in X \leftrightarrow Y$ are *observationally equivalent* (written $k \approx \ell$) if, for every sequence of put objects $P \in (X + Y)^\star$ we have

$$apply(k, k.init, P) = apply(\ell, \ell.init, P).$$

One can then show that $k \approx \ell$ if and only if $k \equiv \ell$.

2.2 Useful Lenses

We now give a few instance of lenses that keep occurring in examples.

Every bijective function gives rise to a lens:

$$\frac{f \in X \to Y \qquad f \text{ bijective}}{\mathit{iso}_f \in X \leftrightarrow Y}$$

$$\begin{array}{lcl} C & = & \mathit{Unit} \\ \mathit{init} & = & () \\ \mathit{putr}(x, ()) & = & (f(x), ()) \\ \mathit{putl}(y, ()) & = & (f^{-1}(y), ()) \end{array}$$

For each set X we have the terminal lens

$$\frac{x \in X}{\mathit{term}_x \in X \leftrightarrow \mathit{Unit}}$$

$$\begin{array}{lcl} C & = & X \\ \mathit{init} & = & x \\ \mathit{putr}(x', c) & = & ((), x') \\ \mathit{putl}((), c) & = & (c, c) \end{array}$$

Being symmetric, lenses can be reversed:

$$\frac{\ell \in X \leftrightarrow Y}{\ell^{op} \in Y \leftrightarrow X}$$

$$\begin{array}{lcl} C & = & \ell.C \\ \mathit{init} & = & \ell.\mathit{init} \\ \mathit{putr}(y, c) & = & \ell.\mathit{putl}(y, c) \\ \mathit{putl}(x, c) & = & \ell.\mathit{putr}(x, c) \end{array}$$

The disconnect lens allows one to hide information from being transmitted and to store it in the complement.

$$\frac{x \in X \qquad y \in Y}{\mathit{disconnect}_{xy} \in X \leftrightarrow Y}$$

$$\mathit{disconnect}_{xy} \;=\; \mathit{term}_x; \mathit{term}_y^{op}$$

We also notice that asymmetric lenses give rise to symmetric lenses and thus can be seen as a special case of the latter.

Besides the sequential composition symmetric lenses can also be juxtaposed thus endowing the category of lenses with a *symmetric monoidal structure*.

$$\frac{k \in X \leftrightarrow Z \qquad \ell \in Y \leftrightarrow W}{k \otimes \ell \in X \times Y \leftrightarrow Z \times W}$$

$$\begin{array}{lcl}
C & = & k.C \times \ell.C \\
\mathit{init} & = & (k.\mathit{init}, \ell.\mathit{init}) \\
\mathit{putr}((x,y),(c_k,c_\ell)) & = & \mathsf{let}\ (z,c'_k) = k.\mathit{putr}(x,c_k)\ \mathsf{in} \\
 & & \mathsf{let}\ (w,c'_\ell) = \ell.\mathit{putr}(y,c_\ell)\ \mathsf{in} \\
 & & ((z,w),(c'_k,c'_\ell)) \\
\mathit{putl}((z,w),(c_k,c_\ell)) & = & \mathsf{let}\ (x,c'_k) = k.\mathit{putl}(z,c_k)\ \mathsf{in} \\
 & & \mathsf{let}\ (y,c'_\ell) = \ell.\mathit{putl}(w,c_\ell)\ \mathsf{in} \\
 & & ((x,y),(c'_k,c'_\ell))
\end{array}$$

This means, that we can represent lenses by wiring diagrams where composition corresponds to chaining and the tensor product to juxtaposition [14].

We do not know whether the category of symmetric lenses admits a *trace* in the sense of traced monoidal categories [2]. In the wiring analogy this would correspond to allowing feedback loops; algebraically, we would need to construct from a lens $\ell : X \times Y \leftrightarrow X \times Z$ a trace $\mathrm{tr}(\ell) : Y \leftrightarrow Z$.

Graphically, this corresponds to joining the two X-ends of ℓ with a "feedback" wire. This trace operation should validate all equations that hold "graphically" in this sense.

2.3 Sums and Lists

We now turn to the definition of lenses acting on disjoint union sets ("sums") and will then use them to synchronise lists and trees using recursion.

$$\frac{k \in X \leftrightarrow Z \qquad \ell \in Y \leftrightarrow W}{k \oplus \ell \in X + Y \leftrightarrow Z + W}$$

$$\begin{array}{lcl}
C & = & k.C \times \ell.C \\
\mathit{init} & = & (k.\mathit{init}, \ell.\mathit{init}) \\
\mathit{putr}(\mathsf{inl}(x),(c_k,c_\ell)) & = & \mathsf{let}\ (z,c'_k) = k.\mathit{putr}(x,c_k)\ \mathsf{in} \\
 & & (\mathsf{inl}(z),(c'_k,c_\ell)) \\
\mathit{putr}(\mathsf{inr}(y),(c_k,c_\ell)) & = & \mathsf{let}\ (w,c'_\ell) = \ell.\mathit{putr}(y,c_\ell)\ \mathsf{in} \\
 & & (\mathsf{inr}(w),(c_k,c'_\ell)) \\
\mathit{putl}(\mathsf{inl}(z),(c_k,c_\ell)) & = & \mathsf{let}\ (x,c'_k) = k.\mathit{putl}(z,c_k)\ \mathsf{in} \\
 & & (\mathsf{inl}(x),(c'_k,c_\ell)) \\
\mathit{putl}(\mathsf{inr}(w),(c_k,c_\ell)) & = & \mathsf{let}\ (y,c'_\ell) = \ell.\mathit{putl}(y,c_\ell)\ \mathsf{in} \\
 & & (\mathsf{inr}(y),(c_k,c'_\ell))
\end{array}$$

This yields another symmetric monoidal structure. We remark that there are various ways to define an *injection lens*, but none of them commutes in the natural way with $\oplus$.

$$inl_x \in X \leftrightarrow X + Y$$

Next, we come to a list mapping lens that synchronises between two lists in a point-wise fashion using a given lens to synchronise entries. The complement maintains a mapping between the respective positions. We use the notation Z^ω for the set of infinite lists with entries from Z and for $z \in Z$ we write z^ω for the list $\langle z, z, z, \dots \rangle \in Z^\omega$.

$$\frac{\ell \in X \leftrightarrow Y}{map(\ell) \in X^\star \leftrightarrow Y^\star}$$

$$\begin{array}{lll}
C & = & (\ell.C)^\omega \\
init & = & (\ell.init)^\omega \\
putr(x, c) & = & \mathsf{let}\ \langle x_1, \dots, x_m \rangle = x\ \mathsf{in} \\
 & & \mathsf{let}\ \langle c_1, \dots \rangle = c\ \mathsf{in} \\
 & & \mathsf{let}\ (y_i, c_i') = \ell.putr(x_i, c_i)\ \mathsf{in} \\
 & & (\langle y_1, \dots, y_m \rangle, \langle c_1', \dots, c_m', c_{m+1}, \dots \rangle) \\
putl & & \text{(similar)}
\end{array}$$

We can also define a lens akin to fold-right known from functional programming provided that we have a decreasing weight function. Given $\ell : Unit + X \times Z \leftrightarrow Z$ can define $\mathsf{fold}(\ell) : X^* \leftrightarrow Z$ such that

$$\begin{tikzcd}
Unit + X \times X^\star \arrow[r, "iso"] \arrow[d, "id_{Unit} \oplus (id_X \otimes \mathsf{fold}(\ell))"'] & X^\star \arrow[d, "\mathsf{fold}(\ell)"] \\
Unit + X \times Z \arrow[r, "\ell"'] & Z
\end{tikzcd}$$

Folding can be generalised to trees and yields the following useful special cases.

- $\mathsf{leaves} : Tree A \leftrightarrow [A]$
- $\mathsf{concat} : [[A]] \leftrightarrow [A]$
- $\mathsf{partition} : [A \uplus B] \leftrightarrow [A] \times [B]$
- $\mathsf{map} : (A \leftrightarrow B) \to ([A] \leftrightarrow [B])$

Notice in particular that the mapping lens defined explicitly before can alternatively obtained as an instantiation of the fold lens just as one knows from functional programming.

We remark that the versions of sums and lists described here are called *retentive*. This means that when we change sides or extend list length we use the "retained" values from the last time we were on that side/had that length.

There is also a *forgetful* version where upon shortening or changing sides we throw data away.

We also note that since lenses are self-dual one can easily define *hylomorphisms*: from $k : Z \leftrightarrow \mathit{Unit} + X \times Z$ and $\ell : \mathit{Unit} + X \times W \leftrightarrow W$ obtain $\mathit{Hy}(\ell, k) : Z \leftrightarrow W$ such that the obvious diagram commutes. One can also define iterators over more than one list and generalise the whole setup to other inductive datatypes.

For other non-inductive datatypes, in particular labelled graphs we can use containers as described in the next subsection.

2.4 Containers

Containers [1] are a general framework for data structures with *positions* holding *data*. They are closely related to combinatorial species [15], see also https://en.wikipedia.org/wiki/Combinatorial_species. Containers subsume inductive types like lists or trees are also containers, but not all containers are inductive types, e.g. labelled graphs are not.

Definition 4. *A container consists of*

- *a set I of shapes*
- *for each shape $i \in I$ a set $B(i)$ of positions.*

A container (I, B) defines a functor on Sets: $F_{I,B}(X) = \sum_{i \in I} X^{B(i)}$. An element of $F_{I,B}(X)$ consists of a shape i and for each position $p \in B(i)$ an element of X.

For lists we take $I = \mathbb{N}$ and $B(i) = \{0, \dots, i-1\}$. In this case, we have $F_{I,B}(X) = \sum_{n \in \mathbb{N}} X^{\{0,\dots,n-1\}}$ which is isomorphic to the set of lists X^*.

In the case of graphs with X-labelled nodes we take for I the set of unlabelled graphs and for $B(i)$ the nodes of i.

For a function $f : X \to Y$ the morphism part $F_{I,B}(f) : F_{I,B}(X) \to F_{I,B}(Y)$ applies f to each entry of a container. It generalises "map" on lists and trees.

Our aim is to generalise $F_{I,B}(f)$ to $F_{I,B}(\ell)$ with ℓ a lens. If the shape does not change we can just apply the lens position-wise. In order to deal with shape changes, say from i to i', we need some additional structure:

Definition 5. *An ordered container is a container (I, B) with the following additional structure.*

- *the set of positions I forms a partial ordering (poset) with binary meets on shapes. Here, $i \leq i'$ means that i is a sub-shape of i', e.g., sub-tree or shorter list. The meet of, say, two trees is the largest common sub-tree of the two.*
- *B forms a functor from the poset I to the category of sets: if $i \leq i'$ then there is an injection $B(i) \to B(i')$ written $b \mapsto b \mid i'$.*
- *If $p \in B(i)$ and $p' \in B(i')$ are equal in $B(j)$, thus, $i \leq j, i' \leq j$ then there must exist unique $q \in B(i \wedge i')$ such that p and p' arise from q by applying injections, i.e. B is a pullback preserving functor from I to the category of sets.*

We are now ready to define a container mapping lens. For (I, B) an ordered container we obtain the following lens combinator.

$$\frac{\ell \in X \leftrightarrow Y}{F_{I,B}(\ell) \in F_{I,B}(X) \leftrightarrow F_{I,B}(Y)}$$

```
$C =$
$\{t \in \prod_{i \in I} B(i) \to \ell.(C) \mid$
  $\forall i, i'.\ i \leq i' \supset \forall b{\in}B(i).\ t(i')(b|i') = t(i)(b)\}$
$init(i)(b) = \ell.init$
$putr((i, f), t) =$
let $f'(b) = \mathsf{fst}(\ell.putr(f(b), t(i)(b)))$ in
let $t'(j)(b) =$
   if $\exists b_0 \in B(i \wedge j).\ b_0|j = b$
   then $\mathsf{snd}(\ell.putr(f(b_0|i), t(j)(b)))$ in
   else $t(j)(b)$
$((i, f'), t')$
$putl$ (similar)
```

In [9] we also discuss a restructuring lens that can synchronise between different container types. See also Sect. 3.

2.5 Spans of Lenses

There is a close relationship between asymmetric and symmetric lenses via spans. This was noted in [9] and elaborated in detail by Johnson et al., see [12,13] and also this volume. Indeed, symmetric lenses are equivalent to *spans of lenses*, i.e., two views on a common source in the sense of asymmetric lenses.

Every asymmetric lens, i.e., a classical lens in the sense of Foster et al., gives rise to a symmetric lens.

$$\frac{\ell \in X \overset{a}{\leftrightarrow} Y}{\ell^{sym} \in X \leftrightarrow Y}$$

$$\begin{array}{lcl} C & = & \{f \in Y \to X \mid \forall y \in Y.\ \ell.get(f(y)) = y\} \\ init & = & \ell.create \\ putr(x, f) & = & (\ell.get(x), f_x) \\ putl(y, f) & = & \mathsf{let}\ x = f(y)\ \mathsf{in}\ (x, f_x) \end{array}$$

Here $f_x(y) = \ell.put(y, x)$. But not all symmetric lenses are of that form, e.g., the opposite of an asymmetric lens is not an asymmetric lens unless it comes from a bijection.

However, for any lens ℓ we can find *asymmetric* lenses k_1, k_2 such that

$$(k_1^{sym})^{op}; k_2^{sym} = \ell$$

For the "intermediate type", i.e., the common source of k_1 and k_2 we use the set of consistent triples for ℓ:

$$S_\ell = \{(x, y, c) \in X \times Y \times \ell.C \mid \ell.putr(x, c) = (y, c)\}$$

If $\ell : X \leftrightarrow Y$ then indeed $k_1 : S_\ell \to X$ and $k_2 : S_\ell \to Y$ where k_1 and k_2 are the obvious projections.

2.6 Summary

Symmetric lenses generalise the asymmetric lenses by Foster et al. to truly bidirectional synchronisation. They can be seen as symmetrised asymmetric lenses with a shared complement, as stateful back-and-forth transformations, and as spans of asymmetric lenses. We have also seen that it is useful to understand symmetric lenses modulo bisimulation which coincides with observational equivalence in a suitable sense. The symmetric lenses with sets as objects form a category whose structure we have begun to explore. In particular, we identified two kinds of tensor product one akin to cartesian product and the other one akin to disjoint union of sets. We also showed how the category of lenses supports inductive and container-like datatypes and thus bidirectional synchronisation between them.

Further work and possible projects in the area of symmetric lenses include the integration with programming and implementations, a further exploration of the possibility of defining lenses by recursion and an investigation of possible higher-order structure.

3 Edit Lenses

As we have seen, both asymmetric and symmetric lenses transport entire states across the link. They do not differentiate between small changes and altogether new setups of either side of a lens. Of course, the complement can be used to store parts or all of the current state of one or both sides of a lens and so one can work out differences using a diff-like algorithm. But, as is well-known such diffing methods invariably have to rely on heuristics and cannot, for example, distinguish between swaps and deletions followed by insertions of elements in a data structure. To remedy this, we introduce *edit languages* which are special kinds of monoids representing edits to a data structure. An *edit lens* then transports and translates elements of the edit language which are to be applied at the other end. We shall see that the ordinary symmetric lenses, henceforth called *state-based* lenses arise as a special case of edit lenses and that much of the existing category-theoretic structure carries over to the edit case. The powerful fold-combinators for inductive data types do, however, not generalise and must therefore be replaced with a selection of combinators for mapping, restructuring, and partitioning.

3.1 Edit Languages

An edit language for a set X is defined as a monoid M together with a partial monoid action of M on X:

$$\mathbf{1}_M \cdot m = m \cdot \mathbf{1}_M = m$$

$$m_1 \cdot (m_2 \cdot m_3) = (m_1 \cdot m_2) \cdot m_3$$

$$\mathbf{1}_M \odot x = x$$

$$(m_1 \cdot m_2) \odot x = m_1 \odot (m_2 \odot x)$$

The last two equations are understood in the sense of Kleene equality: if one side is defined so is the other and they are equal. In general, however, a term $m \odot x$ may be undefined.

If we have a set X we often use the notation ∂X for an associated edit language. For example, if X is a set we may consider the following edit language for the set $X^\star$ of lists over X. We first define *atomic edits* E for X^*:

- modify(p,dx) where $p \in \mathcal{N}, \mathrm{d}x \in \partial X$
- resize(i,j,x) where $i, j \in \mathcal{N}, x \in X$
- reorder(i,f) where f permutes $\{0, \ldots, i\}$

We then take $\partial(X^*)$ as the free monoid E^* of atomic edits. The action of atomic edits is implicit in the names of the edits, e.g. modify$(p, \mathrm{d}x)$ succeeds on $l = [x_0, \ldots, x_{n-1}]$ if $p < n$ and if $\mathrm{d}x \odot x_p$ succeeds. The result then is

$$[x_0, \ldots, x_{p-1}, \mathrm{d}x \odot x_p, x_{p+1}, \ldots, x_{n-1}]$$

This action is then extended to lists of edits by sequential application. In general, $\mathbf{1}$ represents the neutral edit that does nothing and the product $m \cdot m'$ represents the combined edit whose effect is to first apply m' and then m. Often, edits are construed as sequences of atomic edits, but this does not have to be so for sometimes, laws may be imposed between edits, e.g., certain edits might commute with one another or cancel each other out.

This happens, for example in the “overwrite” monoid which we use in order to embed state-based into edit based lenses. For each element $x \in X$ it has an edit whose effect is to replace the current state with x. Another example is given by product monoids which we use as an edit language for cartesian products. There we take the view that edits to different components of a cartesian product commute with each other.

Following algebraic tradition we call a pair of a set and an edit language, i.e., a monoid acting partially on it, a *module*. We have already sketched how every set X can be regarded as a module. Formally, $\partial X = X \cup \{\mathbf{1}\}$ and $x \odot x' = x$, etc.

We also explained how lists and cartesian products can be regarded as module constructions.

As for sums we adopt the following approach. Primitive edits for the disjoint union $X + Y$ are

$$\begin{aligned} G^{\oplus}_{X,Y} = & \{\mathsf{switch}_{iL}(\mathrm{d}x) \mid i \in \{L, R\}, \mathrm{d}x \in \partial X\} \\ & \cup \{\mathsf{switch}_{iR}(\mathrm{d}y) \mid i \in \{L, R\}, \mathrm{d}y \in \partial Y\} \\ & \cup \{\mathsf{stay}_L(\mathrm{d}x) \mid \mathrm{d}x \in \partial X\} \cup \{\mathsf{stay}_R(\mathrm{d}y) \mid \mathrm{d}y \in \partial Y\} \\ & \cup \{\mathsf{fail}\} \end{aligned}$$

We then define $\partial(X \oplus Y) = (G^{\otimes}_{X,Y})^*$. Thus, an edit operation is a sequence of primitive edit operations. Again, we omit the obvious definition of the action of those primitive elements.

We tried to impose reasonable equations on $\partial(X \oplus Y)$ so as to achieve the expected isomorphism $X \oplus (Y \oplus Z) \simeq (X \oplus Y) \oplus Z$, but did not succeed in doing so.

Now reconsider the aforementioned edit language for lists. We give its primitive edits with slightly more detail.

$$\begin{aligned} G^{\text{list}}_X = & \{\mathsf{mod}(p, \mathrm{d}x) \mid p \in \mathbb{N}^+, \mathrm{d}x \in \partial X\} \\ & \cup \{\mathsf{ins}(i) \mid i \in \mathbb{N}\} \quad \cup \quad \{\mathsf{del}(i) \mid i \in \mathbb{N}\} \\ & \cup \{\mathsf{reorder}(f) \mid \forall i \in \mathbb{N}. f(i) \text{ permutes } \{1, \ldots, i\}\} \\ & \cup \{\mathsf{fail}\} \end{aligned}$$

Here, an interesting open question is to what extent such an edit language can be derived automatically from the recursive definition of lists by $X^\star = 1 + X \times X^\star$. Other interesting points for thought include the question why it is important to have a partial action rather than a total one and why it is reasonable to distinguish between a monoid element, i.e., an edit and the function it induces on the underlying set. The paper on edit lenses [10] contains some ideas on these questions.

3.2 Stateful Homomorphisms

We now describe how to synchronise two edit languages. As in the state-based case the translations will be history-dependent and thus require a shared complement set.

Definition 6. *Given monoids M and N and a* complement set *C, a* stateful monoid homomorphism *from M to N over C is a function $h \in M \times C \to N \times C$ satisfying two laws:*

$$\frac{}{h(\mathbf{1}_M, c) = (\mathbf{1}_N, c)}$$

$$\frac{h(m, c) = (n, c') \qquad h(m', c') = (n', c'')}{h(m' \cdot_M m, c) = (n' \cdot_N n, c'')}$$

It is an interesting exercise or question to reformulate this definition as an instance of a standard homomorphism between a different kind of monoids.

3.3 Edit Lenses

We now come to the central definition of an edit lens. It operates between two modules and, in addition to a stateful homomorphisms comprises a consistency relation which, as we recall, was definable in the state-based setting but does not seem to be in the present case.

Definition 7. *An edit lens $\ell : \langle M, X\rangle \leftrightarrow \langle N, Y\rangle$ has:*

- *a complement set C of private data*
- *consistency relation $K \in X \times C \times Y$*
- *stateful monoid homomorphisms*

$$\Rrightarrow : M \times C \to N \times C$$

$$\Lleftarrow : N \times C \to M \times C$$

that preserve consistency

in the sense that if $(x, c, y) \in K$ and $\Rrightarrow(dx, c) = (dy, c')$ then $(dx\ x, c', dy\ y) \in K$ and symmetrically for $\Lleftarrow$. Again, we use dot notation to refer to the components of an edit lens.

As before, we may consider lenses up to equivalence thus obtaining a category.

Definition 8 (Lens equivalence). *Two lenses $k, \ell : X \leftrightarrow Y$ are* equivalent *(written $k \equiv \ell$) if, there exists a relation $S \subseteq X \times k.C \times \ell.C \times Y$ such that*

- *$(init_X, k.init, \ell.init, init_Y) \in S$;*
- *if $(x, c, d, y) \in S$ and $dx\ x$ is defined, then if $(dy_1, c') = k.\Rrightarrow(dx, c)$ and $(dy_2, d') = \ell.\Rrightarrow(dx, d)$, then $dy_1 = dy_2$ (qua monoid elements) and $(dx\ x, c', d', dy_1\ y) \in S$; and*
- *analogously for $\Lleftarrow$.*

Again, there is an equivalent definition with dialogues whose precise definition we omit. We now give the definitions of product and sum lenses.

$$\frac{k \in X \leftrightarrow Z \qquad \ell \in Y \leftrightarrow W}{k \otimes \ell \in X \otimes Y \leftrightarrow Z \otimes W}$$

$$\begin{array}{lcl}
C & = & k.C \times \ell.C \\
init & = & (k.init, \ell.init) \\
K & = & \{\ ((x,z),(c_k,c_\ell),(y,w)) \mid \\
 & & \qquad (x, c_k, y) \in k.K \\
 & & \qquad \wedge (z, c_\ell, w) \in \ell.K\ \} \\
\Rrightarrow((\mathrm{d}x, \mathrm{d}y), (c_k, c_\ell)) & = & ((\mathrm{d}x', \mathrm{d}y'), (c_k', c_\ell')) \\
\text{where } (\mathrm{d}x', c_k') & = & k.\Rrightarrow(\mathrm{d}x, c_k) \\
\text{and } (\mathrm{d}y', c_\ell') & = & \ell.\Rrightarrow(\mathrm{d}y, c_ell) \\
\Lleftarrow((\mathrm{d}x, \mathrm{d}y), (c_k, c_\ell)) & : & \text{analogous}
\end{array}$$

$$\frac{k \in X \leftrightarrow Y \qquad \ell \in Z \leftrightarrow W}{k \oplus \ell \in X \oplus Z \leftrightarrow Y \oplus W}$$

$$\begin{array}{lcl}
C & = & k.C + \ell.C \\
\mathit{init} & = & \mathsf{inl}(k.\mathit{init}) \\
K & = & \{(\mathsf{inl}(x), \mathsf{inl}(c), \mathsf{inl}(y)) \mid (x, c, y) \in k.K\} \\
 & \cup & \{(\mathsf{inr}(z), \mathsf{inr}(c), \mathsf{inr}(w)) \mid (z, c, w) \in \ell.K\} \\
c_k & = & k.\mathit{init} \\
c_\ell & = & \ell.\mathit{init} \\
\Rrightarrow_g(\mathsf{switch}_{LL}(\mathrm{d}x), \mathsf{inl}(c)) & = & \mathsf{let}\ (\mathrm{d}y, c') = k.\Rrightarrow(\mathrm{d}x, c_k) \\
 & & \mathsf{in}\ (\mathsf{switch}_{LL}(\mathrm{d}y), \mathsf{inl}(c')) \\
\Rrightarrow_g(\mathsf{switch}_{RL}(\mathrm{d}x), \mathsf{inr}(c)) & = & \mathsf{let}\ (\mathrm{d}y, c') = k.\Rrightarrow(\mathrm{d}x, c_k) \\
 & & \mathsf{in}\ (\mathsf{switch}_{RL}(\mathrm{d}y), \mathsf{inl}(c')) \\
\Rrightarrow_g(\mathsf{switch}_{LR}(\mathrm{d}z), \mathsf{inl}(c)) & = & \mathsf{let}\ (\mathrm{d}w, c') = \ell.\Rrightarrow(\mathrm{d}z, c_\ell) \\
 & & \mathsf{in}\ (\mathsf{switch}_{LR}(\mathrm{d}w), \mathsf{inr}(c')) \\
\Rrightarrow_g(\mathsf{switch}_{RR}(\mathrm{d}z), \mathsf{inr}(c)) & = & \mathsf{let}\ (\mathrm{d}w, c') = \ell.\Rrightarrow(\mathrm{d}z, c_\ell) \\
 & & \mathsf{in}\ (\mathsf{switch}_{RR}(\mathrm{d}w), \mathsf{inr}(c')) \\
\Rrightarrow_g(\mathsf{stay}_{L}(\mathrm{d}x), \mathsf{inl}(c)) & = & \mathsf{let}\ (\mathrm{d}y, c') = k.\Rrightarrow(\mathrm{d}x, c) \\
 & & \mathsf{in}\ (\mathsf{stay}_{L}(\mathrm{d}y), \mathsf{inl}(c')) \\
\Rrightarrow_g(\mathsf{stay}_{R}(\mathrm{d}z), \mathsf{inr}(c)) & = & \mathsf{let}\ (\mathrm{d}w, c') = \ell.\Rrightarrow(\mathrm{d}z, c) \\
 & & \mathsf{in}\ (\mathsf{stay}_{R}(\mathrm{d}w), \mathsf{inr}(c')) \\
\Rrightarrow_g(e, c) & = & (\mathsf{fail}, c) \text{ in all other cases} \\
\Lleftarrow_g & & \text{is analogous}
\end{array}$$

As already mentioned a list mapping *edit* lens can, as far as we know, be construed from a more general fold-pattern so we give it here as a primitive combinator.

$$\frac{\ell \in X \leftrightarrow Y}{\ell^* \in X^* \leftrightarrow Y^*}$$

$$\begin{array}{lcl}
C & = & \ell.C^* \\
\mathit{init} & = & \varepsilon \\
K & = & \{(x, c, y) \mid |x| = |c| = |y| \wedge \\
 & & \qquad \forall 1{\le}p{\le}|x|.\ (x_p, c_p, y_p) \in \ell.K\} \\
\Rrightarrow_g(\mathsf{mod}(p, \mathrm{d}x), c) & = & \mathsf{let}\ (\mathrm{d}y, c'_p) = \ell.\Rrightarrow(\mathrm{d}x, c_p)\ \mathsf{in} \\
 & & (\mathsf{mod}(p, \mathrm{d}y), c[p \mapsto c'_p])) \\
 & & \text{ when } p \le n \\
\Rrightarrow_g(\mathsf{mod}(p, \mathrm{d}x), c) & = & (\mathsf{fail}, c) \text{ when } p > n \\
\Rrightarrow_g(\mathsf{fail}, c) & = & (\mathsf{fail}, c) \\
\Rrightarrow_g(\mathrm{d}x, c) & = & (\mathrm{d}x, \mathrm{d}x\ c) \text{ in all other cases} \\
\Lleftarrow & & \text{similar}
\end{array}$$

As a concrete application we repeat in Fig. 2 a running example from [10] which illustrates the lenses we have seen so far.

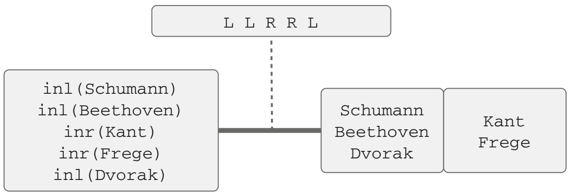
3.4 The Partition Lens

We seek a lens of the form

$$\boxed{\mathit{partition} \in (X \oplus Y)^* \leftrightarrow X^* \otimes Y^*}$$

$$\boxed{\quad\ldots\quad}$$

Once we have it, we can compose many important lenses on lists from it: e.g., we can use mapping to go from Z^* to $(X+Y)^*$ and then on using partition and we can further process $X^* \times Y^*$ by working on both parts separately using the tensor lens after which we can go back to a single list type W^* using the opposite of partitioning. Figure 3 (taken from [10]) contains the code for the partition lens. From a bird's eye perspective we recognise the complement $C = \{L, R\}^*$ which tells where the positions of the LHS belong. The consistent triples can thus be visualised as in



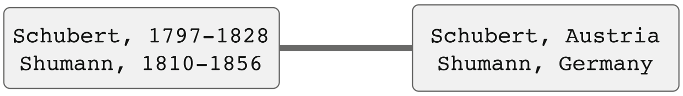
The example in Fig. 4 (where L, R are written `inl`, `inr`) gives an idea how edits are propagated. We refer to loc.cit. for details.

3.5 Containers

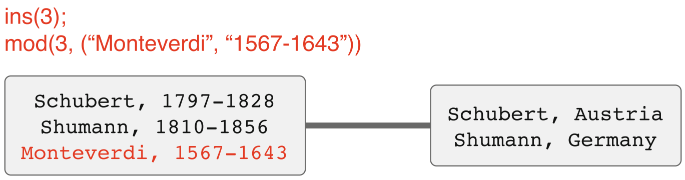
As in the state-based case we now show how to synchronise between container types using edit lenses. For technical reasons we need a slightly different formulation of containers, the difference being on the one hand that shapes and positions must be editable, i.e., modules, and on the other hand that positions are given as a single set rather than a family of sets which facilitates the definition of rearrangements.

Definition 9. *An* editable container, *henceforth "container", is given by the following data:*

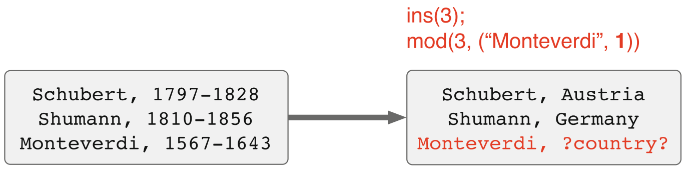
– *A* module I *of shapes additionally endowed with a partial order,*
– *A* fixed *set* P *of positions*
– *For each shape* i *a subset* $\mathsf{live}(i) \subseteq P$.



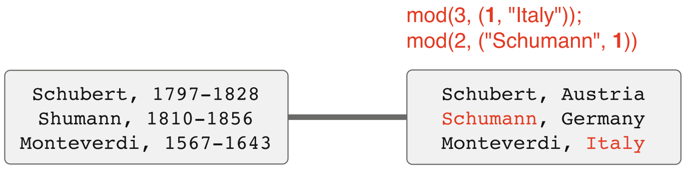
(a) initial replicas



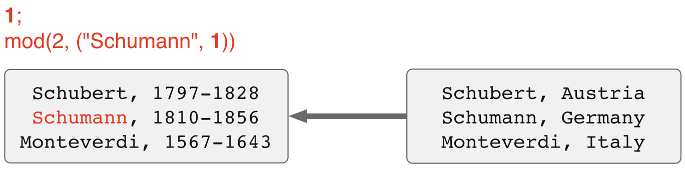
(b) a new composer is added to one replica



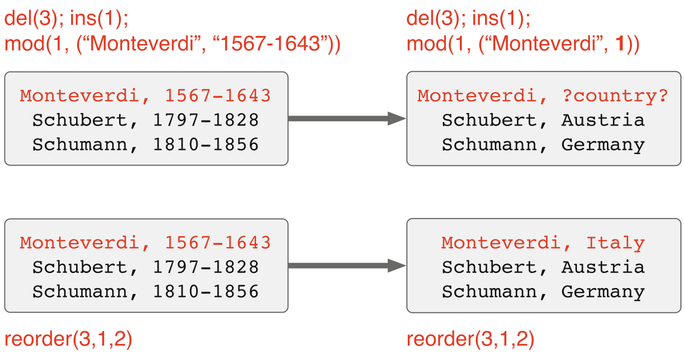
(c) the lens adds the new composer to the other replica



(d) the curator makes some corrections



(e) the lens transports a small edit



(f) two different edits with the same effect on the left

Fig. 2. Synchronising composers

$$\boxed{\mathit{partition} \in (X \oplus Y)^* \leftrightarrow X^* \otimes Y^*}$$

$$
\begin{array}{lll}
C & = & \{L, R\}^* \\
\mathit{init} & = & \varepsilon \\
K & = & \{(z, \mathsf{map}_{\mathsf{tagof}}(z), (\mathsf{lefts}(z), \mathsf{rights}(z))) \mid z \in (|X| + |Y|)^*\} \\
\\
\Rrightarrow_g(\mathsf{mod}(p, \mathrm{d}v), c) & = & (\mathsf{fail}, c) \text{ when } p > |c| \qquad (1) \\
\Rrightarrow_g(\mathsf{mod}(p, \varepsilon), c) & = & (\varepsilon, c) \text{ when } 1 \le p \le |c| \qquad (2) \\
\Rrightarrow_g(\mathsf{mod}(p, \mathrm{d}v\mathrm{d}vs), c) & = & (d'\,d, c'') \text{ where } 1 < n \quad (d, c') = \Rrightarrow_g(\mathsf{mod}(p, \mathrm{d}vs), c) \qquad (3) \\
 & & \qquad 1 \le p \le |c| \quad (d', c'') = \Rrightarrow_g(\mathsf{mod}(p, \mathrm{d}v), c') \\
\Rrightarrow_g(\mathsf{mod}(p, \mathsf{switch}_{jk}(\mathrm{d}v)), c) & = & (d_2 d_1 d_0, c[p \mapsto k]), \text{ where } (p_L, p_R) = \mathsf{count}(p, c) \quad d_0 = \mathsf{map}_{\lambda d.\,\mathsf{tag}(j,d)}(\mathsf{del}'(p_j)) \qquad (4) \\
 & & \qquad d_2 = \mathsf{tag}(k, \mathsf{mod}(p_k, \mathrm{d}v)) \quad d_1 = \mathsf{map}_{\lambda d.\,\mathsf{tag}(k,d)}(\mathsf{ins}'(p_k)) \\
\Rrightarrow_g(\mathsf{mod}(p, \mathsf{stay}_j(\mathrm{d}v)), c) & = & (\mathsf{tag}(j, \mathsf{mod}(p_j, \mathrm{d}v)), c), \text{ where } (p_L, p_R) = \mathsf{count}(p, c) \qquad (5) \\
\Rrightarrow_g(\mathsf{mod}(p, \mathsf{fail}), c) & = & (\mathsf{fail}, c) \qquad (6) \\
\Rrightarrow_g(\mathsf{ins}(i), c) & = & (\mathsf{left}(\mathsf{ins}(i)), \mathsf{ins}(i)\ c) \qquad (7) \\
\Rrightarrow_g(\mathsf{del}(i), c) & = & (d_1 d_0, \mathsf{del}(i)\ c), \text{ where } \quad c' = \mathsf{reverse}(c) \quad d_0 = \mathsf{left}(\mathsf{del}(n_L - 1)) \qquad (8) \\
 & & \qquad (n_L, n_R) = \mathsf{count}(i+1, c') \quad d_1 = \mathsf{right}(\mathsf{del}(n_R - 1)) \\
\Rrightarrow_g(\mathsf{reorder}(f), c) & = & (d_L d_R, c'), \text{ where } \quad h = \mathsf{iso}(c) \quad c' = \mathsf{reorder}(f)\ c \qquad (9) \\
 & & \qquad h' = \mathsf{iso}(c') \quad (n_L, n_R) = \mathsf{count}(|c|, c) \\
 & & \qquad h'' = h'^{-1}; f(|c|); h \quad f_k(n \ne n_k) = \lambda p.\,p \\
 & & \qquad d_L = \mathsf{left}(\mathsf{reorder}(f_L)) \quad f_L(n_L) = \mathsf{inl}; h''; \mathsf{out} \\
 & & \qquad d_R = \mathsf{right}(\mathsf{reorder}(f_R)) \quad f_R(n_R) = \mathsf{inr}; h''; \mathsf{out} \\
\Rrightarrow_g(\mathsf{fail}, c) & = & (\mathsf{fail}, c) \qquad (10) \\
\\
\Lleftarrow_g(\varepsilon, c) & = & (\varepsilon, c) \qquad (11) \\
\Lleftarrow_g(\mathrm{d}v\mathrm{d}vs, c) & = & (d'd, c'') \text{ when } n > 1, \text{ where } (d, c') = \Lleftarrow_g(\mathrm{d}vs, c) \quad (d', c'') = \Lleftarrow_g(\mathrm{d}v, c') \qquad (12) \\
\Lleftarrow_g(\mathsf{left}(\mathsf{mod}(p, \mathrm{d}x)), c) & = & (\mathsf{stay}_L(\mathsf{mod}(p', \mathrm{d}x)), c), \text{ where } p' = \mathsf{iso}(c)^{-1}(\mathsf{inl}(p)) \qquad (13) \\
\Lleftarrow_g(\mathsf{left}(\mathsf{reorder}(f)), c) & = & (\mathsf{reorder}(f'), c), \text{ where } g(\mathsf{inr}(p)) = \mathsf{inr}(p) \quad f'(n \ne |c|) = \lambda p.\,p \qquad (14) \\
 & & \qquad g(\mathsf{inl}(p)) = \mathsf{inl}(f(n_L)(p)) \quad f'(|c|) = h; g; h^{-1} \\
 & & \qquad (n_L, n_R) = \mathsf{count}(|c|, c) \quad h = \mathsf{iso}(c) \\
\Lleftarrow_g(\mathsf{left}(\mathsf{ins}(i)), c) & = & (\mathsf{ins}(i), \mathsf{ins}(i)\ c) \qquad (15) \\
\Lleftarrow_g(\mathsf{left}(\mathsf{del}(0)), c) & = & (\varepsilon, c) \qquad (16) \\
\Lleftarrow_g(\mathsf{left}(\mathsf{del}(i)), c) & = & (d''\ \mathsf{del}'(p), c''), \text{ where } h = \mathsf{iso}(c) \quad (n_L, n_R) = \mathsf{count}(|c|, c) \qquad (17) \\
 & & \qquad p = h^{-1}(\mathsf{inl}(n_L)) \quad (d'', c'') = \Lleftarrow_g(d', c') \\
 & & \qquad c' = \mathsf{del}'(p)\ c \quad d' = \mathsf{left}(\mathsf{del}(i-1)) \\
 & & \text{when } 0 < i \le n_L + 1 \\
\Lleftarrow_g(\mathsf{left}(\mathsf{del}(i)), c) & = & (\mathsf{fail}, c) \text{ otherwise} \qquad (18) \\
\Lleftarrow_g(\mathsf{left}(\mathsf{fail}), c) & = & (\mathsf{fail}, c) \qquad (19) \\
\\
\Lleftarrow_g(\mathsf{right}(\mathrm{d}y), c) & & \text{similar}
\end{array}
$$

Fig. 3. Partition lens, the code view.

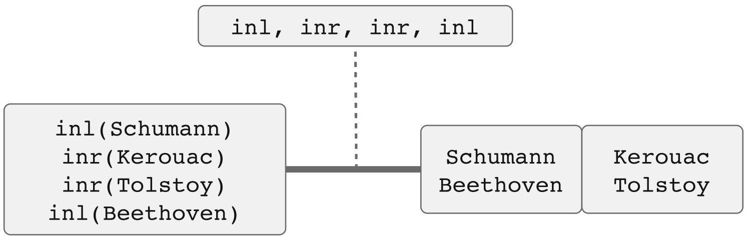
We notice that we can recover the traditional formulation by $B(i) = \{p \mid p \in \mathsf{live}(i)\}$ and as before, a container type $T = \langle I, P, \mathsf{live}\rangle$ defines a type operator by $T(X) = \sum_{i \in I} \mathsf{live}(i)$.

Let $T = \langle I, P, \mathsf{live}\rangle$ be a container type. An edit $\mathrm{d}i \in \partial I$ is an *insertion* if $\mathrm{d}i\ i \ge i$ whenever defined. It is a *deletion* if $\mathrm{d}i\ i \le i$ whenever defined. It is a *rearrangement* if $|\mathsf{live}(\mathrm{d}i\ i)| = |\mathsf{live}(i)|$ (same cardinality) whenever defined. We only employ edits from these three categories as ingredients of container edits; any other edits in the module will remain unused. This division of container edits into "pure" insertions, deletions, and rearrangements facilitates the later definition of lenses operating on such edits.

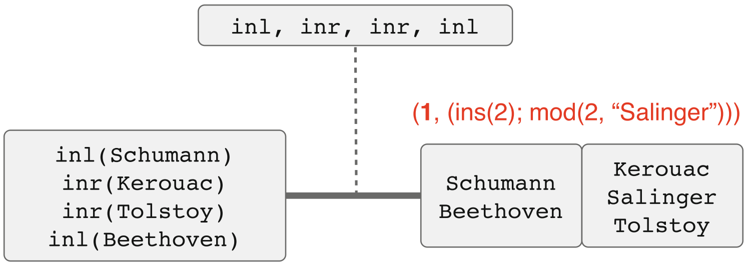
Definition 10. *We define the monoid of edits for a container type* $\langle I, P, \mathsf{live}\rangle$ *as the free monoid generated by*

- *Modifications:* $\mathsf{mod}(p, dx)$ *where* $p \in P$ *and* $dx \in \partial X$*,*
- *Insertions:* $\mathsf{ins}(di)$ *with* di *an insertion,*
- *Deletions:* $\mathsf{del}(di)$ *with* di *a deletion,*
- *Rearrangements:* $\mathsf{rearr}(di, f)$ *with* di *a rearrangement and* $f : \mathsf{live}(i) \simeq \mathsf{live}(di\, i)$*.*
- *Fail:* fail

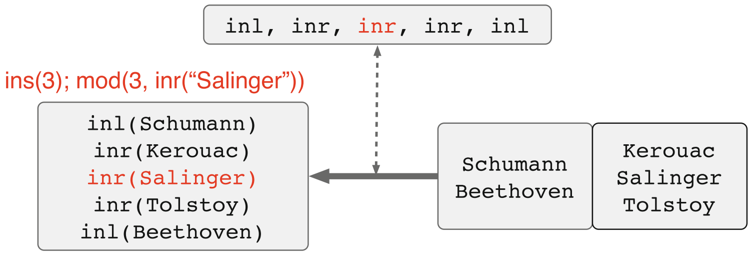
with the action given as follows.



(a) the initial replicas: a tagged list of composers and authors on the left; a pair of lists on the right; a complement storing just the tags



(b) an element is added to one of the partitions



(c) the complement tells how to translate the index

Fig. 4. Propagating edits across the partition lens

$$\begin{array}{l}
\mathsf{fail}\ (i, f)\ \textit{is always undefined} \\
\mathsf{mod}(p, dx)\ (i, f) = (i, f[p \mapsto dx\ \ f(p)])\ \textit{when}\ p \in \mathsf{live}(i) \\
\mathsf{ins}(di)\ (i, f) = (di\ \ i, f') \\
\qquad \textit{where}\ f'(p) = \textit{if}\ p \in \mathsf{live}(i)\ \textit{then}\ f(p)\ \textit{else}\ \textit{init}_X \\[1ex]
\mathsf{del}(di)\ (i, f) = (di\ \ i, f \restriction \mathsf{live}(di\ \ i)) \\
\mathsf{rearr}(di, f)\ (i, g) = (di\ \ i, g') \\
\qquad \textit{where}\ g'(p) = g(f(i)(p))
\end{array}$$

We remark that it would be interesting to consider further edits and also equations between these primitive edits. A container mapping lens which synchronises between two container types of the same kind but with different entry types can now be defined as follows.

$$\frac{\ell \in X \leftrightarrow Y \qquad T = \langle I, P, \mathsf{live}\rangle \ \text{a container type}}{T(\ell) \in T(X) \leftrightarrow T(Y)}$$

$$\begin{array}{lcl}
C & = & T(\ell.C) \\
\mathit{init} & = & (\mathit{init}_I, \lambda p.\ \ell.\mathit{init}) \\
\Rrightarrow_g(\mathsf{mod}(p, \mathrm{d}x), (i, f)) & = & (\mathsf{mod}(p, \mathrm{d}y), (i, f')) \\
\multicolumn{3}{l}{\qquad\qquad \text{when } p \in \mathsf{live}(i) \text{ and where}} \\
\multicolumn{3}{l}{\qquad\qquad f' = f[p{\mapsto}c'], (\mathrm{d}y, c') = \ell.\Rrightarrow(\mathrm{d}x, f(p))} \\
\Rrightarrow_g(\mathsf{mod}(p, \mathrm{d}x), (i, f)) & = & (\mathsf{fail}, (i, f)) \text{ if } p \notin \mathsf{live}(i) \\
\Rrightarrow_g(\mathsf{ins}(\mathrm{d}i), (i, g)) & = & (\mathsf{ins}(\mathrm{d}i), \\
 & & (\mathrm{d}i\ i, g[p{\mapsto}\ell.\mathit{init}])) \\
 & & \ \text{when } \mathrm{d}i\ i \text{ is defined} \\
\Rrightarrow_g(\mathsf{del}(\mathrm{d}i), (i, g)) & = & (\mathsf{del}(\mathrm{d}i), (\mathrm{d}i\ i, g{\restriction}\mathsf{live}(\mathrm{d}i\ i))) \\
 & & \ \text{when } \mathrm{d}i\ i \text{ is defined} \\
\Rrightarrow_g(\mathsf{rearr}(\mathrm{d}i, h), (i, g)) & = & (\mathsf{rearr}(\mathrm{d}i, h), \\
 & & (\mathrm{d}i\ i, \lambda p.g(h(i)(p)))) \\
 & & \ \text{when } \mathrm{d}i\ i \text{ is defined} \\
\Rrightarrow_g(\mathrm{d}z, c) & = & (\mathsf{fail}, c) \text{ in all other cases} \\
\Lleftarrow_g(-,-) & = & \text{analogous} \\
K & = & \{((i, f), (i, g), (i, f')) \mid i \in I \\
 & & \wedge (f(p), g(p), f'(p)) \in \ell.K\}
\end{array}$$

We also define a container restructuring lens between different container types $T = \langle I, P, \mathsf{live}\rangle$ and $T' = \left\langle I', P', \mathsf{live}'\right\rangle$. As input it requires an edit lens $\ell : I \leftrightarrow I'$ between the respective shapes and for each consistent triple $(i, c, i') \in \ell.K$ a bijection $f_{(i,c,i')}$ between $\mathsf{live}(i)$ and $\mathsf{live}(i')$.

$$\frac{T = \langle I, P, \mathsf{live}\rangle \text{ a container type} \quad T' = \langle I', P', \mathsf{live}'\rangle \text{ a container type} \quad \ell \in I \leftrightarrow I'}{[T, T'](\ell) \in T(X) \leftrightarrow T'(X)}$$

$$\begin{array}{lcl}
C & = & \ell.K \\
\mathit{init} & = & (\mathit{init}_I, \ell.\mathit{init}, \mathit{init}_{I'}) \\
\multicolumn{3}{l}{K = \{((i,f),(i,c,i'),(i',f'))} \\
\multicolumn{3}{l}{\qquad \mid (i,c,i') \in \ell.K \wedge \forall p \in \mathsf{live}'(i').f(f_{i,c,i'}(p)) = f'(p)\}} \\
\Rrightarrow_g(\mathsf{mod}(p, \mathrm{d}x), (i,c,i')) & = & (\mathsf{mod}(f^{-1}_{i,c,i'}(p), \mathrm{d}x), (i,c,i') \\
 & & \text{when } p \in \mathsf{live}(i) \\
\Rrightarrow_g(\mathsf{ins}(\mathrm{d}i), (i,c,i')) & = & (\mathsf{rearr}(\mathbf{1}, f_i)\mathsf{ins}(\mathrm{d}i'), \\
 & & (\mathrm{d}i\ i, c', \mathrm{d}i'\ i')) \\
\Rrightarrow_g(\mathsf{del}(\mathrm{d}i), (i,c,i')) & = & (\mathsf{rearr}(\mathbf{1}, f_d)\mathsf{del}(\mathrm{d}i'), \\
 & & (\mathrm{d}i\ i, c', \mathrm{d}i'\ i')) \\
\Rrightarrow_g(\mathsf{rearr}(\mathrm{d}i, f), (i,c,i')) & = & (\mathsf{rearr}(\mathrm{d}i', f_r), \\
 & & (\mathrm{d}i\ i, c', \mathrm{d}i'\ i'))
\end{array}$$

in the last three clauses: $(\mathrm{d}i', c') = \ell.\Rrightarrow(\mathrm{d}i, c)$.

Three families of bijections f_i, f_d, f_r must be chosen in such a way that the container edits in which they appear are well-formed (this is possible since $\mathrm{d}i'$ is an insertion, deletion, or rearrangement as appropriate) and such that the following three constraints are satisfied: in each case i, i', etc., refer to the current values from above and $p \in \mathsf{live}'(\mathrm{d}i'\ i')$ is an arbitrary position.

$$\begin{array}{l}
f_i(\mathrm{d}i'\ i')(p) = f^{-1}_{i,c,i'}(f_{\mathrm{d}i\ i, c', \mathrm{d}i'\ i'}(p)) \\
\qquad\qquad\qquad \text{when } f_{\mathrm{d}i\ i, c', \mathrm{d}i'\ i'}(p) \in \mathsf{live}(i) \\
f_d(\mathrm{d}i'\ i')(p) = f^{-1}_{i,c,i'}(f_{\mathrm{d}i\ i, c', \mathrm{d}i'\ i'}(p)) \\
f_r(\mathrm{d}i'\ i')(p) = f^{-1}_{i,c,i'}(f(i)(f_{\mathrm{d}i\ i, c', \mathrm{d}i'\ i'}(p)))
\end{array}$$

As an example of container restructuring Fig. 5 demonstrates a lens for in-order flattening of a tree. The underlying lens ℓ on shapes is trivial and merely ensures equal number of nodes. In addition it specifies some policy where to add and remove tree nodes. We assume that this happens by filling levels from the left. The bijections $f_{i,c,i'}$ define a bijective correspondence between nodes of synchronised shapes. In the example, we choose the in-order correspondence indicated by the dotted lines. After inserting two fresh nodes we are temporarily in the inconsistent state in the top right of Fig. 5. After applying the corresponding f_i-bijection we get into the consistent state depicted in the bottom right.

3.6 Typed Edit Language

We have experimented with *typed edit languages* which should (tentatively!) comprise

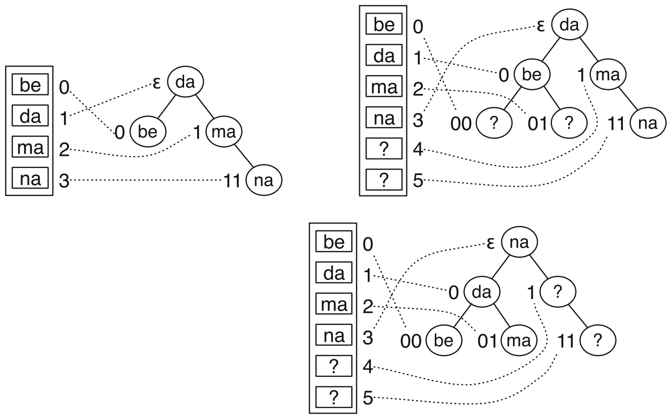


Fig. 5. Inserting two fresh nodes at the end of the list, propagation and restoration of consistency.

- a set T of "types"
- for each $t \in T$ a set $X(t)$ with distinguished element $\mathit{init}_{X(t)} \in X(t)$
- for any two types t, t' a set of edits $\partial X(t, t')$ with composition and identities, i.e. a category!
- an action of ∂X on X: if $e \in \partial X(t, t')$ and $x \in X(t)$ then $e.x \in X(t')$. I.e. $X(-)$ becomes a set-valued functor (presheaf).

This would allow one to distinguish lists by their lengths and thus to avoid border cases with head and tail of the empty list. It might also provide a solution to the problem that disjoint union for edit lenses is not associative by distinguishing the two components of a sum with types. There is also a relationship with the modelling of edits as categories in, for example, Johnson's work [12,13], this volume. There, however, the states, i.e., editable objects themselves, form the object of the category and not the types. As a result, an edit can only be applied to a single object (its domain in the category-theoretic sense). We prefer the viewpoint that an edit is an operation that as such can be applied to different objects. In the case of modules these objects are all elements of the underlying set, in the typed case these are the objects of the domain type of the edit.

Further exploration of the idea of typed edit languages is left as an open problem.

3.7 State-Based to Edit-Based and Back

In this section we explain how to relate state-based lenses with edit based ones. Let X be a set. The free monoid X^* acts on X by

$$(x_n \dots x_1)x = x_n$$

For $x \in X$ define module X_x as $X_x = (X, x, X^*)$. Let $\ell : X \leftrightarrow Y$ be a *state-based* symmetric lens and $\ell.\mathit{putr}(x, \ell.\mathit{missing}) = (y, \ell.\mathit{missing})$ be a consistent triple for ℓ. It is now an easy exercise to define an edit based lens between X_y and Y_y.

Conversely, if X is a module let a *differ* for X be a binary operation $\mathit{dif} \in X \times X \to \partial X$ satisfying $\mathit{dif}(x, x')x = x'$ and $\mathit{dif}(x, x) = \mathbf{1}$. Thus, a differ finds, for given states x, x', an edit operation $\mathrm{d}x$ such that $\mathrm{d}x\ x = x'$ and $\mathrm{d}x$ is "reasonable" at least in the sense that if $x = x'$ then the produced edit is minimal, namely $\mathbf{1}$.

It is instructive to ponder possible differs for the module X^*, say for trivial X.

Now, given an edit-based lens $\ell : X \leftrightarrow Y$ and differs for X and Y we obtain a state-based lens by

$$\overline{k : X \leftrightarrow Y}$$

$$\begin{array}{lcl}
C & = & \ell.K \\
\mathit{init} & = & (\mathit{init}_X, \ell.\mathit{init}, \mathit{init}_Y) \\
\mathit{putr}(x', (x, c, y)) & = & (\mathrm{d}x\ x, c', \mathrm{d}y\ y) \\
 & & \text{ where} \\
 & & \mathrm{d}x = \mathit{dif}(x, x') \text{ and } (\mathrm{d}y, c') = \ell.\Rrightarrow(\mathrm{d}x, c)
\end{array}$$

4 Information Trees

Before concluding this section outlines an approach for edit lens primitives for trees and (later on) graphs with unordered children as in XML or web applications. Various options beyond hand-crafting from the definitions could be containers modulo reordering of children positions. These, in fact, are the aforementioned combinatorial species which in this sense strictly generalise the containers.

Here, instead, we define unordered trees from scratch and then use tree automata to describe well-formed subsets. Using weakest preconditions we are then able to characterise those edit operations which preserve well-formedness and in this way obtain natural instance of typed edit languages.

Defining lenses on top of these edit languages is left for future work here.

4.1 Information Trees

An *information tree* is a finite tree with unordered children whose *edges* are labelled with words over a fixed finite alphabet Σ. Using linear notation with set braces $\{\!|\,|\!\}$ and arrows $\mapsto$ we have the following example

$$\{\!|\mathtt{name} \mapsto \{\!|\mathtt{John} \mapsto \{\!|\,|\!\}|\!\}, \mathtt{email} \mapsto \{\!|\mathtt{john@example.com} \mapsto \{\!|\,|\!\}|\!\}|\!\}$$

or in abbreviated form: same in abbreviated form:

$$\{\!|\mathtt{name} \mapsto \mathtt{John}, \mathtt{email} \mapsto \mathtt{john@example.com}|\!\}$$

We consider the following primitive edits.

$$\begin{aligned} e \ ::= \ & \mathit{insert}(t) \ \mid \\ & \mathit{hoist}(m,n) \ \mid \\ & \mathit{delete}(m) \ \mid \\ & \mathit{rename}(m,n) \ \mid \\ & \mathit{at}(n,e) \end{aligned}$$

where m, n are names, and t is a tree. Here, $\mathit{insert}(t)$ inserts t at the root (assuming that there are no name clashes); $\mathit{hoist}(m,n)$ removes the n-children from the m-labelled sub-trees of the root and adds ("hoists") them to the root; $\mathit{at}(n,e)$ applies e to the n-labelled children of the root and thus, allows the application of edits at arbitrary depths.

4.2 Sheaves Automata

We will specify tree types (document types) by a special kind of automata, namely the sheaves automata from [4,8].

Intuitively, a sheaves automaton has a set of states Q and for each $q \in Q$ a *sheaves formula* which partitions the allowed sub-trees into disjoint classes (recursively using states) and specifies an arithmetic constraint between the numbers of sub-trees falling into each class.

For example, we could have two states "person" and "address". A "person" has one "address" labelled `address` and many "persons" labelled `friend`. An "address" has sub-trees labelled `Street`, `Town`, etc. some of them optional.

Another example of a tree type definable with a sheaves automaton is a type FS of *file systems*:

$$\begin{aligned} &\mathrm{FS} ::= (.^* \to \mathrm{F} \mid \mathrm{D})^* \\ &\mathrm{F} ::= \mathtt{f} \to .^* \\ &\mathrm{D} ::= \mathtt{d} \to \mathrm{FS} \end{aligned}$$

Special naming conventions, file names starting with dot or ending with bin, etc., can also be accommodated. Further examples include tree-structured representation of program text and tree representation of game states (SGF).

It is known [4] that inclusion and nonemptiness of sheaves automata is decidable and that boolean operations are computable. Furthermore, sheaves sheaves automata have been presented as a type system [8] with subtyping where the algorithms for inclusion etc are used in order to do automatic type checking for functional programs (in a restricted syntax) producing trees.

Our result [11] asserts that for sheaves automaton A and tree edit e one can effectively compute a sheaves automaton $e.A$ such that

$$t \in L(e.A) \iff e.t \text{ fails} \vee e.t \in L(A)$$

Writing $e : A \to B$ to mean that $\forall t \in L(A).\ e.t$ defined $\Rightarrow e.t \in L(B)$, we have

$$e : A \to B \iff L(A) \subseteq L(e.B)$$

where the latter condition is decidable by the known results (as usual $L(-)$ stands for the set of accepted trees).

To see the equivalence we reason as follows: Suppose $e : A \to B$ and $t \in L(A)$. If $e.t$ is undefined then $e.t \in L(e.B)$ by definition of $e.B$. So, assume $e.t$ defined. By assumption $e.t \in L(B)$ and, again by definition of $e.B$, we have $t \in L(e.B)$.

For the converse, suppose $L(A) \subseteq L(e.B)$ and $t \in L(A)$ and $e.t$ defined. Then, $t \in L(e.B)$ and, since $e.t$ defined, $e.t \in L(B)$.

The construction of $e.B$ proceeds along the following lines.

- For every edit e define (by induction on e) a sheaves automaton D_e such that $L(D_e) = \{t \mid e.t \text{ undefined}\}$.
- For every edit e define (by induction on e) a sheaves automaton $e{\star}B$ such that whenever $e.t$ is defined then $t \in L(e \star B) \iff e.t \in L(B)$ (by anticipating the action of e). If $e.t$ is undefined then t may or may not be in $e \star B$.
- Then put $e.B = D_e \vee e \star B$ with $\vee$ denoting union construction for sheaves automata.

We remark that the union requires a product construction leading to a quadratic increase of the number of states and hence to exponential blowup upon nesting. It is plausible that this could be remedied by moving to a non-deterministic version of the sheaves automata.

For a concrete example, we consider the construction of $e.B$ when $e = \mathit{insert}(t')$: recall that this edit inserts t' at the root assuming the top-level labels of t' are not present. Thus, D_e checks that one of the top-level labels of t' *is* present (cardinality ≥ 1). To construct $e \star B$ we then add a new initial state s_0'. We then label s_0' just like s_0 (initial state of B but "as if t' is present"). E.g. if t' has an a label and s has an expression matching a then we replace the count variable x by $x + 1$. This also show the need for actual arithmetic constraints.

As usual, we define edits as lists of primitive edits and extend the notation $e : A \to B$ to edits. We notice that by iterating the calculation of weakest preconditions this latter judgement is also decidable.

4.3 Edit Languages for Information Trees

If A is a sheaves automaton we define an edit language A' with underlying set $L(A)$ and $\partial A' = \{e \mid e : A \to A\}$. We can automatically check whether $e \in \partial A'$ for a given edit e. This lends itself naturally to a typed generalisation where types are sheaves automata or a finite subset thereof and $\partial(A, B) = \{e \mid e : A \to B\}$.

5 Conclusion and Next Steps

We have shown the first steps towards editing and synchronising unordered trees defined by tree automata. Our approach integrates smoothly with existing edit lenses framework and combinators. In this setting the typed edit lenses considered earlier occur naturally and can be seen as synthesis with sd/delta lenses by Diskin et al. [6]. which replace edit languages by categories. There are several open ends that could lead to small research projects or even PhD topics. In particular, these are

– to further investigate categorical structure of lenses and edit lenses;
– to explore equations and optimisation, e.g., "deforestation";
– to further develop lenses based on information trees;
– to study connections with logic, e.g. whether specifications can be transported across a lens;
– to make connections to recent work about weak consistency models, e.g. [3] where consistency is not always restored in full but up to certain levels in order to save bandwidth.

References

1. Abbott, M., Altenkirch, T., Ghani, N.: Containers: constructing strictly positive types. Theor. Comput. Sci. **342**(1), 3–27 (2005)
2. Verity, D., Joyal, A., Street, R.: Traced monoidal categories. Math. Proc. Camb. Philos. Soc. **3**, 447–468 (1996). https://en.wikipedia.org/wiki/Traced_monoidal_category
3. Balegas, V., Li, C., Najafzadeh, M., Porto, D., Clement, A., Duarte, S., Ferreira, C., Gehrke, J., Leitão, J., Preguiça, N.M., Rodrigues, R., Shapiro, M., Vafeiadis, V.: Geo-replication: fast if possible, consistent if necessary. IEEE Data Eng. Bull. **39**(1), 81–92 (2016)
4. Dal Zilio, S., Lugiez, D.: XML schema, tree logic and sheaves automata. In: Nieuwenhuis, R. (ed.) RTA 2003. LNCS, vol. 2706, pp. 246–263. Springer, Heidelberg (2003). https://doi.org/10.1007/3-540-44881-0_18
5. Diskin, Z.: Algebraic models for bidirectional model synchronization. In: Czarnecki, K., Ober, I., Bruel, J.-M., Uhl, A., Völter, M. (eds.) MODELS 2008. LNCS, vol. 5301, pp. 21–36. Springer, Heidelberg (2008). https://doi.org/10.1007/978-3-540-87875-9_2
6. Diskin, Z., Xiong, Y., Czarnecki, K., Ehrig, H., Hermann, F., Orejas, F.: From state- to delta-based bidirectional model transformations: the symmetric case. Technical report GSDLAB-TR 2011–05-03. University of Waterloo, May 2011
7. Foster, J.N., Greenwald, M.B., Moore, J.T., Pierce, B.C., Schmitt, A.: Combinators for bi-directional tree transformations: a linguistic approach to the view update problem. ACM Trans. Program. Lang. Syst. **29**(3), 17 (2007). Extended abstract in Principles of Programming Languages (POPL), 2005
8. Foster, J.N., Pierce, B.C., Schmitt, A.: A logic your typechecker can count on: unordered tree types in practice. In: PLAN-X 2007, Programming Language Technologies for XML, an ACM SIGPLAN Workshop Colocated with POPL 2007, Nice, France, 20 January 2007, pp. 80–90 (2007)
9. Hofmann, M., Pierce, B.C., Wagner, D.: Symmetric lenses. In: ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL), Austin, Texas, January 2011. Full version to appear in the Journal of the ACM
10. Hofmann, M., Pierce, B.C., Wagner, D.: Edit lenses. In: Field, J., Hicks, M. (eds.) Proceedings of the 39th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2012, Philadelphia, Pennsylvania, USA, 22–28 January 2012, pp. 495–508. ACM (2012)

11. Hofmann, M., Pierce, B.C., Wagner, D.: Edit languages for information trees. ECE-ASST **57** (2013). https://doi.org/10.14279/tuj.eceasst.57.872
12. Johnson, M., Rosebrugh, R., Wood, R.: Lenses, fibrations, and universal translations. Math. Struct. Comput. Sci. **22**, 25–42 (2012)
13. Johnson, M., Rosebrugh, R.D., Wood, R.: Algebras and update strategies. J. Univ. Comput. Sci. **16**, 729–748 (2010)
14. Joyal, A., Street, R.: The geometry of tensor calculus, I. Adv. Math. **88**(1), 55–112 (1991)
15. Joyal, A.: Une théorie combinatoire des séries formelles. Adv. Math. **42**, 1–82 (1981). https://en.wikipedia.org/wiki/Combinatorial_species
16. Ko, H.-S., Zan, T., Hu, Z.: BiGUL: a formally verified core language for putback-based bidirectional programming. In: Erwig, M., Rompf, T. (eds.) Proceedings of the 2016 ACM SIGPLAN Workshop on Partial Evaluation and Program Manipulation, PEPM 2016, St. Petersburg, FL, USA, 20–22 January 2016, pp. 61–72. ACM (2016)
17. Matsuda, K., Wang, M.: Applicative bidirectional programming with lenses. In: Fisher, K., Reppy, J.H. (eds.) Proceedings of the 20th ACM SIGPLAN International Conference on Functional Programming, ICFP 2015, Vancouver, BC, Canada, 1–3 September 2015, pp. 62–74. ACM (2015)
18. Meertens, L.: Designing constraint maintainers for user interaction (1998). Manuscript
19. Stevens, P.: Bidirectional model transformations in QVT: semantic issues and open questions. In: Engels, G., Opdyke, B., Schmidt, D.C., Weil, F. (eds.) MODELS 2007. LNCS, vol. 4735, pp. 1–15. Springer, Heidelberg (2007). https://doi.org/10.1007/978-3-540-75209-7_1
20. Voigtländer, J.: Bidirectionalization for free! (pearl). In: Shao, Z., Pierce, B.C. (eds.) Proceedings of the 36th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2009, Savannah, GA, USA, 21–23 January 2009, pp. 165–176. ACM (2009)
21. Voigtländer, J., Hu, Z., Matsuda, K., Wang, M.: Combining syntactic and semantic bidirectionalization. In: Hudak, P., Weirich, S. (eds.) ICFP, pp. 181–192. ACM (2010)
22. Wagner, D.: Symmetric edit lenses: a new foundation for bidirectional languages. Ph.D. thesis. University of Pennsylvania (2014)
23. Xiong, Y., Song, H., Hu, Z., Takeichi, M.: Supporting parallel updates with bidirectional model transformations. In: Paige, R.F. (ed.) ICMT 2009. LNCS, vol. 5563, pp. 213–228. Springer, Heidelberg (2009). https://doi.org/10.1007/978-3-642-02408-5_15

Principles and Practice of Bidirectional Programming in BiGUL

Zhenjiang Hu(✉) and Hsiang-Shang Ko

National Institute of Informatics, Tokyo, Japan
{hu,hsiang-shang}@nii.ac.jp

Abstract. Putback-based bidirectional programming allows the programmer to write only one backward transformation, from which the unique corresponding forward transformation is derived for free. A key distinguishing feature of putback-based bidirectional programming is full control over the bidirectional behavior, which is important for specifying intended bidirectional transformations without any ambiguity. In this chapter, we will introduce BiGUL, a simple yet powerful putback-based bidirectional programming language, explaining the underlying principles and showing how various kinds of bidirectional application can be developed in BiGUL.

1 Putback-Based Bidirectional Programming

In this chapter, the kind of bidirectional transformations (BXs) we discuss is *aymmetric lenses* [8], which basically consist of a pair of transformations[1]: a *forward* transformation *get* producing a *view* from a *source*, and a *backward*, or *putback*, transformation *put* which takes a source and a possibly modified view, and reflects the modifications on the view to the source, producing an updated source. These two transformations should be *well-behaved* in the sense that they satisfy the following round-tripping laws:

$$put\ s\ (get\ s) = s \qquad \textsc{GetPut}$$

$$get\ (put\ s\ v) = v \qquad \textsc{PutGet}$$

The GetPut property requires that no change to the view should be reflected as no change to the source, while the PutGet property requires that all changes in the view should be completely reflected to the source so that the changed view can be successfully recovered by applying the forward transformation to the updated source.

The purpose of *bidirectional programming* is to develop well-behaved bidirectional transformations to solve various synchronization problems. A straightforward approach to bidirectional programming is to write two unidirectional

[1] The text of this section is adapted from the first author's FM 2014 paper [10].

Electronic supplementary material The online version of this chapter (https://doi.org/10.1007/978-3-319-79108-1_4) contains supplementary material, which is available to authorized users.

transformations. Although this ad hoc solution provides full control over both get and putback transformations, and can be realized using standard programming languages, the programmer needs to show that the two transformations satisfy the well-behavedness laws, and a modification to one of the transformations requires a redefinition of the other transformation as well as a new well-behavedness proof. To ease and enable maintainable bidirectional programming, it is preferable to write just a single program that can denote both transformations.

Lots of work [2,3,8,9,12,15,16] has been devoted to the *get-based* approach, allowing the programmer to write, mainly, the forward transformation *get*, and deriving a suitable putback transformation. While the get-based approach is friendly, a *get* function will typically not be injective, so there may exist many possible *put* functions that can be combined with it to form a valid BX. This ambiguity of *put* is what makes bidirectional programming challenging and unpredictable in practice. For specific domains where declarative approaches suffice, the get-based approach works fine, but when it comes to problems for which it is essential to precisely control *put* behavior, the get-based approach is inherently awkward: while most get-based languages/systems offer some features for programming *put* behavior, the programmer ends up having to break the *get*-based abstraction and figure out the *put* semantics of their *get* programs in excruciating detail to be able to reliably use these features, largely defeating the purpose of these languages/systems.

The main topic of this chapter is the *putback-based* approach to bidirectional programming. In contrast to the get-based approach, it allows the programmer to write a backward transformation *put* and derives a suitable *get* that can be paired with this *put* to form a bidirectional transformation. Interestingly, while *get* usually loses information when mapping from a source to a view, *put* must preserve information when putting back from the view to the source, according to the PutGet property.

Before explaining how to program *put* in practice, let us briefly review the foundations [5–7], showing that "putback" is the essence of bidirectional programming. We start by defining validity of *put* as follows:

Definition 1 (Validity of *put*). *We say that a put function is* valid *if there exists a get function such that both* GetPut *and* PutGet *are satisfied.*

The first interesting fact is that, for a valid *put*, there exists exactly one *get* that can form a BX with it. This is in sharp contrast to get-based bidirectional programming, where many *put*s may be paired with a *get* to form a BX.

Lemma 1 (Uniqueness of *get*). *Given a put function, there exists at most one get function that forms a well-behaved BX.*

The second interesting fact is that it is possible to check the validity of *put* without mentioning *get*. The following are two important properties of *put*.

– The first, which we call *view determination*, says that the equivalence of updated sources produced by a *put* implies equivalence of views that are put back.

$$\forall\, s, s', v, v'.\ put\ s\ v \;=\; put\ s'\ v' \;\Rightarrow\; v \;=\; v' \qquad \text{ViewDetermination}$$

Note that view determination implies that *put s* is injective (with $s = s'$).

– The second, which we call *source stability*, denotes a slightly stronger notion of surjectivity for every source:

$$\forall\, s.\ \exists\, v.\ put\ s\ v\ =\ s \qquad \textsc{SourceStability}$$

These two properties together provide an equivalent characterization of the validity of *put* [5].

Theorem 1. *A put function is valid if and only if it satisfies* ViewDetermination *and* SourceStability*.*

Practically, there are few languages supporting putback-based bidirectional programming. This is not without reason: as argued by Foster [7], it is more difficult to construct a framework that can directly support putback-based bidirectional programming.

In the rest of this chapter, we will introduce BiGUL [11] (pronounced "beagle"), a simple yet powerful putback-based bidirectional language, which grew out of some prior putback-based languages [13,14]. BiGUL is implemented as an embedded language in Haskell, and we will assume that the reader is reasonably familiar with Haskell. After briefly explaining how to install BiGUL in Sect. 2, we will introduce basic BiGUL programming in Sect. 3, and see a few more examples about lists in Sect. 4. We will then move on to the underlying principles in Sect. 5, explaining the design and implementation of BiGUL in detail. Those readers who are more interested in practical applications or want to see more examples first may safely skip Sect. 5 (which is rather long) and proceed to the last three sections, which will show how various bidirectional applications can be developed, including list alignment in Sect. 6, relational database updating in Sect. 7, and parsing and "reflective" printing in Sect. 8.

2 Preparation: Installing BiGUL

BiGUL is implemented as an embedded domain-specific language in Haskell, and this chapter assumes that the readers have some Haskell background. (If not, see https://wiki.haskell.org/Learning_Haskell for a list of resources for learning Haskell; for the Haskell environment, it is recommended to install Haskell Platform at https://www.haskell.org/platform/.) BiGUL has been released to Hackage, and the latest version (1.0.1 at the time of writing) can be installed using Cabal in the usual way, by executing the following in the command line:

```
$ cabal update
$ cabal install BiGUL
```

If you want to ensure compatibility with this chapter, you can instead install BiGUL-1.0.1 specifically by executing:

```
$ cabal install BiGUL-1.0.1
```

Now you can easily check whether BiGUL is correctly installed. First, create a simple file called `Test.hs` with the following content for importing BiGUL modules.

```
{-# LANGUAGE FlexibleContexts, TemplateHaskell, TypeFamilies #-}
import Generics.BiGUL
import Generics.BiGUL.Interpreter
import Generics.BiGUL.TH
import Generics.BiGUL.Lib
```

Then load it using GHCi.

```
$ ghci Test.hs
GHCi, version 7.10.3: http://www.haskell.org/ghc/  :? for help
[1 of 1] Compiling Main             ( Test.hs, interpreted )
Ok, modules loaded: Main.
*Main>
```

If you see the above message, congratulations on your successful installation.

To make it more convenient to play with the BiGUL code in this chapter, the Haskell source files for Sect. 3 (`Basic.hs`), Sect. 4 (`List.hs`), Sect. 6 (`Alignment.hs`), Sect. 7 (`Brul.hs`), and Sect. 8 (`BiYacc.hs`) are provided at:

https://bitbucket.org/prl_tokyo/bigul/src/master/SSBX16/

They are also available as electronic supplementary material to the online version of this chapter on SpringerLink. There are some dependencies among the files: `List.hs` imports `Basic.hs`, and `Brul.hs` imports `Alignment.hs`. The imported files should be present in the same directory as the files being loaded.

3 A Quick Tour of BiGUL

Intuitively, we can think of a bidirectional BiGUL program

$$bx :: BiGUL\ s\ v$$

as describing how to manipulate a state consisting of a source component of type s and a view component of type v; the goal is to embed all information in the view to proper places in the source. For each $bx :: BiGUL\ s\ v$, we can run it forwards by calling *get* and backwards by calling *put*:

$$\begin{aligned} &get\ \ bx :: s \qquad\quad \rightarrow Maybe\ v \\ &put\ bx :: s \rightarrow v \rightarrow Maybe\ s \end{aligned}$$

Here, *get bx* is a function mapping a source to a view, which can possibly fail: it either returns a successfully computed view wrapped in the *Just* constructor of *Maybe*, or signifies failure by producing the *Nothing* constructor. On the other hand, *put bx* accepts an original source and uses a view to update it to get an updated source (and might fail as well).

In BiGUL, it suffices for the programmer to write the *put* behavior (i.e., how to use a view to update the original source to a new source), and the (unique) *get* behavior is obtained for free. The core of BiGUL consists of a small number of primitives and combinators for constructing well-behaved bidirectional transformations, which we introduce below.

3.1 Skip

The first primitive for writing *put* is

$$Skip :: (s \to v) \to BiGUL\ s\ v$$

The put behavior of *Skip f* keeps the source unchanged, provided that the view is computable from the source by f (while in the get direction, the view is fully computed by applying function f to the source). Consider a simple *put* defined by *Skip square* where

$$\begin{aligned} &square \quad :: Num\ a \Rightarrow a \to a \\ &square\ x = x * x \end{aligned}$$

We can test its put behavior as follows:

```
*Basic> put (Skip square) 10 100
Just 10
```

It first checks if the view 100 is the square of the source 10. If that is the case, the original source is returned. But if the view is changed, say to 250, it should produce *Nothing*:

```
*Basic> put (Skip square) 10 250
Nothing
```

To see why *put* produces *Nothing*, we may use *putTrace* instead of *put* to get more information:

```
*Basic> putTrace (Skip square) 10 250
view not determined by the source
```

Each putback transformation in BiGUL is equipped with a unique *get* for doing forward transformation. We can test the *get* behavior as follows:

```
*Basic> get (Skip square) 5
Just 25
```

In prose: doing the forward transformation of *Skip square* on the source 5 gives the view 25. If *get* fails, we can also use *getTrace* to see more information about the failure, analogous to *putTrace*.

As a simple exercise, can you see what the following *skip1* does?

$$\begin{aligned} &skip1 :: BiGUL\ s\ () \\ &skip1 = Skip\ (const\ ()) \end{aligned}$$

3.2 Replace

The second primitive is

$$Replace :: BiGUL\ s\ s$$

which completely replaces the source with the view. For instance,

```
*Basic> put Replace 1 100
Just 100
```

uses the view 100 to replace the source 1 and gets a new source 100.

3.3 Product

If we want to use a view pair (v_1, v_2) to update a source pair (s_1, s_2), we can write *Prod bx1 bx2* or *bx1 'Prod' bx2*, a product of two putback transformations *bx1* and *bx2*, to use v_1 to update s_1 with *bx1* and v_2 to s_2 with *bx2*.

$$\mathit{Prod} :: \mathit{BiGUL}\ s_1\ v_1 \rightarrow \mathit{BiGUL}\ s_2\ v_2 \rightarrow \mathit{BiGUL}\ (s_1, s_2)\ (v_1, v_2)$$

For instance, we can use *Prod* to combine *Skip* and *Replace* to put a view pair into a source pair.

```
*Basic> put (skip1 `Prod' Replace) (5,1) ((),100)
Just (5,100)
```

Generally, we can use nested *Prod*s to describe a complicated structural mapping:

```
*Basic> put ((skip1 `Prod' Replace) `Prod' Replace) ((5,1),2)
    (((),100),200)
Just ((5,100),200)
```

3.4 Source/View Rearrangement

So far, the source and view have been of the same structure. What if we wish to put a view (v_1, v_2) into a source of a different structure, say $((s_0, s_1), s_2)$, to replace s_1 by v_1 and s_2 by v_2? To do that, we need to rearrange the source and view into the same structure, and BiGUL provides a way of rearranging either the source or view through a "simple" λ-expression e:

$$\begin{aligned}&^{\$}(\mathit{rearrS}\ [\![\, e :: s_1 \rightarrow s_2 \,]\!]) :: \mathit{BiGUL}\ s_2\ v\ \ \rightarrow \mathit{BiGUL}\ s_1\ v\\ &^{\$}(\mathit{rearrV}\ [\![\, e :: v_1 \rightarrow v_2 \,]\!]) :: \mathit{BiGUL}\ s\ \ v_2 \rightarrow \mathit{BiGUL}\ s\ \ v_1\end{aligned}$$

The "simple" λ-expression e should be wrapped inside Template Haskell quasiquotes $[\![\ldots]\!]$ (written as `[| ... |]` in plain text Haskell); it is then processed and expanded by *rearrS* or *rearrV* to "core" BiGUL code, which is spliced (pasted) into the invocation site by Template Haskell, as instructed by $^{\$}(\ldots)$. By "simple" we mean that there should be no wildcards '_' in the argument pattern, and that the body can only contain the argument variables and constructors, and must mention all the argument variables. We will discuss the details later in Sect. 5.6. Returning to the problem of putting a pair into a triple, we may define the following putback transformation

$$\begin{aligned}\mathit{putPairOverNPair}\ &::\ (\mathit{Show}\ s_0, \mathit{Show}\ s_1, \mathit{Show}\ s_2)\\ &\Rightarrow \mathit{BiGUL}\ ((s_0, s_1), s_2)\ (s_1, s_2)\end{aligned}$$

$$\begin{aligned}\mathit{putPairOverNPair} = {}&{}^{\$}(\mathit{rearrV}\ [\![\, \lambda(v_1, v_2) \to (((), v_1), v_2)\,]\!])^{\$} \\ &(\mathit{skip1}\ \text{`}\mathit{Prod}\text{'}\ \mathit{Replace})\ \text{`}\mathit{Prod}\text{'}\ \mathit{Replace}\end{aligned}$$

by first rearranging the view (v_1, v_2) to a triple $(((), v_1), v_2)$ with the same structure as the source, and then using $(\mathit{skip1}$ '*Prod*' *Replace*) '*Prod*' *Replace* to put the arranged view $(((), v_1), v_2)$ into the source $((s_0, s_1), s_2)$. The type context $(\mathit{Show}\ s_0, \mathit{Show}\ s_1, \mathit{Show}\ s_2)$ above is required by BiGUL for printing debugging messages. And note that the two '\$' signs in the definition off *putPairOverNPair* have different meanings: the first one marks the beginning of a Template Haskell splice, while the second one is the low-precedence application operator.

The mechanism of source/view rearrangement enables us to process algebraic data structures such as lists and trees, by mapping an algebraic structure to the (nested) pair structure. The following example uses the view to replace the first element of a nonempty source list:

$$\begin{aligned}&\mathit{pHead} :: \mathit{Show}\ s \Rightarrow \mathit{BiGUL}\ [s]\ s \\ &\mathit{pHead} = {}^{\$}(\mathit{rearrS}\ [\![\, \lambda(s : ss) \to (s, ss)\,]\!])^{\$} \\ &\qquad\quad {}^{\$}(\mathit{rearrV}\ [\![\, \lambda v \to (v, ())\,]\!])^{\$} \\ &\qquad\qquad \mathit{Replace}\ \text{`}\mathit{Prod}\text{'}\ \mathit{skip1}\end{aligned}$$

It rearranges the source (a nonempty list) to a pair with its head element s and its tail ss, and the view v to a pair $(v, ())$, so that we can use v to replace s and $()$ to keep ss.

```
*Basic> put pHead [1,2,3,4] 100
Just [100,2,3,4]
```

What if we wish to define a general putback transformation that uses the view to replace the ith element of the source list? We can define it recursively as follows:

$$\begin{aligned}&\mathit{pNth} \quad :: \mathit{Show}\ s \Rightarrow \mathit{Int} \to \mathit{BiGUL}\ [s]\ s \\ &\mathit{pNth}\ i = \mathbf{if}\ i == 0\ \mathbf{then}\ \mathit{pHead} \\ &\qquad\qquad\qquad \mathbf{else}\ \ {}^{\$}(\mathit{rearrS}\ [\![\, \lambda(x : xs) \to (x, xs)\,]\!])^{\$} \\ &\qquad\qquad\qquad\qquad {}^{\$}(\mathit{rearrV}\ [\![\, \lambda v \to ((), v)\,]\!])^{\$} \\ &\qquad\qquad\qquad\qquad\quad \mathit{skip1}\ \text{`}\mathit{Prod}\text{'}\ \mathit{pNth}\ (i - 1)\end{aligned}$$

If i is 0, we simply use *pHead* to update the head element of the source with the view. Otherwise, we do the same arrangements on the view and the source as we did for *pHead*, but then keep the head element unchanged and replace the $(i - 1)$th element of the tail of the source by the view.

```
*Basic> put (pNth 3) [1..10] 100
Just [1,2,3,100,5,6,7,8,9,10]
```

As we know, any putback function in BiGUL is equipped with a *get* function. For *pNth*, we can test its *get* behavior as follows; its corresponding *get* function is actually the familiar index function (!!).

```
*Basic> get (pNth 3) [1..10]
Just 4
```

Both *pHead* and *pNth* contain the programming pattern in which both the source and view are rearranged into a product and then further updates are performed on corresponding components. This is a ubiquitous pattern in BiGUL, for which we provide a more compact syntax:

$$^{\$}(\mathit{update}\ {}^{\mathbb{P}}[\![\, \mathit{sourcePattern} \,]\!]\ {}^{\mathbb{P}}[\![\, \mathit{viewPattern} \,]\!]\ {}^{\mathbb{D}}[\![\, \mathit{updates} \,]\!])$$

The source and view are respectively decomposed using *sourcePattern* and *viewPattern* inside the pattern quasi-quotes ${}^{\mathbb{P}}[\![\ldots]\!]$ (written as `[p| ... |]` in plain text Haskell), and corresponding elements are updated using the programs provided in the declaration quasi-quote ${}^{\mathbb{D}}[\![\ldots]\!]$ (`[d| ... |]` in plain text Haskell). For example, we may describe (*skip1* '*Prod*' *Replace*) '*Prod*' *Replace* by

$$\begin{aligned}
&\mathit{testUpdate} :: (\mathit{Show}\ a, \mathit{Show}\ b, \mathit{Show}\ c) \Rightarrow \mathit{BiGUL}\ ((a,b),c)\ (((),b),c)\\
&\mathit{testUpdate} = {}^{\$}(\mathit{update}\ {}^{\mathbb{P}}[\![\, ((x,y),z) \,]\!]\\
&\qquad\qquad\qquad\qquad {}^{\mathbb{P}}[\![\, ((x,y),z) \,]\!]\\
&\qquad\qquad\qquad\qquad {}^{\mathbb{D}}[\![\, x = \mathit{skip1}; y = \mathit{Replace}; z = \mathit{Replace} \,]\!])
\end{aligned}$$

In this concrete example, the three elements of the tuple (in both the source and view) are bound to the variables x, y, and z, and they are sent to the three combinators as arguments in the ${}^{\mathbb{D}}[\![\ldots]\!]$ part. Note that since *skip1* does nothing on its source but checks if its view is (), we can just match that source element with a wildcard '_' in the source pattern and avoid writing *skip1* in ${}^{\mathbb{D}}[\![\ldots]\!]$.

$$\begin{aligned}
&\mathit{testUpdate}' :: (\mathit{Show}\ a, \mathit{Show}\ b, \mathit{Show}\ c) \Rightarrow \mathit{BiGUL}\ ((a,b),c)\ (((),b),c)\\
&\mathit{testUpdate}' = {}^{\$}(\mathit{update}\ {}^{\mathbb{P}}[\![\, ((\_\,,y),z) \,]\!]\\
&\qquad\qquad\qquad\qquad {}^{\mathbb{P}}[\![\, (((),y),z) \,]\!]\\
&\qquad\qquad\qquad\qquad {}^{\mathbb{D}}[\![\, y = \mathit{Replace}; z = \mathit{Replace} \,]\!])
\end{aligned}$$

3.5 Case

The *Case* combinator is for case analysis, and the general structure is as follows:

$$\begin{aligned}
&\mathit{Case}\ [{}^{\$}(\mathit{normal}\ [\![\, \mathit{mainCond} :: s \to v \to \mathit{Bool} \,]\!]\ [\![\, \mathit{exitCond} :: s \to \mathit{Bool} \,]\!])\\
&\qquad \Longrightarrow (\mathit{bx} :: \mathit{BiGUL}\ s\ v)\\
&\quad ,\ \ldots\\
&\quad ,{}^{\$}(\mathit{adaptive}\ [\![\, \mathit{mainCond} :: s \to v \to \mathit{Bool} \,]\!])\\
&\qquad \Longrightarrow (f :: s \to v \to s)\\
&\quad ,\ \ldots\\
&\quad]\\
&:: \mathit{BiGUL}\ s\ v
\end{aligned}$$

It contains a sequence of cases, each of which is either *normal* or *adaptive*. We try the conditions of these cases in order and decide which branch we go into.

– For a normal case, $(*normal* ...) takes two predicates, which we call the *main condition* and the *exit condition*. The predicate for the main condition is very general, and we can use any function of type $(s \to v \to \mathit{Bool})$ to examine the source and view. The predicate for the exit condition checks the source only. If the main and the exit conditions are satisfied, then the BiGUL program after the arrow '$\Longrightarrow$' (written '`==>`' in plain text Haskell and defined in the module `Generics.BiGUL.Lib`) is executed. The exit conditions in different branches are expected to be disjoint for efficient execution of the forward transformation.
– For an adaptive case, if the main condition is satisfied, a function of type $(s \to v \to s)$ is used to produce an adapted source from the current source and view before the whole *Case* is rerun, with the expectation that one of the normal cases will be applicable this time. Note that if adaptation does not lead to a normal case, an error will be reported at runtime. This is to ensure that BiGUL does not stuck in adaptation and fail to terminate.

As a simple example, consider using the view to replace each element in the source list. To do so, we use *Case* to describe a case analysis.

```
replaceAll :: (Eq s, Show s) ⇒ BiGUL [s] s
replaceAll =
  Case [$(normal ⟦ λs v → length s == 1 ⟧ ⟦ λs → length s == 1 ⟧)
          ⟹ $(rearrS ⟦ λ[x] → x ⟧) Replace
       , $(normal ⟦ λs v → length s > 1 ⟧ ⟦ λs → length s > 1 ⟧)
          ⟹ $(rearrS ⟦ λ(x : xs) → (x, xs) ⟧)$
                $(rearrV ⟦ λv → (v, v) ⟧)$
                  Replace ‘Prod’ replaceAll
       , $(adaptive ⟦ λs v → length s == 0 ⟧)
          ⟹ λs v → [⊥]
       ]
```

It consists of two normal cases and one adaptive case. The first normal case says that if the source is of length 1 (containing a single element), we rearrange the source list by extracting the single element, and replace this element with the view. The second normal case says that if the source has more than 1 element, we rearrange the source list to a pair of its head element and its tail, rearrange the view by duplicating it to a pair, and use one copy of the view to replace the head element, and the other copy to recursively replace each element in the tail of the source. The last adaptive case says that if the source is empty, we adapt the source to a singleton list with the *don't-care* element $\bot$ ('`undefined`' in plain text Haskell), and rerun the whole *Case* executing the first normal case.

```
*Basic> put replaceAll [] 100
Just [100]
*Basic> put replaceAll [1..10] 100
Just [100,100,100,100,100,100,100,100,100,100]
```

Note that in the first running example, the source [] is first adapted to $[\bot]$, and the *don't care* element $\bot$ is replaced by 100 at the rerun of the whole *Case*.

As another interesting example, we define *emb*, which can safely embed any pair of well-behaved *get* and *put* into BiGUL. It is defined as follows:

```
emb :: Eq v ⇒ (s → v) → (s → v → s) → BiGUL s v
emb g p =
   Case [$(normal ⟦λs v → g s == v⟧ ⟦λs → True⟧)
           ⟹ Skip g
         ,$(adaptive ⟦λ_ _ → otherwise⟧)
           ⟹ p
         ]
```

where, given a pair (g, p) of well-behaved *get* and *put* functions, if the view is the same as that produced by applying g to the source, we make no change on the source with *Skip g* (hinting that the view can be produced using g), otherwise we adapt the source using p to reflect the change on the view to the source. Note that if p and g form a well-behaved bidirectional transformation, in the rerun of the whole *Case* after the adaptation, the first normal case will always be applicable. To see a use of *emb*, we may define the following putback function to update a pair with its sum.

```
pSum2 :: BiGUL (Int, Int) Int
pSum2 = emb g p
  where g (x, y)   = x + y
        p (x, y) v = (v − y, y)
```

While we allow a general function to describe the main condition or the exit condition, it is usually more concise to use patterns to describe these conditions. For instance, we may replace the condition $[\![\, \lambda s \rightarrow \mathit{length}\ s == 1 \,]\!]$ by

$$[\![\, \lambda[x] \rightarrow \mathit{True} \,]\!]$$

Here, the meaning of a boolean-valued pattern-matching lambda-expression is redefined as a total function which computes to *False* when an input does not match the pattern; this meaning is different from that of a general pattern-matching lambda-expression, which fails to compute (and throws an exception) when the pattern is not matched. For example, in general the lambda-expression $\lambda[x] \rightarrow \mathit{True}$ will fail to compute if the first input is not a singleton list; when used in branch construction, however, the lambda-expression will compute to *False* upon encountering an empty list. A unary condition like $[\![\, \lambda[x] \rightarrow \mathit{True} \,]\!]$ where only the pattern part matters can be abbreviated to

$${}^{\mathbb{P}}[\![\, [x] \,]\!]$$

to further reduce syntactic noise. Finally, to also allow this kind of abbreviation in main conditions, BiGUL provides a special form for the *normal* case where the main condition is specified as the conjunction of two unary predicates on the source and view respectively:

$$\begin{aligned} &{}^{\$}(\mathit{normalSV}\ [\![\ \mathit{sourceCond} :: s \to \mathit{Bool}\,]\!]\\ &\qquad\qquad\quad [\![\ \mathit{viewCond} \quad :: v \to \mathit{Bool}\,]\!]\\ &\qquad\qquad\quad [\![\ \mathit{exitCond} \quad\ :: s \to \mathit{Bool}\,]\!])\\ &\quad \Longrightarrow (\mathit{bx} :: \mathit{BiGUL}\ s\ v) \end{aligned}$$

and a special form for the *adaptive* case where the main condition is specified as the conjunction of two unary predicates on the source and view respectively:

$$\begin{aligned} &{}^{\$}(\mathit{adaptiveSV}\ [\![\ \mathit{sourceCond} :: s \to \mathit{Bool}\,]\!]\\ &\qquad\qquad\quad\ [\![\ \mathit{viewCond} \quad :: v \to \mathit{Bool}\,]\!])\\ &\quad \Longrightarrow (f :: s \to v \to s) \end{aligned}$$

3.6 View Dependency

Sometimes, a view may contain derived values that are computed from other parts of the view, and the view should be consistently changed. For instance, for the view $(x, \mathit{even}\ (x))$, the second component is an indicator showing whether or not the first component is an even number. To capture this, BiGUL provides

$$\mathit{Dep} :: \mathit{Eq}\ v' \Rightarrow (v \to v') \to \mathit{BiGUL}\ a\ v \to \mathit{BiGUL}\ a\ (v, v')$$

to describe this intention. We may, for example, define

$$\begin{aligned} &\mathit{replaceAll2}\ ::\ \mathit{BiGUL}\ [\mathit{Int}]\ (\mathit{Int}, \mathit{Bool})\\ &\mathit{replaceAll2} = \mathit{Dep}\ \mathit{even}\ \mathit{replaceAll} \end{aligned}$$

to replace all elements of the source by the first component of the view, while checking whether the second component is consistent with the first component.

```
*Basic> put replaceAll2 [1..10] (100,True)
Just [100,100,100,100,100,100,100,100,100,100]
*Basic> put replaceAll2 [1..10] (100,False)
Nothing
*Basic> putTrace replaceAll2 [1..10] (100,False)
second view component not determined by the first
```

As seen in the last running of *put*, it reports an error because the view $(100, \mathit{False})$ is inconsistent: 100 is an even number, so the second component should be *True*.

3.7 Composition

BiGUL programs can be composed sequentially:

$$\mathit{Compose} :: \mathit{BiGUL}\ a\ u \to \mathit{BiGUL}\ u\ b \to \mathit{BiGUL}\ a\ b$$

This combinator is straightforward in the *get* direction: *get* ($Compose\ l\ r$) (where $l :: BiGUL\ a\ u$ and $r :: BiGUL\ u\ b$) simply applies *get* l to its input of type a to compute an intermediate value of type u, which is then processed by *get* r to produce the final result of type b. Its *put* direction is more complex: *put* ($Compose\ l\ r$) starts with a source $s :: a$ and a view $v :: b$, and the aim is to produce an updated source of type a. The only way to proceed is to use *put* r to put v into some intermediate source m of type u, and to produce this m we are forced to use *get* l on s. We can then update m with v to m' using *put* r, and update a with m' using *put* l. In general, programs involving *Compose* are significantly harder to think about since we have to think in both *put* and *get* directions to figure out precisely what is going on.

As a simple example, consider that we wish to use the view to update the head element of the head element of a list of lists. We can define such a putback function as the following *pHead2* by composing *pHead* with *pHead*.

$$
\begin{aligned}
&pHead2 :: Show\ a \Rightarrow BiGUL\ [[a]]\ a\\
&pHead2 = pHead\ `Compose`\ pHead
\end{aligned}
$$

The following is an example to demonstrate this:

```
*Basic> put pHead2 [[1,2],[3,4,5],[]] 100
Just [[100,2],[3,4,5],[]]
```

4 Bidirectional Programming on Lists

To give some more involved examples, in this section we demonstrate that many list functions can be bidirectionalized using BiGUL. The putback behaviors of these functions are in fact non-trivial, and the reader might want to skip to later sections in which more examples are developed, starting from Sect. 6.

To show the correspondence with the original list functions, we prefix the original forward function names with *lens*. Note that in our context, the original forward functions can be automatically derived from the new putback transformations by calling *get*.

We shall focus on bidirectionalizing *foldr*, an important higher-order function on lists:

$$
\begin{aligned}
&foldr :: (a \to b \to b) \to b \to [a] \to b\\
&foldr\ f\ e\ [\,] \qquad\ = e\\
&foldr\ f\ e\ (x : xs) = f\ x\ (foldr\ f\ e\ xs)
\end{aligned}
$$

Many interesting functions can be defined in terms of *foldr*:

$$
\begin{aligned}
&sum \quad\ = foldr\ (+)\ 0\\
&map\ f = foldr\ (\lambda a\ r \to f\ a : r)\ [\,]
\end{aligned}
$$

where *sum* sums up all the elements in a list, and *map f* applies *f* to every element in a list.

We start by developing a putback function for *foldr* in BiGUL:

$$\begin{aligned}&\mathit{lensFoldr} :: (\mathit{Show}\ a, \mathit{Show}\ v)\\&\qquad \Rightarrow \mathit{BiGUL}\ (a, v)\ v \to (v \to \mathit{Bool}) \to \mathit{BiGUL}\ ([a], v)\ v\end{aligned}$$

where we hope to define a putback program of type $\mathit{BiGUL}\ ([a], v)\ v$ that is to use the view to update the source, a list together with a value, by recursively applying a simpler putback function of type $\mathit{BiGUL}\ (a, v)\ v$ (until a condition is satisfied or all the list elements have been visited). The program is somewhat tricky, and is probably not easy to understand since *Compose* is involved.

$$\begin{aligned}&\mathit{lensFoldr}\ \mathit{bx}\ \mathit{pv} =\\&\quad \mathit{Case}\ [{}^{\$}(\mathit{adaptive}\ [\![\, \lambda(x, y)\ v \to \mathit{pv}\ v \wedge \mathit{length}\ x \not\equiv 0 \,]\!])\\&\qquad\quad \Longrightarrow \lambda(x, y)\ v \to ([\,], y)\\&\qquad\ \ , {}^{\$}(\mathit{normal}\ [\![\, \lambda(\mathit{xs}, \_)\ v \to \mathit{null}\ \mathit{xs} \,]\!]\ [\![\, \lambda(\mathit{xs}, \_) \to \mathit{null}\ \mathit{xs} \,]\!])\\&\qquad\quad \Longrightarrow {}^{\$}(\mathit{rearrV}\ [\![\, \lambda v \to ((), v) \,]\!])^{\$}\\&\qquad\qquad\qquad {}^{\$}(\mathit{update}\ {}^{\mathbb{P}}[\![\, (\_, v) \,]\!]\ {}^{\mathbb{P}}[\![\, ((), v) \,]\!]\ {}^{\mathbb{D}}[\![\, v = \mathit{Replace} \,]\!])\\&\qquad\ \ , {}^{\$}(\mathit{normalSV}\ {}^{\mathbb{P}}[\![\, \_ \,]\!]\ {}^{\mathbb{P}}[\![\, \_ \,]\!]\ [\![\, \lambda(\mathit{xs}, \_) \to \mathit{not}\ (\mathit{null}\ \mathit{xs}) \,]\!])\\&\qquad\quad \Longrightarrow {}^{\$}(\mathit{rearrS}\ [\![\, \lambda((x : \mathit{xs}), e) \to (x, (\mathit{xs}, e)) \,]\!])^{\$}\\&\qquad\qquad\qquad (\mathit{Replace}\ `\mathit{Prod}`\ \mathit{lensFoldr}\ \mathit{bx}\ \mathit{pv})\ `\mathit{Compose}`\ \mathit{bx}\\&\qquad\ \ ]\end{aligned}$$

The *lensFoldr* program accepts a putback function *bx* and a view condition *pv*, and performs a case analysis to put the view v to the source (xs, e). If the view v satisfies *pv* but the list *xs* in the source is not empty, then it adapts the list to be empty. If the list *xs* in the source is empty, we do nothing but use the view to replace the second component of the source. Otherwise, we rearrange the source from the form of $(x : \mathit{xs}, e)$ to that of $(x, (\mathit{xs}, e))$, and apply *lensFoldr* recursively with a composition with *bx*. One may understand the composition through the following picture (where $r = \mathit{Replace}\ `\mathit{Prod}`\ \mathit{lensFoldr}\ \mathit{bx}\ \mathit{pv}$).

$$(x, (\mathit{xs}, e)) \overset{r}{\leftrightarrow} (x, e') \overset{\mathit{bx}}{\leftrightarrow} v$$

With *lensFoldr*, we can redefine many list functions from the putback point of view. As the first example, consider *mapAppend*:

$$\mathit{mapAppend}\ f\ (\mathit{xs}, \mathit{ys}) = \mathit{map}\ f\ \mathit{xs} +\!\!+\ \mathit{ys}$$

We can define its putback function as follows.

$$\begin{aligned}&\mathit{lensMapAppend} :: (\mathit{Show}\ a, \mathit{Show}\ b) \Rightarrow \mathit{BiGUL}\ a\ b \to \mathit{BiGUL}\ ([a], [b])\ [b]\\&\mathit{lensMapAppend}\ \mathit{pf} = \mathit{lensFoldr}\ \mathit{bx}\ \mathit{null}\\&\quad \textbf{where}\ \mathit{bx} = {}^{\$}(\mathit{rearrV}\ [\![\, \lambda(v : \mathit{vs}) \to (v, \mathit{vs}) \,]\!])^{\$}\\&\qquad\qquad\qquad\quad \mathit{pf}\ `\mathit{Prod}`\ \mathit{Replace}\end{aligned}$$

Here *bx* has the type of $\mathit{BiGUL}\ (a, [b])\ [b]$ and is defined on *pf* that has the type of $\mathit{BiGUL}\ a\ b$.

```
*List> put (lensMapAppend dec1) ([0..10],[]) [100..110]
Just ([99,100,101,102,103,104,105,106,107,108,109],[])
*List> get (lensMapAppend dec1) ([1..10],[])
Just [2,3,4,5,6,7,8,9,10,11]
```

Note that, for testing, we embed into our framework the bijective functions for increasing and decreasing a number by 1.

$$\begin{aligned}&\mathit{dec1} :: (\mathit{Eq}\ a, \mathit{Num}\ a) \Rightarrow \mathit{BiGUL}\ a\ a\\&\mathit{dec1} = \mathit{emb}\ g\ p\\&\quad \textbf{where}\ g\ s \quad = s + 1\\&\qquad\qquad p\ s\ v = v - 1\end{aligned}$$

For a second example, consider the function *sum* (*xs*, *e*), which is to sum up all elements of the list *xs* starting from the seed *e*. If the sum is changed, there are many ways to reflect this change to the input (*xs*, *e*). The following describes one way in BiGUL:

$$\begin{aligned}&\mathit{lensSum} :: \mathit{BiGUL}\ ([\mathit{Int}], \mathit{Int})\ \mathit{Int}\\&\mathit{lensSum} = \mathit{lensFoldr}\ \mathit{pSum2}\ (\mathit{const}\ \mathit{False})\end{aligned}$$

which will reflect the change difference on the view to the head element of *xs* if *xs* is not empty, or to the seed *e* otherwise. We may choose other ways, say to reflect the change difference on the view only to the seed by defining

$$\begin{aligned}&\mathit{lensSum'} :: \mathit{BiGUL}\ ([\mathit{Int}], \mathit{Int})\ \mathit{Int}\\&\mathit{lensSum'} = \mathit{lensFoldr}\ ({}^{\$}(\mathit{rearrS}\ [\![\, \lambda(x,y) \rightarrow (y,x)\,]\!])\ \mathit{pSum2})\ (\mathit{const}\ \mathit{False})\end{aligned}$$

Note that although $\mathit{get}\ \mathit{lensSum}\ ([1,2,3],0) = \mathit{get}\ \mathit{lensSum'}\ ([1,2,3],0) = \mathit{Just}\ 6$, their putback behaviors are different:

$$\begin{aligned}&\mathit{put}\ \mathit{lensSum}\ \ ([1,2,3],0)\ 16 = \mathit{Just}\ ([11,2,3],0)\\&\mathit{put}\ \mathit{lensSum'}\ ([1,2,3],0)\ 16 = \mathit{Just}\ ([1,2,3],10)\end{aligned}$$

It is worth noting that our definition of *lensFoldr* is just one putback function for *foldr*, and there are many others. This reflects the fact that one *foldr* can have many *put*s, each describing one updating strategy.

5 BiGUL's Bidirectionality

We have been writing *put* programs, usually having a corresponding *get* in mind but not explicitly describing it, and yet BiGUL is capable of finding the right *get* behaviour as if it could read our mind. How? We will see that, when writing a BiGUL program, we are always simultaneously describing both a *put* function and a *get* function, which are guaranteed to be a well-behaved pair. And the "mind-reading" ability is far from magic: It is the consequence of the fact that well-behavedness directly implies that *get* is uniquely determined by *put*, which is the main motivation for taking a putback-based approach. In this section, we will first review the theory, this time explicitly taking *partiality* into account, and then we will dive into BiGUL's internals to get a taste of putback-based design.

This is a fairly long section, but it is not a prerequisite for subsequent sections; readers who wish to see more examples first or are more interested in practical BiGUL applications can safely skip this section and proceed to Sect. 6.

5.1 Lenses, Well-Behavedness, and the Fundamental Theorem

Formally, we call a well-behaved pair of *put* and *get* a *lens*:

Definition 2 (lens). *A* lens *between a source type s and a view type v consists of two functions:*

$$\begin{aligned} put &:: s \to v \to Maybe\ s \\ get &:: s \qquad\ \to Maybe\ v \end{aligned}$$

satisfying two well-behavedness laws:

$$\begin{aligned} put\ s\ v = Just\ s' \quad &\Rightarrow \quad get\ s' = Just\ v \qquad \textsc{PutGet} \\ get\ s = Just\ v \quad &\Rightarrow \quad put\ s\ v = Just\ s \qquad \textsc{GetPut} \end{aligned}$$

In the original formulation [8], a lens refers to just a pair of functions having the right types, and one needs to explicitly say "well-behaved lens" to mean a well-behaved pair; we will, however, discuss well-behaved lenses only, so we build well-behavedness into our definition of lenses by default. Note that this definition models partial transformations explicitly as *Maybe*-valued functions: *put* and *get* are *total* functions that can nevertheless produce *Nothing* to indicate failure. From now on, this definition replaces the one in Sect. 1, where only total lenses were discussed. Also note that these well-behavedness laws are actually easy to satisfy vacuously, by making the transformations produce *Nothing* all (or most of) the time. One important task of the BiGUL programmer is thus to meet certain side conditions for guaranteeing the totality of their BiGUL programs. These side conditions will be introduced below along with the relevant BiGUL constructs.

From this revised definition of well-behavedness, we can immediately prove a reformulation of Lemma 1:

Theorem 2 (uniqueness of *get*). *Given two lenses whose put components are equal, their get components are also equal.*

Proof. Let l and r be two lenses; denote their *put*/*get* components as $put\ l/get\ l$ and $put\ r/get\ r$ respectively, and assume that $put\ l = put\ r$. Then for any

s and v,

$$\begin{aligned}
& \mathit{get}\ l\ s = \mathit{Just}\ v \\
\Leftrightarrow\ & \{\ \text{well-behavedness of } l\ \} \\
& \mathit{put}\ l\ s\ v = \mathit{Just}\ s \\
\Leftrightarrow\ & \{\ \mathit{put}\ l = \mathit{put}\ r\ \} \\
& \mathit{put}\ r\ s\ v = \mathit{Just}\ s \\
\Leftrightarrow\ & \{\ \text{well-behavedness of } r\ \} \\
& \mathit{get}\ r\ s = \mathit{Just}\ v
\end{aligned}$$

(This also entails that $\mathit{get}\ l\ s = \mathit{Nothing}$ if and only if $\mathit{get}\ r\ s = \mathit{Nothing}$.) □

This might be called the "fundamental theorem" of putback-based bidirectional programming, as the theorem guarantees that the BiGUL programmer is in full control of the bidirectional behaviour—programming the *put* behavior is sufficient to determine the *get* behaviour. Also, to the language designer, the theorem gives a kind of reassurance that, once the *put* behaviour of a construct is determined, there is no need to worry about which *get* behaviour should be adopted—there is at most one possibility. This is in contrast to *get*-based design, in which there are usually more than one viable *put* semantics that can be assigned to a *get*-based construct, and the designer needs to justify the choice or provide several versions.

For the rest of this section, we will look at several constructs of BiGUL in detail to get a taste of putback-based design. Each BiGUL construct is conceived, at the design stage, as a lens (like *Skip* and *Replace*) or a lens *combinator* (like *Case*), which constructs a more complex lens from simpler ones. The *put* and *get* components of these lenses usually have to be developed together, but for each lens we will employ a more "*put*-oriented" design process: We start from an intended *put* behaviour, and then add restrictions so that we can find a corresponding *get*. This does not guarantee that the lenses we arrive at will have a "strong *put* flavour"—that is, some of the lenses will be as (or even more) suitable for *get*-based programming as for putback-based programming. But we will also see that some other lenses are more naturally understood in terms of their *put* behaviour.

5.2 Replacement

The simplest lens is probably *Replace*, which replaces the entire source with the view:

$$\mathit{put}\ \mathit{Replace}\ s\ v = \mathit{Just}\ v$$

Is there a *get* semantics that can be paired with this *put*? Yes, quite obviously—in fact, PUTGET directly gives us the definition of *get Replace*:

$$\mathit{get}\ \mathit{Replace}\ v = \mathit{Just}\ v$$

We still need to verify GETPUT, which can be easily checked to be true.

5.3 Skipping

Coming up next is *Skip*, whose natural behaviour is

$$put\ Skip\ s\ v = Just\ s$$

Considering PutGet, though, we immediately see that this behaviour is too liberal: If the view is simply thrown away, how can *get Skip* possibly recover it? One way out is to require that the view is trivial enough such that it can be thrown away and still be recovered, by setting the view type of *Skip* to the unit type (). Then it is easy for *get Skip* to recover the view, for which there is only one choice:

$$get\ Skip\ s = Just\ ()$$

This is the approach adopted prior to BiGUL 1.0.

More generally, we can establish well-behavedness as long as *get Skip* has only one view choice for each source, regardless of what the view type is. The existence of this "unique choice" is witnessed by a function $f :: s \rightarrow v$, which we add as an additional argument to *Skip*. The *get* direction is then

$$get\ (Skip\ f)\ s = Just\ (f\ s)$$

From the *put* direction, we may think of this function f as specifying a consistency relation, saying that the view information is completely included in the source (since you can compute the view from the source) and can be safely discarded. *Skip f* can be used if and only if the source and view are consistent in that sense, and this is the side condition about *Skip* that the BiGUL programmers need to be aware of if they want their programs using *Skip* to be total. We thus arrive at:

$$put\ (Skip\ f)\ s\ v = \mathbf{if}\ v == f\ s\ \mathbf{then}\ return\ s\ \mathbf{else}\ Nothing$$

This pair of *put* and *get* can be verified to be well-behaved. *Skip f*, which features in BiGUL 1.0, is one lens which turns out to be more easily understood from the *get* direction—it bidirectionalizes any *get* function whose codomain has decidable equality, albeit trivially. We recover the first version of *Skip* as a special case by setting f to *const* ().

5.4 Product

For a simplest example of a lens combinator, we look at *Prod*. Both the source and view types should be pairs; *Prod* accepts two lenses, say l and r, and applies them respectively to the left and right components:

```
put (l `Prod` r) (sl, sr) (vl, vr) = do sl' ← put l  sl  vl
                                        sr' ← put r  sr  vr
                                        return (sl', sr')
```

The *get* direction is unsurprising:

$$\begin{aligned} \mathit{get}\ (l\ \text{`}\mathit{Prod}\text{'}\ r)\ (\mathit{sl}, \mathit{sr}) = \mathbf{do}\ & \mathit{vl}\ \leftarrow \mathit{get}\ l\ \ \mathit{sl} \\ & \mathit{vr} \leftarrow \mathit{get}\ r\ \mathit{sr} \\ & \mathit{return}\ (\mathit{vl}, \mathit{vr}) \end{aligned}$$

Having constructed *put* and *get* from l and r, we also expect that their well-behavedness is a consequence of the well-behavedness of l and r. While this may look obvious, we take this opportunity to show how a well-behavedness proof for a lens combinator can be carried out formally and in detail. To prove PutGet, for example, we should prove that the assumption

$$\mathit{put}\ (l\ \text{`}\mathit{Prod}\text{'}\ r)\ (\mathit{sl}, \mathit{sr})\ (\mathit{vl}, \mathit{vr}) = \mathit{Just}\ (\mathit{sl}', \mathit{sr}') \tag{1}$$

implies the conclusion

$$\mathit{get}\ (l\ \text{`}\mathit{Prod}\text{'}\ r)\ (\mathit{sl}', \mathit{sr}') = \mathit{Just}\ (\mathit{vl}, \mathit{vr}) \tag{2}$$

Both equations say that a somewhat complicated monadic *Maybe*-program computes successfully to some value. It may seem that we need some messy case analysis, but what we know about *Maybe*-programs tells us that such a program computes successfully if and only if every step of the program does, and this helps us to split both (1) and (2) into simpler equations. Formally, we have this lemma:

Lemma 2. *Let* $\mathit{mx} :: \mathit{Maybe}\ a$ *and* $f :: a \rightarrow \mathit{Maybe}\ b$*. Then, for all* $y :: b$*,*

$$\mathit{mx} \gg\!= f = \mathit{Just}\ y$$

if and only if

$$\mathit{mx} = \mathit{Just}\ x \quad \text{and} \quad f\ x = \mathit{Just}\ y \qquad \text{for some } x :: a$$

Proof. Case analysis on mx. □

This lemma can be nicely applied to *Maybe*-programs written in the **do**-notation, transforming such programs into *predicates* saying that a program computes to some given value. To do it more formally: Define a translation $\mathcal{S}$ from **do**-blocks of type $\mathit{Maybe}\ a$ to predicates on a by

$$\begin{aligned} \mathcal{S}\ (\mathbf{do}\ \{x \leftarrow \mathit{mx}; B\})\ y &= (\exists x.\ \mathit{mx} = \mathit{Just}\ x \wedge \mathcal{S}\ (\mathbf{do}\ B)\ y) \\ \mathcal{S}\ (\mathbf{do}\ \{\mathit{my}\})\ y &= (\mathit{my} = \mathit{Just}\ y) \end{aligned}$$

Then we can extend Lemma 2 to the following:

Lemma 3. *The proposition*

$$\mathcal{S}\ (\mathbf{do}\ B)\ y$$

is true if and only if

$$\mathbf{do}\ B = \mathit{Just}\ y$$

Proof. By induction on the list structure of B, using Lemma 2 repeatedly. □

For example, applying $\mathcal{S}$ to $put\ (l\ `Prod`\ r)\ (sl, sr)\ (vl, vr)$ yields

$$\begin{aligned}\lambda(sl'', sr'').\ &\exists sl'.\ put\ l\ sl\ vl = Just\ sl' \wedge \\ &\exists sr'.\ put\ r\ sr\ vr = Just\ sr' \wedge \\ &return\ (sl', sr') = Just\ (sl'', sr'')\end{aligned}$$

where the last equation is equivalent to $sl' = sl'' \wedge sr' = sr''$ (since $return = Just$ for the *Maybemonad*). Applying Lemma 3 and doing some simplification, (1) is equivalent to

$$put\ l\ sl\ vl = Just\ sl' \quad \wedge \quad put\ r\ sr\ vr = Just\ sr'$$

Similarly, (2) can be shown to be equivalent to

$$get\ l\ sl' = Just\ vl \quad \wedge \quad get\ r\ sr' = Just\ vr$$

The entailment is then just PutGet for l and r.

5.5 Case Analysis

This is a representative combinator in BiGUL, and arguably the most complex one. For simplicity, let us consider a two-branch variant of *Case*. A branch is a condition and a body; since in *put* we manipulate both a source and a view, the conditions in general can be binary predicates on both the source and view. We thus define the type of branches as:

$$\textbf{type}\ CaseBranch\ s\ v = (s \to v \to Bool, BiGUL\ s\ v)$$

and consider the following variant of *Case*:

$$Case :: CaseBranch\ s\ v \to CaseBranch\ s\ v \to BiGUL\ s\ v$$

The straightforward behaviour is

$$\begin{aligned}put\ (Case\ (pl, l)\ (pr, r))\ s\ v = \ &\textbf{if}\ \ \ \ \ \ \ pl\ \ s\ v\ \textbf{then}\ put\ l\ \ s\ v \\ &\textbf{else if}\ pr\ s\ v\ \textbf{then}\ put\ r\ s\ v \\ &\textbf{else}\ Nothing\end{aligned}$$

That is, depending on which condition is satisfied (with *pl* having higher priority), we execute either *put l* or *put r*, or fail the computation if neither of the conditions is satisfied. Now, again, we ask the question: Can we find a *get* behaviour to pair with this *put*?

Ruling out branch switching for PutGet. An important working assumption here is that we want lens combinators to be *compositional*: When we looked at *Prod*, for example, we defined its *put* and *get* in terms of those of the smaller lenses, and derived the overall well-behavedness from that of the smaller lenses. For *Case*, this implies that, when establishing well-behavedness, we want a *get* following a *put* (or a *put* following a *get*) to use the same branch taken by the *put* (or the *get*), so we can make use of PutGet (or GetPut) of the branch. The current *put* behaviour of *Case* does not leave any clue in the updated source about which branch is used to produce it, though, so it is impossible for *get* to always choose the correct branch.

One solution, which does not require changing the syntax of *Case*, is to check that the ranges of the branches are *disjoint*. In general, for a lens, the range of a *put* can be shown to coincide with the domain of the corresponding *get*. So the *get* behaviour of *Case* can simply try to execute both branches on the input source, and there will be at most one branch that computes successfully. We can put (expensive) disjointness checks into *put* such that if *put* succeeds, the subsequent *get* will have at most one branch to choose:

```
put (Case (pl, l) (pr, r)) s v =
  if      pl s v then do s' ← put l s v
                         maybe (return s') (const Nothing) (get r s')
  else if pr s v then do s' ← put r s v
                         maybe (return s') (const Nothing) (get l s')
  else Nothing
```

The *maybe* function is from Haskell's prelude and has type $b \to (a \to b) \to \mathit{Maybe}\ a \to b$; depending on whether the third, *Maybe*-typed, argument is *Nothing* or a *Just*-value, the result is either the first argument or the second argument applied to the value wrapped inside *Just*. In the first branch of the code above, if *put l s v* successfully produces an updated source s', we will ensure that *get r* s' does not succeed: If *get r* s' is *Nothing* as we want, we will *return* s'; otherwise we emit *Nothing*.

If *get* favours the first branch, meaning that it declares success as soon as the first branch succeeds (without requiring that the second branch fails),

```
get (Case (pl, l) (pr, r)) s = maybe (get r s) return (get l s)
```

then we can also omit the check in *put*'s first branch:

```
put (Case (pl, l) (pr, r)) s v =
  if      pl s v then put l s v
  else if pr s v then do s' ← put r s v
                         maybe (return s') (const Nothing) (get l s')
  else Nothing
```

Ruling out branch switching for GetPut. The GETPUT direction, on the other hand, still does not avoid branch switching—the outcome of *get* does not say anything about which of *pl* and *pr* will be satisfied in the subsequent *put*. So we add some checks to *get* such that *get*'s success will tell us which branch will be chosen by *put*:

```
get (Case (pl, l) (pr, r)) s = maybe (do v ← getBranch (pr, r) s
                                         if pl s v then Nothing
                                                   else return v)
                                     return
                                     (getBranch (pl, l) s)
getBranch (p, b) s = do v ← get b s
                        if p s v then return v
                                 else Nothing
```

The definition of *put* should also be revised to use *getBranch* for the disjointness check. This fixes GETPUT, but breaks PUTGET! Since *put* does not guarantee that the *updated* (not the original) source and the view satisfy the condition of the branch executed, even though *get* will be able to choose the correct branch, the subsequent, newly added check is not guaranteed to succeed. We thus also need to add similar checks to *put*:

```
put (Case (pl, l) (pr, r)) s v =
  if      pl s v then do s' ← put l s v
                         if pl s' v then return s'
                                    else Nothing
  else if pr s v then do s' ← put r s v
                         if pr s' v then maybe (return s')
                                               (const Nothing)
                                               (getBranch (pl, l) s')
                                    else Nothing
  else Nothing
```

Now this pair of *put* and *get* can be verified to be well-behaved.

Improving the Efficiency of *get*. The efficiency of the current *get* does not look very good, especially when, in general, more than two branches are allowed, and *get* has to try to execute each branch, possibly with a high cost, until it reaches a successful one; also, inefficient *get* affects the efficiency of *put*, since this calls *get* to check range disjointness. An idea is to ask the programmer to make a rough "prediction" of the range of each branch: We enrich *CaseBranch* with a third component, which is a source predicate:

```
type CaseBranch s v = (s → v → Bool, BiGUL s v, s → Bool)
```

This new predicate is supposed to be satisfied by the updated source; we again add checks to *put* to ensure this:

```
put (Case (pl, l, ql) (pr, r, qr)) s v =
  if      pl s v then do s' ← put l s v
                         if pl s' v ∧ ql s' then return s'
                                            else Nothing
  else if pr s v then do s' ← put r s v
                         if pr s' v ∧ qr s'
                           then maybe (return s')
                                      (const Nothing)
                                      (getBranch (pl, l, ql) s')
                           else Nothing
  else Nothing
```

Let us call *pl* and *pr* the *main conditions*, and *ql* and *qr* the *exit conditions*. The exit condition, in general, over-approximates the range of a branch. Well-behavedness tells us that the range of *put* is exactly the domain of the corresponding *get*. Thus, in the *get* direction, every source in the domain of a branch satisfies the exit condition. Contrapositively, if a source does not satisfy the exit condition, then *get* for that branch will necessarily fail, and we do not need to try to execute the branch at all. This leads to the following revised definition of *getBranch*:

```
getBranch (pl, l, ql) s = if ql s then do v ← get l s
                                          if pl s v then return v
                                                    else Nothing
                                  else Nothing
```

If we do not care about efficiency, we can simply use *const True* as exit conditions, and the behaviour will be exactly the same as the previous version. But if we supply disjoint exit conditions, then *get* will try at most one branch. Incidentally (but actually no less importantly), making exit conditions explicit also encourages the programmer to think about range disjointness, which is essential to guaranteeing the totality of *Case*.

Adaptation. We have seen that, to make *Case* total, one thing we need to ensure is that the main condition of a branch should be satisfied again after the update. In practice, the main condition is usually closely related to the consistency relation, and we will only be able to deal with sources and views that are already more or less consistent; this is a rather severe restriction. As we have seen in Sect. 3.5, the solution is to introduce a different kind of branch called *adaptive branches*, which can deal with sources and views that are too inconsistent by adapting the source to establish enough consistency such that a normal branch becomes applicable. Again, for simplicity, we consider only a variant of *Case* which has just one adaptive branch at the end:

```
type CaseAdaptiveBranch s v = (s → v → Bool, s → v → s)
Case :: CaseBranch s v → CaseBranch s v →
        CaseAdaptiveBranch s v → BiGUL s v
```

The execution structure of *put* becomes slightly more complicated, as the whole thing has to be run again after adaptation; to ensure termination, we require that the second run does not match an adaptive branch again. This is realized in BiGUL in continuation-passing style:

```
put (Case bl br ba) s v =
  putWithAdaptation bl br ba s v (λsa →
    putWithAdaptation bl br ba sa v (const Nothing))
putWithAdaptation ::
  CaseBranch s v → CaseBranch s v → CaseAdaptiveBranch s v →
  s → v → (s → Maybe s) → Maybe s
putWithAdaptation (pl, l, ql) (pr, r, qr) (pa, f) s v cont =
  if      pl  s v then do s' ← put l  s v
                          if pl s' v ∧ ql s' then return s'
                                             else  Nothing
  else if pr  s v then do s' ← put r  s v
                          if pr s' v ∧ qr s'
                            then maybe (return s')
                                       (const Nothing)
                                       (getBranch (pl, l, ql) s')
                            else  Nothing
  else if pa  s v then cont (f  s v)
  else Nothing
```

Major work is now moved into a separate function *putWithAdaptation*, which takes an extra *cont* argument of type $s \to Maybe\ s$. This extra argument is a continuation that takes over after the body of an adaptive branch is executed, and is invoked with the adapted source. The requirement of not doing adaptation twice is met by setting *putWithAdaptation* itself as a continuation, and this inner *putWithAdaptation* takes the continuation that always fails.

What about *get*? It turns out that *get* can simply ignore the adaptive branch! If you have doubt about this "choice", just invoke the fundamental theorem (Theorem 2): The *put* behaviour is exactly what we want, and we can verify that the pair of *put* and *get* is well-behaved, so we are reassured that our "choice" is "correct", simply because there is no other choice of *get*.

To sum up, we have arrived at a simpler variant of *Case* which nevertheless has all the features of the multi-branch *Case* in BiGUL. We have inserted various dynamic checks into the *put* semantics, and the BiGUL programmer needs to be aware of these constraints to make execution of *Case* succeed: For each normal branch, (i) the main condition should be satisfied after the update, (ii) the main conditions of the branches before this one should not be satisfied after the update, and (iii) the exit condition should be satisfied by the updated source. Also the ranges of all the normal branches should be disjoint; the programmer is encouraged to write disjoint exit conditions, which imply disjointness of the ranges, and improve the efficiency of *get*. Finally, for each adaptive branch, the adapted source and the view should match the main condition of a normal branch.

5.6 Rearrangement

Source and view rearrangements are also among the more complex constructs of BiGUL. Their complexity lies in the strongly and generically typed treatment of pattern matching, though, rather than their bidirectional behavior. (We are referring to "pattern matching" in functional programming, where a pattern matching checks whether a value has a specific shape and decomposes it into components. For example, matching a list with a pattern $x : y : xs$ checks whether the list has two or more elements, and then binds x to the first element, y to the second one, and xs to the rest of the list.) The two kinds of rearrangement are similar, and we will discuss view rearrangement only. We will start by formalizing pattern matching as a bidirectional operation—in fact an isomorphism. Based on pattern matching, evaluation and inverse evaluation of rearranging λ-expressions can be defined, again forming an isomorphism. The semantics of a view rearrangement is then the composition of this latter isomorphism with the lens obtained by interpreting the inner BiGUL program.

Strongly typed pattern matching, bidirectionally. Pattern matching is inherently a bidirectional operation: In one direction, we break something into a collection of its components at the variable positions of a pattern. This collection can be considered as indexed by the variable positions, and acting like an *environment* for expression evaluation. Indeed, conversely, if we have a pattern and a corresponding environment, we can treat the pattern as an expression and evaluate it in the environment. These two directions are inverse to each other, i.e., they form a (partial) isomorphism. For the language designer, it may be slightly tedious to establish such isomorphisms, but for the programmer, pattern matching and evaluation are arguably the most natural way to decompose and rearrange things. Previous bidirectional languages usually provide theoretically simpler combinators for decomposition and rearrangement, but they are hard to use in practice. BiGUL's native support of pattern matching, on the other hand, turns out to be one important contributing factor in its usability.

BiGUL's patterns are strongly typed: The programmer has to declare a target type for a pattern, and the pattern is guaranteed, through typechecking, to make sense for that target type. This can be achieved by defining the datatype of patterns as a generalised algebraic datatype:

```
data Pat a where
  PVar   :: Eq a ⇒                    Pat a
  PConst :: Eq a ⇒ a →                Pat a
  PProd  :: Pat a → Pat b →           Pat (a, b)
  PLeft  :: Pat a →                   Pat (Either a b)
  PRight :: Pat b →                   Pat (Either a b)
  PIn    :: InOut a ⇒ Pat (F a) → Pat a
```

A pattern can be a (nameless) variable, a constant, a product, a *Left* or *Right* injection (for the *Either* type), or a generic constructor, and its target type is given as the index in its type. For example, the pattern *PLeft* (*PConst* ()) has type *Pat* (*Either* () *b*), and can only be used to match those values of type *Either* () *b* (and matching succeeds only for the value *Left* ()). The *InOut* type-class contains the types that are isomorphic to (and therefore interconvertible with) a sum-of-products representation. The isomorphism is witnessed by

$$\mathit{inn} :: \mathit{InOut}\ a \Rightarrow F\ a \rightarrow a \qquad \text{and} \qquad \mathit{out} :: \mathit{InOut}\ a \Rightarrow a \rightarrow F\ a$$

which will be used to define pattern matching and evaluation. For example, $[a]$ is an instance of *InOut*, and $F\ [a]$, an isomorphic sum-of-products representation of $[a]$, is $\mathit{Either}\ ()\ (a, [a])$. The two functions witnessing the isomorphism for lists are defined by

```
inn (Left  ())       = []
inn (Right (x, xs))  = x : xs
out []       = Left  ()
out (x : xs) = Right (x, xs)
```

How do we define pattern matching? As we mentioned above, the result of matching a value against a pattern is an environment indexed by the variable positions of the pattern. For example, matching a list against the cons pattern

$$\mathit{PIn}\ (\mathit{PRight}\ (\mathit{PProd}\ \mathit{PVar}\ \mathit{PVar})) \tag{3}$$

should produce an environment containing its head and tail. Here we want a safe (but not necessarily efficient) representation of the environment type, in the sense that the indices into the environment should be exactly the variable positions of the pattern, and we want that to be enforced statically by typechecking. In other words, this environment type depends on the pattern, and a way to compute this type is to encode it as a second index of the *Pat* datatype:

```
data Pat a env where
  PVar   :: Eq a ⇒                     Pat a (Var a)
  PConst :: Eq a ⇒ a →                 Pat a ()
  PProd  :: Pat a a' → Pat b b' b'' →  Pat (a, b) (a', b')
  PLeft  :: Pat a a' →                 Pat (Either a b) a'
  PRight :: Pat b b' →                 Pat (Either a b) b'
  PIn    :: InOut a ⇒ Pat (F a) b →    Pat a b
```

Notice that an environment type is just a product of *Var* types—for example, the environment type computed for the cons pattern (3) is

$$(\mathit{Var}\ a, \mathit{Var}\ [a]) \tag{4}$$

We will discuss *Var* later, which is simply defined by

```
newtype Var a = Var a
```

Now we can define the (strongly typed) pattern matching operation:

```
deconstruct :: Pat a env → a → Maybe env
deconstruct PVar          x          = return (Var x)
deconstruct (PConst c)    x          = if c == x then return () else Nothing
deconstruct (l ‘PProd’ r) (x, y)     = liftM2 (, ) (deconstruct l x)
                                                    (deconstruct r y)
deconstruct (PLeft p)     (Left x)   = deconstruct p x
deconstruct (PLeft _)     _          = Nothing
deconstruct (PRight p)    (Right x)  = deconstruct p x
deconstruct (PRight _)    _          = Nothing
deconstruct (PIn p)       x          = deconstruct p (out x)
```

and its inverse (which is total):

```
construct :: Pat a env → env → a
construct PVar          (Var x)      = x
construct (PConst c)    _            = c
construct (l ‘PProd’ r) (envl, envr) = (construct l envl, construct r envr)
construct (PLeft p)     env          = Left  (construct p env)
construct (PRight p)    env          = Right (construct p env)
construct (PIn p)       env          = inn (construct p env)
```

Precisely speaking, we have

$$deconstruct\ p\ x = Just\ e \quad \Leftrightarrow \quad construct\ p\ e = x$$

for all $p :: Pat\ a\ env$, $x :: a$, and $e :: env$, establishing a (half-) partial isomorphism between *env* and *a*.

λ-expressions for rearrangement and their evaluation. Now consider view rearrangement, which evaluates a "simple" pattern-matching λ-expression on the view and continues execution with the transformed view. The body of the λ-expression refers to the variables appearing in the pattern. How do we represent such references? We have seen that an environment type is a product, i.e., a binary tree; to refer to a component in an environment, we can use a *path* that goes from the root to a sub-tree. In BiGUL, these paths are called *directions*:

```
data Direction env a where
  DVar   ::                    Direction (Var a) a
  DLeft  :: Direction a t → Direction (a, b) t
  DRight :: Direction b t → Direction (a, b) t
```

The type of a direction is indexed by the environment type it points into and the component type it points to. Note that the type of *DVar* is specified to work with only environment types marked with *Var*; this is for ensuring that a direction goes all the way down to an actual component at a variable position of the pattern, rather than stopping half-way and pointing to a sub-tree which include more than one component. For example, for the environment type (4) for the cons pattern, only two directions are valid, namely *DLeft DVar* and *DRight DVar*, whereas *DVar* alone would point to the entire environment instead of one of the variable positions, and is ruled out by typechecking (in the sense that it is impossible for $DVar$ to have type $Direction\ (Var\ a, Var\ [a])\ b$ for any b). It is easy to extract a component from an environment following a direction:

```
retrieve :: Direction env a → env → a
retrieve DVar       (Var x) = x
retrieve (DLeft d)  (x, _)  = retrieve d x
retrieve (DRight d) (_, y)  = retrieve d y
```

Now we can define *expressions*, which are similar to patterns but include directions rather than variables, to represent the body of rearranging λ-expressions:

```
data Expr env a where
  EDir   :: Direction env a → Expr env a
  EConst :: (Eq a) ⇒ a →                      Expr env a
  EProd  :: Expr env a → Expr env b →         Expr env (a, b)
  ELeft  :: Expr env a →                      Expr env (Either a b)
  ERight :: Expr env b →                      Expr env (Either a b)
  EIn    :: (InOut a) ⇒ Expr env (F a) →      Expr env a
```

For example, the rearranging λ-expression

$$\lambda(x : xs) \rightarrow (x, xs) \tag{5}$$

is represented by the cons pattern (3) and the pair expression

$$EProd\ (EDir\ (DLeft\ DVar))\ (EDir\ (DRight\ DVar)) \tag{6}$$

Evaluating an expression under an environment is similar to inverse pattern matching:

```
eval :: Expr env a → env → a
eval (EDir d)       env = retrieve d env
eval (EConst c)     env = c
eval (l `EProd` r)  env = (eval l env, eval r env)
eval (ELeft e)      env = Left  (eval e env)
eval (ERight e)     env = Right (eval e env)
eval (EIn e)        env = inn (eval e env)
```

The type of $RearrV$ is then:

$$RearrV :: Pat\ v\ env \rightarrow Expr\ env\ v' \rightarrow BiGUL\ s\ v' \rightarrow BiGUL\ s\ v$$

Note that in the type of $RearrV$, the types of the pattern and expression share the same environment type index, ensuring that the directions in the expression can only refer to the variable positions in the pattern. And the *put* behaviour of $RearrV$ is simply:

```
put (RearrV p e b) s v = do env ← deconstruct p v
                            put b s (eval e env)
```

Inverse evaluation of rearranging λ-expressions. For the *get* direction, after executing the inner BiGUL program to obtain an intermediate view, we should reverse the roles of the pattern and body in the rearranging λ-expression $\lambda p \to e$, using e as a (possibly non-linear) pattern to match the intermediate view, and computing the final view by evaluating p. For example, the *put* direction of view rearrangement with the λ-expression (5) turns a view list into a pair, on which the inner program operates; in the *get* direction, the inner program will extract from the source an intermediate view pair, which should be converted back to a list by the inverse λ-expression $\lambda(x, xs) \to (x : xs)$. In more detail, given an intermediate view pair (x, xs), we match it with the pair expression (6), and see that x is associated with the direction *DLeft DVar* and xs with *DRight DVar*. From such associations we can reconstruct an environment of type (4) with x and xs in the right places, and then we can evaluate the cons pattern (3) in this reconstructed environment, arriving at the final view $x : xs$.

In general, the intermediate view will be decomposed according to the body expression, and eventually each of its components will be paired with a direction indicating which variable position the component should go into in the reconstructed environment. To do the reconstruction, we can prepare a "container" which is similar to an environment except that the variable positions are initially empty. For each pair of a component and a direction, we try to put that component into the place in the container pointed to by the direction; if two components are put into the same position (indicating that the λ-expression uses a variable more than once), then they must be equal. In the end, we check that all places in the container are filled, and then use it as an environment to evaluate the pattern. Again, to compute the type of containers from a pattern, we add a third index to *Pat*:

```
data Pat a env con where
  PVar    :: Eq a ⇒ Pat a (Var a) (Maybe a)
  PConst  :: Eq a ⇒ a →                     Pat a () ()
  PProd   :: Pat a a' a'' → Pat b b' b'' →  Pat (a, b) (a', b') (a'', b'')
  PLeft   :: Pat a a' a'' →                 Pat (Either a b) a' a''
  PRight  :: Pat b b' b'' →                 Pat (Either a b) b' b''
  PIn     :: InOut a ⇒ Pat (F a) b c →      Pat a b c
```

A container type is just like an environment type except that the variable positions give rise to *Maybe* instead of *Var*. For the cons example, the computed container type is

$$(Maybe\ a, Maybe\ [a]) \tag{7}$$

The first step—matching a value with an expression—can then be implemented as:

```
uneval :: Pat a env con → Expr env b → b → con → Maybe con
uneval p (EDir d)    x      con = unevalD p d x con
uneval p (EConst c)  x      con = if c == x then return con
```

```
                                                    else  Nothing
uneval p (EProd l r) (x, y)     con = uneval p l x con >>= uneval p r y
uneval p (ELeft e)   (Left x)   con = uneval p e x con
uneval p (ELeft _)   x          con = Nothing
uneval p (ERight e)  (Right x)  con = uneval p e x con
uneval p (ERight _)  x          con = Nothing
uneval p (EIn e)     x          con = uneval p e (out x) con

unevalD :: Pat a env con → Direction env b → b → con → Maybe con
unevalD PVar           DVar       x (Just y)     = if x == y
                                                   then return (Just x)
                                                   else  Nothing
unevalD PVar           DVar       x Nothing      = return (Just x)
unevalD (PConst c)     _          x con          = return con
unevalD (l `PProd` r)  (DLeft  d) x (conl, conr) = liftM (, conr)
                                                      (unevalD l d x conl)
unevalD (l `PProd` r)  (DRight d) x (conl, conr) = liftM (conl, )
                                                      (unevalD r d x conr)
unevalD (PLeft p)      d          x con          = unevalD p d x con
unevalD (PRight p)     d          x con          = unevalD p d x con
unevalD (PIn p)        d          x con          = unevalD p d x con
```

This function *uneval* initially takes an empty container, which is generated by:

```
emptyContainer :: Pat v env con → con
emptyContainer PVar          = Nothing
emptyContainer (PConst c)    = ()
emptyContainer (l `PProd` r) = (emptyContainer l, emptyContainer r)
emptyContainer (PLeft p)     = emptyContainer p
emptyContainer (PRight p)    = emptyContainer p
emptyContainer (PIn p)       = emptyContainer p
```

And then we can try to convert a container to an environment, checking whether the container is full in the process:

```
fromContainerV :: Pat v env con → con → Maybe env
fromContainerV PVar          Nothing      = Nothing
fromContainerV PVar          (Just v)     = return (Var v)
fromContainerV (PConst c)    con          = return ()
fromContainerV (l `PProd` r) (conl, conr) = liftM2 (, )
                                               (fromContainerV l  conl)
                                               (fromContainerV r conr)
fromContainerV (PLeft p)     con          = fromContainerV pat con
fromContainerV (PRight p)    con          = fromContainerV pat con
fromContainerV (PIn p)       con          = fromContainerV pat con
```

We can let out a sigh of relief once we successfully get hold of an environment, since the last step—inverse pattern matching—is total. To sum up:

$$
\begin{aligned}
get\ (RearrV\ p\ e\ b)\ s = \mathbf{do}\ & v' \quad \leftarrow get\ b\ s \\
& con \leftarrow uneval\ p\ e\ v'\ (emptyContainer\ p) \\
& env \leftarrow fromContainerV\ p\ con \\
& return\ (construct\ p\ env)
\end{aligned}
$$

To be concrete, let us go through the steps of inverse rearranging in the cons example. Starting with an intermediate view (x, xs) and an empty container $(Nothing, Nothing)$ of type (7), *uneval* will invoke *unevalD* twice, the first time updating the container to $(Just\ x, Nothing)$ and the second time to $(Just\ x, Just\ xs)$. The resulting container is full, and thus *fromContainerV* will successfully turn it into an environment $(Var\ x,\ Var\ xs)$ of type (4), in which we evaluate the cons pattern (3) and obtain $x : xs$.

Conceptually, this is just reversing pattern matching and expression evaluation. To actually prove the well-behavedness, though, we need to reason about stateful computation (which is what *uneval* essentially is), which involves coming up with suitable invariants and proving that they are maintained throughout the computation.

It is interesting to mention that there would be a catch if we designed this combinator from the *get* direction: It is tempting to think that, since a rearranging λ-expression gives rise to a partial isomorphism, which can be lifted to a lens, we can simply compose the lens lifted from the isomorphism with the inner lens to give a lens semantics to *RearrV*. This would result in a redundant computation of an intermediate source which is immediately discarded, and now the success of the whole computation would unnecessarily depend on that of the intermediate source. To eliminate the redundant computation, we would need to use a special composition which composes a lens directly with an isomorphism on the right. Such a need would be hard to notice since the *get* behaviour of the two compositions are the same; that is, we really have to think in terms of *put* to see that the special composition is needed.

5.7 Summary

In one (long) section, we have examined the internals of BiGUL. After seeing the definition of (well-behaved) lenses that takes partiality explicitly into account, we have gone through the development of most of BiGUL's constructs and justified their well-behavedness—in the case of *Prod*, we have even seen a more formal and detailed well-behavedness proof. The *Case* construct is the most interesting one in terms of its design for achieving bidirectionality, while the rearrangement operations showcase more advanced datatype-generic programming techniques in Haskell for guaranteeing type safety. We will now shift our focus back to BiGUL programming, this time looking at some larger examples.

6 Position-, Key-, and Delta-Based List Alignment

In the next three sections, we will talk about some applications in BiGUL, starting with the list alignment problem. List alignment is one of the tasks that frequently show up when developing bidirectional applications. When the source and view are both lists, and the *get* direction (i.e., the consistency relation) is a *map*, how do we put an updated view—the updates on which might involve insertions, deletions, in-place modifications, and reordering—into the source? This topic has be treated by Barbosa et al.'s matching lenses [1], which are special-purpose lenses into which several fixed alignment strategies are hard-coded. Below we will see how a number of alignment strategies can be programmed with BiGUL's general-purpose constructs, instead of having to extend the language with special-purpose alignment constructs.

Throughout the section, we use a concrete example to introduce three variations of list alignment. Suppose that we represent a payroll database as a list. (This is a slightly inadequate setting for explaining list alignment, because entries in a database are usually unordered. But let us assume that order matters.) Each entry is a triple—more precisely, a pair whose second component is again a pair—consisting of an identification number ("id" henceforth), a name, and a salary number:

```
type Source = (Id, (Name, Salary))
type Id     = Int
type Name   = String
type Salary = Int
```

For example, here is a sample payroll database:

```
employees :: [Source]
employees = [(0, ("Zhenjiang", 1000))
            ,(1, ("Josh"     , 400  ))
            ,(2, ("Jeremy"   , 2000))]
```

Suppose that the human resource department is in charge of hiring or sacking employees but does not handle salary numbers, so the entries of the database are presented to them only as pairs of ids and names:

```
type View = (Id, Name)
```

For example, *employees* is presented to them as

```
[(0, "Zhenjiang"), (1, "Josh"), (2, "Jeremy")]
```

on which they can make modifications. It is easy to write a BiGUL program to synchronize the source and view elements:

```
bx :: BiGUL Source View
bx = $(rearrV [[ λ(id, name) → (id, (name, ())) ]])$
        Replace ‘Prod’ (Replace ‘Prod’ Skip (const ()))
```

The problem is then how the correspondences between sources and views in the two lists can be determined, so that *bx* can be applied to the right pairs.

6.1 Position-Based Alignment

As a first exercise, we consider the simplest strategy, which matches source and view elements by their positions in the lists. If the source list has more elements than the view list, the extra elements at the tail are simply dropped; if the source list has fewer elements, then new source elements have to be created, which we can specify as a function:

$$\begin{array}{l} \mathit{cr} :: \mathit{View} \rightarrow \mathit{Source} \\ \mathit{cr}\ (i, n) = (i, (n, 0)) \end{array}$$

The salary is set to zero, which could be taken care of by, say, the accounting department later. We will use *bx* and *cr* as the element synchronizer and creator respectively for our payroll database throughout this section, but our alignment programs will not be restricted to the payroll database setting—we will develop our alignment programs generically, setting the source and view types as polymorphic type parameters (s and v below) and also the element synchronizer and element creator as parameters (b and c below), so the alignment programs can be widely applicable. Here is how we implement position-based alignment, which is fairly standard:

$$\begin{array}{l} \mathit{posAlign} :: (\mathit{Show}\ s, \mathit{Show}\ v) \Rightarrow \mathit{BiGUL}\ s\ v \rightarrow (v \rightarrow s) \rightarrow \mathit{BiGUL}\ [s]\ [v] \\ \mathit{posAlign}\ b\ c = \mathit{Case} \\ \quad [{}^{\$}(\mathit{normalSV}\ {}^{\mathbb{P}}[\![\, [\,] \,]\!]\ {}^{\mathbb{P}}[\![\, [\,] \,]\!]\ {}^{\mathbb{P}}[\![\, [\,] \,]\!]) \\ \quad\quad \Longrightarrow {}^{\$}(\mathit{update}\ {}^{\mathbb{P}}[\![\, [\,] \,]\!]\ {}^{\mathbb{P}}[\![\, [\,] \,]\!]\ {}^{\mathbb{D}}[\![\ ]\!]) \\ \quad , {}^{\$}(\mathit{normalSV}\ {}^{\mathbb{P}}[\![\, \_ : \_ \,]\!]\ {}^{\mathbb{P}}[\![\, \_ : \_ \,]\!]\ {}^{\mathbb{P}}[\![\, \_ : \_ \,]\!]) \\ \quad\quad \Longrightarrow {}^{\$}(\mathit{update}\ {}^{\mathbb{P}}[\![\, x : \mathit{xs} \,]\!]\ {}^{\mathbb{P}}[\![\, x : \mathit{xs} \,]\!]\ {}^{\mathbb{D}}[\![\, x = b; \mathit{xs} = \mathit{posAlign}\ b\ c \,]\!]) \\ \quad , {}^{\$}(\mathit{adaptiveSV}\ {}^{\mathbb{P}}[\![\, \_ : \_ \,]\!]\ {}^{\mathbb{P}}[\![\, [\,] \,]\!]) \\ \quad\quad \Longrightarrow \lambda\_\ \_ \rightarrow [\,] \\ \quad , {}^{\$}(\mathit{adaptiveSV}\ {}^{\mathbb{P}}[\![\, [\,] \,]\!]\ {}^{\mathbb{P}}[\![\, \_ : \_ \,]\!]) \\ \quad\quad \Longrightarrow \lambda\_\ (v : \_) \rightarrow [c\ v] \\ \quad] \end{array}$$

The normal branches deal with the situations where both lists are empty or non-empty, and the adaptive branches remove or create elements when the lengths of the two lists differ.

The *get* direction of *posAlign* does exactly what we want it to do:

```
*Alignment> get (posAlign bx cr) employees
Just [(0,"Zhenjiang"),(1,"Josh"),(2,"Jeremy")]
```

It should be quite obvious, though, that the *put* direction is not so useful for our purpose. If we sack Josh:

$$\begin{array}{l} \mathit{updatedEmployees0} :: [\,\mathit{View}\,] \\ \mathit{updatedEmployees0} = [(0, \texttt{"Zhenjiang"}), (2, \texttt{"Jeremy"})] \end{array}$$

then the database will be updated to:

```
*Alignment> put (posAlign bx cr) employees updatedEmployees0
Just [(0,("Zhenjiang",1000)),(2,("Jeremy",400))]
```

where Jeremy inadvertently gets Josh's original salary. Even if we do not remove any employee, we may still want to reorder them:

$$\begin{aligned} &\mathit{updatedEmployees1} :: [\mathit{View}] \\ &\mathit{updatedEmployees1} = [(2,\texttt{"Jeremy"}),(0,\texttt{"Zhenjiang"}),(1,\texttt{"Josh"})] \end{aligned}$$

and now everyone gets the wrong salary:

```
*Alignment> put (posAlign bx cr) employees updatedEmployees1
Just [(2,("Jeremy",1000)),(0,("Zhenjiang",400)),(1,("Josh",2000)
   )]
```

This first exercise shows that the alignment problem is inherently one that should be solved from the *put* direction. It is easy to implement the *get* direction correctly, but what matters is the *put* behavior.

6.2 Key-Based Alignment

A more reasonable strategy is to match source and view elements by some *key* value. In our example, we can use the id as the key. Key-based alignment might seem much more complex than position-based alignment, but, in fact, we can just revise *posAlign* to get a BiGUL program for key-based alignment!

First of all, we need to somehow obtain the keys. In our example, on both the source and view we can use *fst* to extract the key value. In general, we can further parametrize the alignment program with key extraction functions $ks :: s \to k$ and $kv :: v \to k$ for some type k of key values:

$$\begin{aligned} \mathit{keyAlign} &:: (\mathit{Show}\ s, \mathit{Show}\ v, \mathit{Eq}\ k) \\ &\quad \Rightarrow (s \to k) \to (v \to k) \to \mathit{BiGUL}\ s\ v \to (v \to s) \to \mathit{BiGUL}\ [s]\ [v] \end{aligned}$$

The first normal branch of *posAlign* still works perfectly. As for the second normal branch, we should revise the main condition to also require that the head elements of the two lists have the same key value:

$$\lambda(s : ss)\ (v : vs) \to ks\ s == kv\ v$$

The first adaptive branch, again, works well. The second adaptive branch, on the other hand, is no longer applicable: since the main condition of the second normal branch has been tightened, it is no longer the case that this adaptive branch will receive only empty source lists. In fact, whether the source list is empty or not is irrelevant here—what matters now is whether the key of the first view is in the source list. If it is, then we bring the (first) source element with the same key value to the head position, and the second normal branch can take over; otherwise, we create a new source element. This gives us key-based alignment:

```
keyAlign :: forall s v k. (Show s, Show v, Eq k)
         ⇒ (s → k) → (v → k) → BiGUL s v → (v → s) → BiGUL [s] [v]
keyAlign ks kv b c = Case
  [$(normalSV ℙ⟦ [] ⟧ ℙ⟦ [] ⟧ ℙ⟦ [] ⟧)
     ⟹ $(update ℙ⟦ [] ⟧ ℙ⟦ [] ⟧ 𝔻⟦ ⟧)
  , $(normal ⟦ λ(s : ss) (v : vs) → ks s == kv v ⟧ ℙ⟦ _ : _ ⟧)
     ⟹ $(update ℙ⟦ x : xs ⟧ ℙ⟦ x : xs ⟧ 𝔻⟦ x = b; xs = keyAlign ks kv b c ⟧)
  , $(adaptiveSV ℙ⟦ _ : _ ⟧ ℙ⟦ [] ⟧)
     ⟹ λ_ _ → []
  , $(adaptive ⟦ λss (v : vs) → kv v ∈ map ks ss ⟧)
     ⟹ λss (v : _) → uncurry (:) (extract (kv v) ss)
  , $(adaptiveSV ℙ⟦ _ ⟧ ℙ⟦ _ : _ ⟧)
     ⟹ λss (v : _) → c v : ss
  ]
  where
    extract :: k → [s] → (s, [s])
    extract k (x : xs) | ks x == k  = (x, xs)
                       | otherwise = let (y, ys) = extract k xs
                                     in  (y, x : ys)
```

Note that the program does not assume that keys are unique—if there are n view elements having the same key, then the first n source elements with that key will be retained and synchronised with those view elements in order. This strategy is a somewhat arbitrary choice, but can be changed by, for example, using a different *extract*. (On the other hand, in practice it is probably wiser to enforce uniqueness of keys, so that we can be sure which source element will be used to match a view element, and do not need to rely on the choices made by the implementation.)

Back to our payroll database example. The *get* direction behaves the same:

```
*Alignment> get (keyAlign fst fst bx cr) employees
Just [(0,"Zhenjiang"),(1,"Josh"),(2,"Jeremy")]
```

Unlike position-based alignment, view element deletion can now be reflected correctly:

```
*Alignment> put (keyAlign fst fst bx cr) employees
    updatedEmployees0
Just [(0,("Zhenjiang",1000)),(2,("Jeremy",2000))]
```

And reordering as well:

```
*Alignment> put (keyAlign fst fst bx cr) employees
    updatedEmployees1
Just [(2,("Jeremy",2000)),(0,("Zhenjiang",1000)),(1,("Josh",400)
    )]
```

So it seems that key-based alignment is just what we need. Indeed, key-based alignment usually works well, but there is an important assumption: the key values should not be changed. If, for example, we decide to assign a different id to Josh:

$$\mathit{updatedEmployees2} :: [\mathit{View}]$$
$$\mathit{updatedEmployees2} = [(0, \texttt{"Zhenjiang"}), (100, \texttt{"Josh"}), (1, \texttt{"Jeremy"})]$$

Then the effect is the same as sacking Josh and then hiring him again, and his salary is thus reset:

```
*Alignment> put (keyAlign fst fst bx cr) employees
    updatedEmployees2
Just [(0,("Zhenjiang",1000)),(100,("Josh",0)),(1,("Jeremy",400))
    ]
```

The problem is that we cannot distinguish modification from deletion and insertion pairs. To be able to have such distinction, we need the notion of *deltas* [4], which allows us to explicitly represent and keep track of the correspondences between source and view elements.

6.3 Delta-Based Alignment

A (horizontal) *delta* between a source list and a view list is a list of pairs of corresponding positions:

$$\textbf{type}\ \mathit{Delta} = [(\mathit{Int}, \mathit{Int})]$$

For example, the delta we have in mind between the source list *employees* and the view list *updatedEmployees2* is $[(0,0),(1,1),(2,2)]$, which, in particular, associates the source and view entries for Josh since $(1,1)$ is included, instead of $[(0,0),(2,2)]$, which indicates that Josh's source entry does not correspond to any view entry and should be deleted, and that Josh's view entry does not correspond to any source entry and is thus new. Deltas can easily represent reordering as well. For example, we would supply the delta between *employees* and *updatedEmployees1* as $[(0,1),(1,2),(2,0)]$, associating the 0th element in the source—namely the one for Zhenjiang—with the 1st element in the view, and so on. Comparing this treatment with the key-based one, we might say that keys are "poor man's correspondences", which are not as explicit and unambiguous as *Delta*. A *Delta* between source and view lists directly describes the accurate correspondences between them, whereas with keys the correspondences can only be inferred, sometimes inaccurately.

So the input now includes not only source and view lists but also a delta between them. Recall key-based alignment: what it does overall is to bring the first matching source element to the front for each view element, so the source list is updated throughout execution, with the links between the source and view elements gradually and implicitly restored. If we are doing something similar with delta-based alignment, then when the source list is updated, the delta should also be updated to reflect the restored consistency. This suggests that the delta should be paired with the source list, so that it can be updated. The type we use for the delta-based alignment program is thus:

```
deltaAlign :: (Show s, Show v)
           ⇒ BiGUL s v → (v → s) → BiGUL ([s], Delta) [v]
```

Here we take a simpler approach to implementing *deltaAlign*, analyzing the problem into just two cases: The delta can tell us either that the source and view elements are all in correspondence, in which case a simple position-based alignment suffices, or that we need to do some rearrangement of the source elements, which can be done by adaptation. In BiGUL:

```
idDelta    :: [s] → Delta
idDelta ss = [(i, i) | i ← [0 .. length ss]]
deltaAlign :: (Show s, Show v)
           ⇒ BiGUL s v → (v → s) → BiGUL ([s], Delta) [v]
deltaAlign b c = Case
  [$(normal ⟦ λ(ss, d) vs → length ss == length vs ∧ d == idDelta ss ⟧
             ℙ⟦ _ ⟧)
     ⟹ $(rearrV ⟦ λvs → (vs, ()) ⟧)$ posAlign b c ‘Prod’ Skip (const ())
  , $(adaptive ⟦ λ_ _ → otherwise ⟧)
     ⟹ λ(ss, d) vs →
           let d'  = map swap d
               ss' = [maybe (c v) (ss!!) (lookup j d') | (v, j) ← zip vs [0 ..]]
           in (ss', idDelta ss')
  ]
```

The source and view lists are in full correspondence if and only if they have the same length and the delta associates all their elements positionally. This full positional delta can be computed by *idDelta*. When this is the case, it suffices to call *posAlign* to carry out element-wise synchronization, since no rearrangement is required. Otherwise, we enter the adaptive branch, which constructs a new source list in full correspondence with the view list, drawing elements from the original source list or creating new ones as the delta dictates. The new source list is in full correspondence with the view list, so the delta we pair with it is the one computed by *idDelta*.

Only when performing *put* does a delta make sense. When performing *get*, however, we still need to supply a delta since it is part of the source; but there is a natural choice, namely *idDelta*. So we define:

```
putDeltaAlign :: (Show s, Show v)
              ⇒ BiGUL s v → (v → s) → [s] → Delta → [v] → Maybe [s]
putDeltaAlign b c ss d vs = fmap fst (put (deltaAlign b c) (ss, d) vs)
getDeltaAlign :: (Show s, Show v)
              ⇒ BiGUL s v → (v → s) → [s] → Maybe [v]
getDeltaAlign b c ss = get (deltaAlign b c) (ss, idDelta ss)
```

It is easy to prove that, given the same b and c, these two functions do form a lens. The key observation is that the delta produced by *put* (*deltaAlign* b c) is necessarily the one computed by *idDelta*, so, for example, in PutGet, throwing away the delta in the *put* direction is fine because it can be recomputed by *idDelta*, and the *get* direction can resume from exactly the same source pair.

Back to our example. We can now update Josh's id without resetting his salary by providing a full delta indicating that there are only in-place updates:

```
*Alignment> putDeltaAlign bx cr employees [(0,0), (1,1), (2,2)]
    updatedEmployees2
Just [(0,("Zhenjiang",1000)),(100,("Josh",400)),(1,("Jeremy
    ",2000))]
```

Besides obvious modifications like reordering, we can also do some fairly subtle modifications now: If we actually sack Josh and replace him with a new Josh (inheriting the original Josh's id) whose salary should be reset (to be reconsidered by the accounting department), we can say so by providing a partial delta:

```
*Alignment> putDeltaAlign bx cr employees [(0,0), (2,2)] =<<
    getDeltaAlign bx cr employees
Just [(0,("Zhenjiang",1000)),(1,("Josh",0)),(2,("Jeremy",2000))]
```

One alignment to rule them all. Where do deltas come from? In general, we may provide a special view editor which monitors how the view is modified and produces a suitable delta. But in more specialized scenarios, deltas can simply be computed by, for example, comparing the source and view. We can formalize this delta computation as:

$$\textbf{type}\ \mathit{DeltaStrategy}\ s\ v = [s] \rightarrow [v] \rightarrow \mathit{Delta}$$

and further parametrize *putDeltaAlign*:

$$\begin{aligned}
&\mathit{putDeltaAlignS} :: (\mathit{Show}\ s, \mathit{Show}\ v) \Rightarrow \mathit{DeltaStrategy}\ s\ v \\
&\qquad\qquad\qquad \rightarrow \mathit{BiGUL}\ s\ v \rightarrow (v \rightarrow s) \rightarrow [s] \rightarrow [v] \rightarrow \mathit{Maybe}\ [s] \\
&\mathit{putDeltaAlignS}\ \mathit{dst}\ b\ c\ \mathit{ss}\ \mathit{vs} = \mathit{putDeltaAlign}\ b\ c\ \mathit{ss}\ (\mathit{dst}\ \mathit{ss}\ \mathit{vs})\ \mathit{vs}
\end{aligned}$$

Position-based and key-based alignment can then be seen as special cases of delta-based alignment using specific delta-computing strategies. For position-based alignment, we simply compute the identity delta:

$$\begin{aligned}
&\mathit{byPosition} :: \mathit{DeltaStrategy}\ s\ v \\
&\mathit{byPosition}\ \mathit{ss}\ \_ = \mathit{idDelta}\ \mathit{ss}
\end{aligned}$$

And for key-based alignment, we compute a delta associating source and view elements with the same key:

$$\begin{aligned}
&\mathit{byKey} :: \mathit{Eq}\ k \Rightarrow (s \rightarrow k) \rightarrow (v \rightarrow k) \rightarrow \mathit{DeltaStrategy}\ s\ v \\
&\mathit{byKey}\ \mathit{ks}\ \mathit{kv}\ \mathit{ss}\ \mathit{vs} =
\end{aligned}$$

```
let sis = zip ss [0 ..]
in  catMaybes [ fmap (λ(_, i) → (i, j)) (find (λ(s, _) → ks s == kv v) sis)
              | (v, j) ← zip vs [0 ..]]
```

We can check that these strategies indeed give us position-based and key-based alignment:

```
*Alignment> putDeltaAlignS byPosition bx cr employees
    updatedEmployees0
Just [(0,("Zhenjiang",1000)),(2,("Jeremy",400))]
*Alignment> putDeltaAlignS (byKey fst fst) bx cr employees
    updatedEmployees1
Just [(2,("Jeremy",2000)),(0,("Zhenjiang",1000)),(1,("Josh",400)
    )]
```

7 Bidirectionalizing Relational Queries with BiGUL

In work on relational databases, the view-update problem is about how to translate update operations on the view table to corresponding update operations on the source table properly[2]. Relational lenses [3] try to solve this problem by providing a list of combinators that let the user write get functions (queries) with specified updated policies for put functions (updates); however this can only provide limited control of update policies. To resolve this problem, we define a new library BRUL [17], where two *putback*-based combinators (operators) are designed to specify update policies, from which forward queries (selection, projection, join) can be automatically derived.

- *align* is to update a source list with a view list by aligning part of source elements filtered by a predicate with view elements according to a matching criteria between source element and view element;
- *unjoin* is to decompose a join view to update two sources.

In this section, we will focus on *align*. As will be seen in Sect. 7.3, it can describe more flexible update strategies (related to selection/projection queries) than relational lenses, while the well-behavedness is guaranteed for free.

7.1 Relational Database Representation

A relational table (*RT*) is denoted by a list of records (where the order does not really matter), and each record (*Record*) is denoted by a list of attributes of type *RType*, which could be an integer, a string, a floating point number, or a double-precision floating point number.

```
type RT     = [Record]
type Record = [RType]
```

[2] The text of this section is adapted from our BX 2016 paper [17].

Track	Date	Rating	Album	Quantity
Lullaby	1989	3	Galore	2
Lullaby	1989	3	Show	3
Lovesong	1989	5	Galore	2
Lovesong	1989	5	Paris	4
Trust	1992	4	Wish	5

Fig. 1. Source table

```
data RType = RInt Int
           | RString String
           | RFloat Float
           | RDouble Double
           deriving (Show, Eq, Ord)
```

To allow pattern matching on the newly defined algebraic data type *RType* in BiGUL, we need to add the following declaration.

```
deriveBiGULGeneric ''RType
```

Consider the table in Fig. 1 that stores five music track records, and each record contains its Track name, release Date, Rating, Album, and the Quantity of this Album. We can represent it as follows, where all the records have the same structure.

```
s = [[RString "Lullaby"  , RInt 1989, RInt 3, RString "Galore", RInt 1]
    ,[RString "Lullaby"  , RInt 1989, RInt 3, RString "Show"  , RInt 3]
    ,[RString "Lovesong", RInt 1989, RInt 5, RString "Galore", RInt 1]
    ,[RString "Lovesong", RInt 1989, RInt 5, RString "Paris" , RInt 4]
    ,[RString "Trust"    , RInt 1992, RInt 4, RString "Wish"  , RInt 5]
    ]
```

7.2 Relation Alignment

The alignment of two relational tables, which is related by a selection/projection query, is similar to the key-based list alignment in Sect. 6. The difference is that we need to consider filtering on (i.e., selection of) the source records based on a condition.

Let us see how to extend *keyAlign* (in Sect. 6) to implement the new align *pAlign* that can deal with filtering of source elements. We extend *keyAlign* with two new arguments; one is the predicate p for filtering source elements, and the other is the function h for hiding/concealing source elements if their corresponding elements are removed from the view. As seen below, *pAlign* has a similar case structure as that of *keyAlign*, except that we refine the third case of *keyAlign* into two cases (the third and the fourth cases of *pAlign*): the third case says that if the view v is empty but the first record in the source satisfies p, we should hide this record using h, and the fourth case says that if the first record of the source does not satisfy p, we simply ignore it and continue with the remaining records.

```
pAlign :: forall s v k. (Show s, Show v, Eq k)
        ⇒ (s → Bool)   -- predicate
        → (s → k) → (v → k) → BiGUL s v → (v → s)
        → (s → Maybe s)   -- conceal function
        → BiGUL [s] [v]
pAlign p ks kv b c h = Case
   [$(normalSV ℙ⟦[ ]⟧ ℙ⟦[ ]⟧ ℙ⟦[ ]⟧)
      ⟹ $(update ℙ⟦[ ]⟧ ℙ⟦[ ]⟧ 𝔻⟦ ⟧)
   , $(normal ⟦λ(s : ss) (v : vs) → p s ∧ ks s == kv v⟧ ⟦λ(s : ss) → p s⟧)
      ⟹ $(update ℙ⟦x : xs⟧ ℙ⟦x : xs⟧ 𝔻⟦x = b; xs = pAlign p ks kv b c h⟧)
   , $(adaptive ⟦λ(s : ss) v → p s ∧ null v⟧)
      ⟹ λ(s : ss) v → maybe [ ] (:[ ]) (h s) ++ ss
   , $(normal ⟦λ(s : ss) v → not (p s)⟧ ⟦λ(s : ss) → not (p s)⟧)
      ⟹ $(update ℙ⟦_ : xs⟧ ℙ⟦xs⟧ 𝔻⟦xs = pAlign p ks kv b c h⟧)
   , $(adaptive ⟦λss (v : vs) → kv v ∈ map ks (filter p ss)⟧)
      ⟹ λss (v: _) → uncurry (:) (extract (kv v) ss)
   , $(adaptiveSV ℙ⟦_⟧ ℙ⟦_: _⟧)
      ⟹ λss (v: _) → filterCheck p (c v) : ss
   ]
   where
     extract :: k → [s] → (s, [s])
     extract k (x : xs) | p x ∧ ks x == k = (x, xs)
                        | otherwise        = let (y, ys) = extract k xs
                                             in  (y, x : ys)
     filterCheck p v | p v       = v
                     | otherwise = error "error in filter checking"
```

To test, recall the example in Sect. 6. Consider the following use of *pAlign*, denoting that the view is selected from those records from the source whose salary is greater than 1000, and that if a view record is removed, the corresponding record in the source will be removed (and thus hidden).

```
pSelProj = pAlign (λ(k, (n, s)) → s > 1000) fst fst bx cr′ (const Nothing)
   where cr′ (k, n) = (k, (n, 2000))
```

We have:

```
*Brul> get pSelProj employees
Just [(2,"Jeremy")]
*Brul> put pSelProj employees updatedEmployees0
Just [(0,("Zhenjiang",1000)),(1,("Josh",400)),(0,("Zhenjiang
   ",2000)),(2,("Jeremy",2000))]
```

7.3 Describing Update Policies in Selection/Projection

With *pAlign*, we can describe various update policies for the selection/projection queries. To be concrete, consider the following selection/projection query:

$$\begin{array}{ll}\textbf{select} & \mathit{Track}, \mathit{Rating}, \mathit{Album}, \mathit{Quantity}\ \textbf{as}\ v\\ \textbf{from} & s\\ \textbf{where} & \mathit{Quantity} > 2\end{array}$$

which extracts the track, rating, album and quality information from those music tracks in the source s whose quantity is greater than 2. Let us see how to write a single BiGUL program so that its *get* does the above query and its *put* describes a specific update policy.

The first BiGUL program is *u0* below.

```
u0    :: RType → BiGUL [Record] [Record]
u0 d = pAlign
          (λr → (r !! 4) > RInt 2)
          (λs → (s !! 0, s !! 3))
          (λv → (v !! 0, v !! 2))
          $(update ℙ⟦ (t : _ : r : a : q : [ ]) ⟧
                   ℙ⟦ (t : r : a : q : [ ]) ⟧
                   𝔻⟦ t = Replace; r = Replace; a = Replace; q = Replace ⟧)
          (λ(t : r : a : q : [ ]) → (t : d : r : a : q : [ ]))
          (const Nothing)
```

It tries to match the source records whose *Quantity* is greater than 2 with the view records by the key (*Track*, *Album*). There are three cases:

- A source record is matched with a view record: we first use a rearrangement function to rearrange the view from a four-element list $[t, r, a, q]$ to a five-element list $[t, \_, r, a, q]$ with the second element matched against a widecard. This rearrangement function reshapes the view to match the shape of the source. Then, the element in the source is *Replace*d by the corresponding element in the view.
- A view record that has no matching source record: a new source record is created with a default value d filled into the Date.
- A source record that has no matching view record: we simply delete this record by returning *Nothing*.

Now if we wish to hide the source record by setting its *Quantity* to 0 rather than deleting it if it has no matching view record, we could simply change the last line of *u0* and get *u1* as follows.

```
u1    :: RType → BiGUL [Record] [Record]
u1 d = pAlign
          (λr → (r !! 4) > RInt 2)
```

$$
\begin{array}{l}
(\lambda s \rightarrow (s \mathbin{!!} 0, s \mathbin{!!} 3)) \\
(\lambda v \rightarrow (v \mathbin{!!} 0, v \mathbin{!!} 2)) \\
{}^{\$}(\mathit{update}\ {}^{\mathbb{P}}[\![\, (t : \_ : r : a : q : [\,])\,]\!] \\
\qquad\qquad {}^{\mathbb{P}}[\![\, (t : r : a : q : [\,])\,]\!] \\
\qquad\qquad {}^{\mathbb{D}}[\![\, t = \mathit{Replace}; r = \mathit{Replace}; a = \mathit{Replace}; q = \mathit{Replace}\,]\!]) \\
(\lambda (t : r : a : q : [\,]) \rightarrow (t : d : r : a : q : [\,])) \\
(\lambda (t : d : r : a : \_ : [\,]) \rightarrow \mathit{Just}\ (t : d : r : a : \mathit{RInt}\ 0 : [\,]))
\end{array}
$$

To test, let us see some concrete running examples of using *u0*. Recall *s* defined in Sect. 7.1. We can confirm that *get* performs the query given at the start of this subsection.

```
*Brul> get (u0 (RInt 2000)) s
Just [[RString "Lullaby", RInt 3, RString "Show", RInt 3],[RString "
    Lovesong", RInt 5,RString "Paris",RInt 4],[RString "Trust",RInt 4,
    RString "Wish",RInt 5]]
```

Now suppose that we change the above result (view) to the following by raising the rating of *Lullaby* from 3 to 4, raising the quality of *lovesong* from 4 to 7, and deleting *Trust*:

$$
\begin{array}{l}
v = [\,[\mathit{RString}\ \texttt{"Lullaby"}, \mathit{RInt}\ 4, \mathit{RString}\ \texttt{"Show"}, \mathit{RInt}\ 3] \\
\quad\ , [\mathit{RString}\ \texttt{"Lovesong"}, \mathit{RInt}\ 5, \mathit{RString}\ \texttt{"Paris"}, \mathit{RInt}\ 7] \\
\quad\ ]
\end{array}
$$

We can reflect these changes to the source by performing *put* with *u0*.

```
*Brul> put (u0 (RInt 2000)) s v
Just [[RString "Lullaby",RInt 1989,RInt 3,RString "Galore",RInt 1],[
    RString "Lullaby",RInt 1989,RInt 4,RString "Show",RInt 3],[RString
    "Lovesong",RInt 1989,RInt 5,RString "Galore",RInt 1],[RString "
    Lovesong",RInt 1989,RInt 5,RString "Paris",RInt 7]]
```

In the updated source, the changes of rating and quality are correctly reflected, and the music track *Trust* is removed. Note that we may reflect the changes to the source by performing *put* with *u1*, another update strategy, and we will keep the music track *Trust* while setting its quality to be 0.

```
*Brul> put (u1 (RInt 2000)) s v
Just [[RString "Lullaby",RInt 1989,RInt 3,RString "Galore",RInt 1],[
    RString "Lullaby",RInt 1989,RInt 4,RString "Show",RInt 3],[RString
    "Lovesong",RInt 1989,RInt 5,RString "Galore",RInt 1],[RString "
    Lovesong",RInt 1989,RInt 5,RString "Paris",RInt 7],[RString "Trust
    ",RInt 1992,RInt 4,RString "Wish",RInt 0]]
```

8 Parsing and Reflective Printing

When we mention the *front-end* of a compiler, we usually think of a *parser* that turns *concrete syntax*, which is designed to be programmer-friendly and provides convenient syntactic sugar, into *abstract syntax*, which is concise, structured, and easily manipulable by the compiler back-end. There is another direction, though, in which a *printer* turns abstract syntax back into concrete syntax. This is useful, for example, for reporting the result of compiler optimizations done on abstract syntax to the programmer, who knows only concrete syntax. In this case, though, we would want to print the optimized program in a form that is as close to the original program as possible, so the programmer can spot what has changed—and not changed—correctly and more easily. This is where the notion of *reflective printing* comes in: By taking both the original concrete program and the optimized abstract program as input, we can try to retain the look of the original program as much as possible. Below we will use a simplified arithmetic expression language to explain how reflective printing can be implemented in BiGUL.

8.1 Well-Behavedness

It is probably obvious that the idea of reflective printing comes from *put* transformations; parsing, then, is the *get* direction. Before we proceed to implement parsing and reflective printing in BiGUL, a natural question to ask is: is well-behavedness meaningful in the context of parsing of reflective printing? The answer is yes, especially for PutGet: An abstract syntax tree (AST) may be thought of as a concise and canonical representation of a concrete program, so it would be strange if a concrete program printed from an AST could not be parsed back to the same AST. GetPut, on the other hand, is in fact not strong enough for our purpose, as it only says that, when an AST is unmodified, printing it reflectively to the original program does not change anything, whereas we would have liked to also say that "small" changes to the AST lead to only "small" changes to the concrete program. That is, we would like reflective printing to conform to some sort of least-change principle, a topic which is still unsettled and actively investigated by the BX community. It is at least a good start to have GetPut, though. We thus conclude that BiGUL is indeed a suitable language for implementing reflective printers and corresponding parsers.

8.2 Additive Expressions

Here we use a minimal example which is simple and yet can demonstrate what reflective printing is capable of. Consider the following abstract syntax of arithmetic expressions consisting of integer constants, addition, and subtraction:

```
data Arith = Num Int
           | Add Arith Arith
           | Sub Arith Arith
           deriving Show
```

This is a nice representation for the compiler, but we cannot expect the programmer to write something like "*Sub* (*Num* 1) (*Add* (*Num* 2) (*Num* 3))", and should provide a concrete syntax so that they can write "1 − (2 + 3)". Such a concrete syntax is usually defined in terms of a BNF grammar:

```
Exp    → Exp '+' Factor
       | Exp '-' Factor
       | Factor
Factor → Int
       | '-' Factor
       | '(' Exp ')'
```

The two-level structure of *Exp* and *Factor* ensures that plus and minus associate to the left by default; to change association, we should use parentheses. And, to spice up the problem a little, we allow minus to be used also as a negative sign, as specified by the second production rule for *Factor*. BiGUL deals with structured data only, so we should represent a string generated using this grammar as a concrete syntax tree of the following type:

```
data Exp = Plus Exp Factor
         | Minus Exp Factor
         | EF Factor
         | ENull
data Factor = Lit Int
            | Neg Factor
            | Paren Exp
            | FNull
```

Again, we need to provide one *deriveBiGULGeneric* statement for each of the above datatypes to allow BiGUL to operate on them:

```
deriveBiGULGeneric ''Arith
deriveBiGULGeneric ''Exp
deriveBiGULGeneric ''Factor
```

Apart from the *Null* constructors, which are inserted to represent incomplete trees that can occur during reflective printing, these two datatypes are in direct correspondence with the grammar, so it is easy to recover the string from a concrete syntax tree:

```
instance Show Exp where
  show (Plus  e f) = show e ++ "+" ++ show f
  show (Minus e f) = show e ++ "-" ++ show f
  show (EF      f) = show f
  show  ENull      = "."
instance Show Factor where
  show (Lit   n) = show n
  show (Neg   f) = "-" ++ show f
```

$$
\begin{array}{lll}
\mathit{show}\ (\mathit{Paren}\ \ e) & = & \texttt{"("} \mathbin{+\!\!+} \mathit{show}\ e \mathbin{+\!\!+} \texttt{")"} \\
\mathit{show}\ \ \mathit{FNull} & = & \texttt{"."}
\end{array}
$$

Conversely, using modern parser technologies like Haskell's `parsec` parser combinator library, we can easily implement a "concrete parser" that turns a string into a concrete syntax tree:

$$\mathit{parseExp} :: \mathit{String} \to \mathit{Exp}$$

The rest of the job is then to write a BiGUL program between *Exp* and *Arith*.

8.3 Reflective Printing in BiGUL

The program is basically a case analysis: For example, when the concrete side is a plus and the abstract side is an addition, they match, and we can go into their sub-trees recursively. For the concrete side, the right sub-tree is of type *Factor* instead of *Exp*, so in fact we will write two (mutually recursive) programs:

$$
\begin{array}{ll}
\mathit{pExpArith} & :: \mathit{BiGUL}\ \mathit{Exp}\ \mathit{Arith} \\
\mathit{pExpArith} & = \mathit{Case}\ \bot \\
\mathit{pFactorArith} & :: \mathit{BiGUL}\ \mathit{Factor}\ \mathit{Arith} \\
\mathit{pFactorArith} & = \mathit{Case}\ \bot
\end{array}
$$

The branch for plus and addition can then be written as:

$$^{\$}(\mathit{update}\ ^{\mathbb{P}}[\![\,\mathit{Plus}\ l\ r\,]\!]\ ^{\mathbb{P}}[\![\,\mathit{Add}\ l\ r\,]\!]\ ^{\mathbb{D}}[\![\,l = \mathit{pExpArith}; r = \mathit{pFactorArith}\,]\!])$$

Following the same line of thought, we can fill in other branches to relate all abstract constructors with concrete production rules:

$$
\begin{array}{l}
\mathit{pExpArith} :: \mathit{BiGUL}\ \mathit{Exp}\ \mathit{Arith} \\
\mathit{pExpArith} = \mathit{Case} \\
\quad [^{\$}(\mathit{normalSV}\ ^{\mathbb{P}}[\![\,\mathit{Plus}\ \_\ \_\,]\!]\ ^{\mathbb{P}}[\![\,\mathit{Add}\ \_\ \_\,]\!]\ ^{\mathbb{P}}[\![\,\mathit{Plus}\ \_\ \_\,]\!]) \\
\qquad \Longrightarrow {}^{\$}(\mathit{update}\ ^{\mathbb{P}}[\![\,\mathit{Plus}\ l\ r\,]\!]\ ^{\mathbb{P}}[\![\,\mathit{Add}\ l\ r\,]\!] \\
\qquad\qquad\qquad\qquad ^{\mathbb{D}}[\![\,l = \mathit{pExpArith}; r = \mathit{pFactorArith}\,]\!]) \\
\quad ,^{\$}(\mathit{normalSV}\ ^{\mathbb{P}}[\![\,\mathit{Minus}\ \_\ \_\,]\!]\ ^{\mathbb{P}}[\![\,\mathit{Sub}\ \_\ \_\,]\!]\ ^{\mathbb{P}}[\![\,\mathit{Minus}\ \_\ \_\,]\!]) \\
\qquad \Longrightarrow {}^{\$}(\mathit{update}\ ^{\mathbb{P}}[\![\,\mathit{Minus}\ l\ r\,]\!]\ ^{\mathbb{P}}[\![\,\mathit{Sub}\ l\ r\,]\!] \\
\qquad\qquad\qquad\qquad ^{\mathbb{D}}[\![\,l = \mathit{pExpArith}; r = \mathit{pFactorArith}\,]\!]) \\
\quad ,^{\$}(\mathit{normalSV}\ ^{\mathbb{P}}[\![\,\mathit{EF}\ \_\,]\!]\ ^{\mathbb{P}}[\![\,\_\,]\!]\ ^{\mathbb{P}}[\![\,\mathit{EF}\ \_\,]\!]) \\
\qquad \Longrightarrow {}^{\$}(\mathit{update}\ ^{\mathbb{P}}[\![\,\mathit{EF}\ t\,]\!]\ ^{\mathbb{P}}[\![\,t\,]\!] \\
\qquad\qquad\qquad\qquad ^{\mathbb{D}}[\![\,t = \mathit{pFactorArith}\,]\!]) \\
\quad] \\
\mathit{pFactorArith} :: \mathit{BiGUL}\ \mathit{Factor}\ \mathit{Arith} \\
\mathit{pFactorArith} = \mathit{Case} \\
\quad [^{\$}(\mathit{normalSV}\ ^{\mathbb{P}}[\![\,\mathit{Lit}\ \_\,]\!]\ ^{\mathbb{P}}[\![\,\mathit{Num}\ \_\,]\!]\ ^{\mathbb{P}}[\![\,\mathit{Lit}\ \_\,]\!]) \\
\qquad \Longrightarrow {}^{\$}(\mathit{update}\ ^{\mathbb{P}}[\![\,\mathit{Lit}\ i\,]\!]\ ^{\mathbb{P}}[\![\,\mathit{Num}\ i\,]\!]\ ^{\mathbb{D}}[\![\,i = \mathit{Replace}\,]\!])
\end{array}
$$

```
  , $(normalSV ℙ⟦Neg _⟧ ℙ⟦Sub (Num 0) _⟧ ℙ⟦Neg _⟧)
      ⟹ $(update ℙ⟦Neg t⟧ ℙ⟦Sub (Num 0) t⟧ 𝔻⟦t = pFactorArith⟧)
  , $(normalSV ℙ⟦Paren _⟧ ℙ⟦_⟧ ℙ⟦Paren _⟧)
      ⟹ $(update ℙ⟦Paren t⟧ ℙ⟦t⟧ 𝔻⟦t = pExpArith⟧)
  ]
```

This covers only "normal" cases though, namely when the source and view are "the same" except for parentheses and literals. What about the cases where the source and view have mismatched shapes? For these cases, we need adaptation. Corresponding to each branch we have already written, we add an adaptive branch which looks at the shape of the view only, throws away a mismatched source, and creates an incomplete one whose shape matches that of the view; the source will be completely created through recursive processing. For example, corresponding to the plus/addition branch, we write:

```
$(adaptiveSV ℙ⟦_⟧ ℙ⟦Add _ _⟧)
   ⟹ λ_ _ → Plus ENull FNull
```

The full programs are:

```
pExpArith :: BiGUL Exp Arith
pExpArith = Case
  [$(normalSV ℙ⟦Plus _ _⟧ ℙ⟦Add _ _⟧ ℙ⟦Plus _ _⟧)
      ⟹ $(update ℙ⟦Plus l r⟧ ℙ⟦Add l r⟧
                    𝔻⟦l = pExpArith; r = pFactorArith⟧)
  , $(normalSV ℙ⟦Minus _ _⟧ ℙ⟦Sub _ _⟧ ℙ⟦Minus _ _⟧)
      ⟹ $(update ℙ⟦Minus l r⟧ ℙ⟦Sub l r⟧
                    𝔻⟦l = pExpArith; r = pFactorArith⟧)
  , $(normalSV ℙ⟦EF _⟧ ℙ⟦_⟧ ℙ⟦EF _⟧)
      ⟹ $(update ℙ⟦EF t⟧ ℙ⟦t⟧
                    𝔻⟦t = pFactorArith⟧)
  , $(adaptiveSV ℙ⟦_⟧ ℙ⟦Add _ _⟧)
      ⟹ λ_ _ → Plus ENull FNull
  , $(adaptiveSV ℙ⟦_⟧ ℙ⟦Sub _ _⟧)
      ⟹ λ_ _ → Minus ENull FNull
  , $(adaptiveSV ℙ⟦_⟧ ℙ⟦_⟧)
      ⟹ λ_ _ → EF FNull
  ]
pFactorArith :: BiGUL Factor Arith
pFactorArith = Case
  [$(normalSV ℙ⟦Lit _⟧ ℙ⟦Num _⟧ ℙ⟦Lit _⟧)
      ⟹ $(update ℙ⟦Lit i⟧ ℙ⟦Num i⟧ 𝔻⟦i = Replace⟧)
  , $(normalSV ℙ⟦Neg _⟧ ℙ⟦Sub (Num 0) _⟧ ℙ⟦Neg _⟧)
```

$$
\begin{array}{l}
\quad\quad \Longrightarrow {}^{\$}(\mathit{update}\ {}^{\mathbb{P}}[\![\, \mathit{Neg}\ t \,]\!]\ {}^{\mathbb{P}}[\![\, \mathit{Sub}\ (\mathit{Num}\ 0)\ t \,]\!]\ {}^{\mathbb{D}}[\![\, t = \mathit{pFactorArith} \,]\!]) \\
\quad ,{}^{\$}(\mathit{normalSV}\ {}^{\mathbb{P}}[\![\, \mathit{Paren}\ \_ \,]\!]\ {}^{\mathbb{P}}[\![\, \_ \,]\!]\ {}^{\mathbb{P}}[\![\, \mathit{Paren}\ \_ \,]\!]) \\
\quad\quad \Longrightarrow {}^{\$}(\mathit{update}\ {}^{\mathbb{P}}[\![\, \mathit{Paren}\ t \,]\!]\ {}^{\mathbb{P}}[\![\, t \,]\!]\ {}^{\mathbb{D}}[\![\, t = \mathit{pExpArith} \,]\!]) \\
\quad ,{}^{\$}(\mathit{adaptiveSV}\ {}^{\mathbb{P}}[\![\, \_ \,]\!]\ {}^{\mathbb{P}}[\![\, \mathit{Num}\ \_ \,]\!]) \\
\quad\quad \Longrightarrow \lambda\_\ \_ \rightarrow \mathit{Lit}\ 0 \\
\quad ,{}^{\$}(\mathit{adaptiveSV}\ {}^{\mathbb{P}}[\![\, \_ \,]\!]\ {}^{\mathbb{P}}[\![\, \mathit{Sub}\ (\mathit{Num}\ 0)\ \_ \,]\!]) \\
\quad\quad \Longrightarrow \lambda\_\ \_ \rightarrow \mathit{Neg}\ \mathit{FNull} \\
\quad ,{}^{\$}(\mathit{adaptiveSV}\ {}^{\mathbb{P}}[\![\, \_ \,]\!]\ {}^{\mathbb{P}}[\![\, \_ \,]\!]) \\
\quad\quad \Longrightarrow \lambda\_\ \_ \rightarrow \mathit{Paren}\ \mathit{ENull} \\
\quad]
\end{array}
$$

8.4 Reflecting Optimizations and Evaluation Sequences

The BiGUL programs, being bidirectional, can be executed in the *put* direction as a reflective printer, or in the *get* direction as a parser. Let us look at parsing first. For example:

```
*BiYacc> get pExpArith (parseExp "(-(3+0))")
Just (Sub (Num 0) (Add (Num 3) (Num 0)))
```

Note that a unary minus is regarded as syntactic sugar, and is desugared into a subtraction whose left operand is zero. Also note that parentheses are turned into correct structure of the abstract syntax tree, and nothing more—excessive parentheses are cleanly discarded.

For reflective printing, as we mentioned, one application is reporting what compiler optimizations do. We can optimize the sub-expression $3 + 0$ by getting rid of the superfluous $+0$, for example, and the reflective printer will be able to retain the excessive parentheses:

```
*BiYacc> put pExpArith (parseExp "(-(3+0))") (Sub (Num 0) (Num
    3))
Just (-(3))
```

Notice also that the unary minus is preserved. If the original concrete expression uses a binary minus instead, it will be preserved as well:

```
*BiYacc> put pExpArith (parseExp "(0-(3+0))") (Sub (Num 0) (Num
    3))
Just (0-(3))
```

In the above example, the pair of parentheses around 3 is also preserved. This is more a coincidence, though—if we change *Sub* to *Add*, for example, the pair of parentheses will not be preserved:

```
*BiYacc> put pExpArith (parseExp "(0-(3+0))") (Add (Num 0) (Num
    3))
Just (0+3)
```

This behavior is indeed what we described with our BiGUL program: the concrete binary minus does not match the abstract *Add*, so the whole concrete expression `0-(3+0)` inside the outermost pair of parentheses is discarded, and a new concrete expression `0+3` is generated by adaptation. This behavior does not give us "least change", however: the pair of parentheses around 3 could have been kept. This is one example showing that, while GetPut (no view change implies no source change) is guaranteed by BiGUL, least-change behavior (small view change implies small source change) is another matter completely, and requires extra care and effort to achieve.

Another thing we can do is reflecting the steps in an evaluation sequence of an abstract syntax tree to concrete syntax. For example, starting from:

```
*BiYacc> get pExpArith (parseExp "1+(2+3)")
Just (Add (Num 1) (Add (Num 2) (Num 3)))
```

it takes two steps to evaluate this expression:

```
*BiYacc> put pExpArith (parseExp "1+(2+3)") (Add (Num 1) (Num 5)
    )
Just 1+(5)
*BiYacc> put pExpArith (parseExp "1+(5)") (Num 6)
Just 6
```

This means that if we have an evaluator on the abstract syntax, we will automatically get an evaluator on the concrete syntax!

A reflective printer can also be used as an ordinary printer by setting the original source to an empty one. For example:

```
*BiYacc> put pExpArith ENull (Sub (Num 0) (Add (Num 1) (Num 1)))
Just 0-(1+1)
```

Note that the subtraction is reflected as a binary minus instead of a unary one, despite that the left operand is zero. This behavior is easily customizable: By adding an adaptive branch before the one dealing generically with *Sub* in *pExpArith*:

$$\begin{aligned}&{}^{\$}(\mathit{adaptiveSV}\ {}^{\mathbb{P}}[\![\,\_\,]\!]\ {}^{\mathbb{P}}[\![\,\mathit{Sub}\ (\mathit{Num}\ 0)\ \_\,]\!])\\ &\quad \Longrightarrow \lambda\_\ \_ \rightarrow \mathit{EF}\ \mathit{FNull}\end{aligned}$$

the above abstract syntax tree can be printed as:

```
*BiYacc> put pExpArith ENull (Sub (Num 0) (Add (Num 1) (Num 1)))
Just 0-(1+1)
```

8.5 A Domain-Specific Language

As a final remark, the above programs may look long, but at the core of them are merely the correspondences between concrete production rules and abstract constructors. We can design a domain-specific language (DSL) that expresses such correspondences concisely, and then expand programs in this DSL into BiGUL. In fact, we have already done so, and the DSL is called *BiYacc*. For example, all the programs we have written can be generated from the following eight-line BiYacc program:

```
Arith +> Exp
Add l r +> (l +> Exp) '+' (r +> Factor);
Sub l r +> (l +> Exp) '-' (r +> Factor);
f       +> (f +> Factor);

Arith +> Factor
Num n         +> (n +> Int);
Sub (Num 0) r +> '-' (r +> Factor);
f             +> '(' (f +> Exp) ')';
```

See our SLE 2016 paper [18] for more interesting experiments about reflective printing, done on a more realistic imperative language.

9 Conclusion

We have given an introduction to BiGUL programming, explained the underlying design of its putback-based language constructs, and presented a number of applications. BiGUL in its current form is merely one step toward a versatile bidirectional programming language, though. We conclude this chapter by laying out some future directions.

While BiGUL is designed to ensure that programmers can freely describe whatever consistency restoration strategies they have in mind and guarantees that the described strategies are well-behaved, well-behavedness guarantees may be trivial if a described strategy is not actually well-behaved and consequently fails some dynamic checks at runtime. Working with the current BiGUL can thus involve a lot of tedious testing to see if those dynamic checks can go through; also, since we must keep the dynamic checks in place to ensure well-behavedness, at runtime they can incur serious performance overheads. We need ways to precisely characterize the behavior of the dynamic checks, so that it is possible to know that they are redundant and can be safely skipped during execution.

Also we have observed that, as consistency relations or consistency restoration strategies become more complex, BiGUL programs can quickly become awkward to write and hard to read. It is also not that easy to develop reusable libraries because BiGUL programs are not easily composable. (The only general composition operator, namely the classical lens composition, behaves obscurely in the putback direction and is difficult to understand in practice. A discussion of this problem is offered by, e.g., Diskin et al. [4, Sect. 2.2].) We need to design new language constructs that improve composability of BiGUL programs, discover programming patterns and architectures, and eventually build reusable libraries to facilitate program development.

Apart from language-specific issues, there are also challenges faced by the functional programming approach to BXs in general. For one, graphs have always been a kind of data structure that is hard to deal with in functional programming, but the application domains of other BX sub-communities usually require the ability to work with graphs; this is an area where BiGUL and other functional programming–based languages/tools need to catch up. More fundamentally, while programmable from one single direction, asymmetric lenses are a less expressive BX formalism, and we probably should not restrict the future version of BiGUL and new bidirectional languages to the framework of asymmetric lenses. We should recognize that the essence of BiGUL is its full programmability of bidirectional behavior, not the framework of asymmetric lenses which it currently supports, and we should strive to bring this programmability into other existing BX formalisms, or, if that is difficult, come up with new formalisms that are designed with such programmability in mind.

Acknowledgements. We would like to thank James Cheney, Jeremy Gibbons, and, in particular, Anthony Anjorin for their meticulous and helpful comments on this paper. This work is supported by JSPS Grant-in-Aid for Scientific Research (A) No. 25240009 of Japan, the Nation Basic Research Program (973 Program) of China under Grant No. 2015CB352201, and the National Natural Science Foundation of China under Grant No. 61620106007.

References

1. Barbosa, D.M.J., Cretin, J., Foster, J.N., Greenberg, M., Pierce, B.C.: Matching lenses: alignment and view update. In: International Conference on Functional Programming, pp. 193–204. ACM (2010). https://doi.org/10.1145/1863543.1863572
2. Bohannon, A., Foster, J.N., Pierce, B.C., Pilkiewicz, A., Schmitt, A.: Boomerang: resourceful lenses for string data. In: Symposium on Principles of Programming Languages, pp. 407–419. ACM (2008). https://doi.org/10.1145/1328438.1328487
3. Bohannon, A., Pierce, B.C., Vaughan, J.A.: Relational lenses: a language for updatable views. In: Symposium on Principles of Database Systems, pp. 338–347. ACM (2006). https://doi.org/10.1145/1142351.1142399
4. Diskin, Z., Xiong, Y., Czarnecki, K.: From state- to delta-based bidirectional model transformations: the asymmetric case. J. Object Technol. **10**(6), 6:1–6:25 (2011). https://doi.org/10.5381/jot.2011.10.1.a6
5. Fischer, S., Hu, Z., Pacheco, H.: A clear picture of lens laws. In: Hinze, R., Voigtländer, J. (eds.) MPC 2015. LNCS, vol. 9129, pp. 215–223. Springer, Cham (2015). https://doi.org/10.1007/978-3-319-19797-5_10
6. Fischer, S., Hu, Z., Pacheco, H.: The essence of bidirectional programming. Sci. China Inf. Sci. **58**(5), 1–21 (2015). https://doi.org/10.1007/s11432-015-5316-8
7. Foster, J.: Bidirectional programming languages. Ph.D. thesis, University of Pennsylvania, December 2009
8. Foster, J.N., Greenwald, M.B., Moore, J.T., Pierce, B.C., Schmitt, A.: Combinators for bidirectional tree transformations: a linguistic approach to the view-update problem. ACM Trans. Program. Lang. Syst. **29**(3), 17 (2007). https://doi.org/10.1145/1232420.1232424
9. Hidaka, S., Hu, Z., Inaba, K., Kato, H., Matsuda, K., Nakano, K.: Bidirectionalizing graph transformations. In: International Conference on Functional Programming, pp. 205–216. ACM (2010). https://doi.org/10.1145/1932681.1863573

10. Hu, Z., Pacheco, H., Fischer, S.: Validity checking of putback transformations in bidirectional programming. In: Jones, C., Pihlajasaari, P., Sun, J. (eds.) FM 2014. LNCS, vol. 8442, pp. 1–15. Springer, Cham (2014). https://doi.org/10.1007/978-3-319-06410-9_1
11. Ko, H.S., Zan, T., Hu, Z.: BiGUL: a formally verified core language for putback-based bidirectional programming. In: Workshop on Partial Evaluation and Program Manipulation, pp. 61–72. ACM (2016). https://doi.org/10.1145/2847538.2847544
12. Matsuda, K., Hu, Z., Nakano, K., Hamana, M., Takeichi, M.: Bidirectionalization transformation based on automatic derivation of view complement functions. In: International Conference on Functional Programming, pp. 47–58. ACM (2007). https://doi.org/10.1145/1291220.1291162
13. Pacheco, H., Hu, Z., Fischer, S.: Monadic combinators for "putback" style bidirectional programming. In: Workshop on Partial Evaluation and Program Manipulation, pp. 39–50. ACM (2014). https://doi.org/10.1145/2543728.2543737
14. Pacheco, H., Zan, T., Hu, Z.: BiFluX: a bidirectional functional update language for XML. In: International Symposium on Principles and Practice of Declarative Programming, pp. 147–158 (2014). https://doi.org/10.1145/2643135.2643141
15. Voigtländer, J.: Bidirectionalization for free! In: Symposium on Principles of Programming Languages, pp. 165–176. ACM (2009). https://doi.org/10.1145/1480881.1480904
16. Xiong, Y., Liu, D., Hu, Z., Zhao, H., Takeichi, M., Mei, H.: Towards automatic model synchronization from model transformations. In: International Conference on Automated Software Engineering, pp. 164–173. ACM (2007). https://doi.org/10.1145/1321631.1321657
17. Zan, T., Liu, L., Ko, H.S., Hu, Z.: Brul: a putback-based bidirectional transformation library for updatable views. In: International Workshop on Bidirectional Transformations, pp. 77–89. CEUR-WS (2016). http://ceur-ws.org/Vol-1571/paper_3.pdf
18. Zhu, Z., Zhang, Y., Ko, H.S., Martins, P., Saraiva, J., Hu, Z.: Parsing and reflective printing, bidirectionally. In: International Conference on Software Language Engineering, pp. 2–14. ACM (2016). https://doi.org/10.1145/2997364.2997369

Engineering Bidirectional Transformations

Richard F. Paige[✉]

Department of Computer Science, University of York, York, UK
richard.paige@york.ac.uk

Abstract. Bidirectional transformations, like software, need to be carefully engineered in order to provide guarantees about their correctness, completeness, acceptability and usability. This paper summarises a collection of lectures pertaining to engineering bidirectional transformations using Model-Driven Engineering techniques and technologies. It focuses on stages of a typical engineering lifecycle, starting with requirements and progressing to implementation and verification. It summarises Model-Driven Engineering approaches to capturing requirements, architectures and designs for bidirectional transformations, and suggests an approach for verification as well. It concludes by describing some challenges for future research into engineering bidirectional transformations.

1 Introduction

This paper constitutes the notes for a set of lectures on a collection of techniques and tools that can be used for engineering bidirectional transformations (BX). The motivation for these lectures is our view that transformations in general – and BX in the specific – are like other software systems: they are designed to be executed on a machine, are complicated (they involve many components that interact in a variety of ways), are in some cases complex (they exhibit behaviour that cannot be directly predicted from the behaviour of the individual parts), and are difficult to build correctly. As such, like software, transformations should be engineered by following a rigorous process. The advantages of doing so are the same as for software, including:

- *Repeatability:* by following a process, we potentially make it easier for others to repeat our work, or to reduce the amount of effort required to build a similar system in the future.
- *Review and Scale:* by decomposing a large engineering problem into stages, we potentially make it easier to audit and validate the results of each stage, and to solve larger problems than we would be able to if we treated the problem monolithically.
- *Automation:* by following a process we have greater opportunities to automate parts of it, e.g., generation of code or documents.
- *Training:* by following and documenting a rigorous engineering process we may make it easier to train others.

https://doi.org/10.1007/978-3-319-79108-1_5

BX are special kinds of transformations with, in our opinion, complicated execution semantics. As such, BX may especially benefit from following a repeatable, reviewable, scalable, automated process with training/guidance, for their development.

1.1 BX as Software

The assumption that we are making in the preceding is that BX are software systems. A software system is an executable artefact: given a specification of software (e.g., in a programming language or suitable modelling language), its expected outputs can be produced by executing the specification on a suitable machine (e.g., a server, a virtual machine, a simulator). A BX is an executable artefact: assuming that the BX is expressed in a suitable programming language or modelling language (and we review some of the key state of the art in Sect. 2) then its expected outputs can be produced by executing the BX on a suitable machine.

Like software, BX must satisfy functional and non-functional requirements, can (and probably should) be designed, and can exhibit unacceptable behaviour – that is, BX can contain faults, which may lead to failures. As we will see, depending on the technologies used to represent and specify BX, different types of failures may arise (e.g., inconsistencies) and different techniques may be used to verify the BX to help ensure that faults are caught during engineering. As we become increasingly ambitious in our attempts to solve complex problems using BX, our need for rigorous engineering techniques for BX construction will only increase.

1.2 Scope

There are numerous techniques and approaches that can be used to build and engineer BX; in Sect. 2 we will consider some of these. However, the focus of this paper will be on Model-Driven Engineering (MDE) techniques. Many of the techniques that we present in later sections can be used both with and without MDE tools, and if there are particular aspects that depend specifically on MDE, we will point these out where such a dependence isn't clear.

1.3 Background

Before we commence with the technical content of this paper, we provide some basic definitions and terminology, in order that the paper remain reasonably self-contained.

As mentioned, we are focusing on Model-Driven Engineering techniques for engineering BX. The key concepts of MDE are as follows.

– MDE involves the semi-automated construction and manipulation of *models*, which are structured, machine-implemented specifications of phenomena of interest. Models are meant to be processable by automated tools, and capture static and dynamic characteristics of systems.

- Models in MDE are *structured*; this structure can be defined in a number of ways, primarily via *metamodels*, which are specifications of abstract syntax (you can think of a metamodel as the definition of the abstract syntax of a language).
  A model is said to *conform* to a metamodel. Related approaches to defining the structure of models include schemas (e.g., XML), type rules and constraints. Many of these approaches define structure using graphs or graph-like concepts. As such, models themselves are often (but not exclusively) graphs. This is a key distinction between MDE (and so-called *modelware* approaches to engineering), and grammar-based (or *grammarware*) approaches.
- Models are typically specified alongside a set of *constraints* that capture well-formedness rules that cannot normally be specified with a metamodel. For example, a metamodel might be used to express that a model may include containers, and that containers may be nested (e.g., packages in UML). But a metamodel – which captures abstract syntax – will not normally express that containers have unique names. This can be expressed by a separate constraint, which is normally packaged up with the metamodel or models. If a model conforms to a metamodel, it must also normally be checked against any constraints, in order to establish that it is well formed.
- Standard technologies exist for capturing models, metamodels and constraints in the MDE world. The de facto standard technology used for metamodelling is Ecore (a part of the Eclipse Modelling Framework (EMF)). For constraints, engineers typically use the Object Constraint Language (OCL), which also has an official Eclipse implementation. There are other languages and technologies available as well for metamodelling and for expressing constraints.
- Models by themselves typically encapsulate business value, but are also meant to be processed by automated tools. These tools implement a variety of operations applicable to models, including the aforementioned transformations, but also comparisons, merging, migration, matching and others.

Transformations are a key operation in MDE, and have been the subject of widespread study (e.g., see recent proceedings of the long-running conference on model transformation [1]). Numerous classifications and surveys have been published on transformations in general, and BX in the specific. Four common categories of transformations in MDE are:

- *Unidirectional transformations*, from a source model to a target model. Such transformations are usually implemented in terms of metamodels, and are typically used when the source and target metamodel are linguistically similar, e.g., between different dialects of UML, or from an object-oriented model to a relational database model. Unidirectional transformations typically are written in one of three styles: purely declarative, operational, and hybrid (i.e., a mixture of operational and declaration parts). In our experience, many complicated transformations are very difficult to express in a purely declarative style. As such, hybrid transformation languages (such as ATL [2] and ETL [3]) tend to see the most use in industrial practice.

– *Update-in-place transformations*, which specify modifications made to one and only one model. Update-in-place transformations can be specified using languages suitable for unidirectional transformations, or specialist languages such as EWL [4].
– *Model-to-text* (sometimes called model-to-grammar) transformations, where the source/input to the transformation is a model, but the output no longer conforms to a metamodel, e.g., free-form text or text conforming to a grammar. Model-to-text transformations are used in order to step outside of the modelware technical space and move to the grammarware technical space. An example scenario for use of model-to-text transformation is code generation.
– Bidirectional transformations, which is the subject of the next section.

Transformations (and other operations on models) have side-effects. This includes purely declarative transformations. The side-effect in question is the production of *traceability* information, i.e., so-called *trace-links*, which relate source and target model elements. Trace-links can be generated automatically by transformation tools (such as Epsilon or ATL) and they can be stored for later audit and analysis. Trace-links are important in the context of transformations and BX as they provide (a) the basis for verification and validation of transformations; and (b) the connection to the theory behind BX, specifically delta lenses (in particular, delta lenses are a sound theory for trace models, encoded in an algebraic form [5]).

1.4 Structure

We start with a brief review of the state-of-the-art in engineering BX with MDE, focusing firstly on BX scenarios of use in MDE, followed by an overview of MDE languages, tools and techniques for supporting BX. The remainder of the paper considers different aspects of a BX engineering lifecycle, starting with an overview of techniques for requirements engineering for BX, focusing on requirements specification and requirements analysis. We then move to an overview of techniques for architecture and design of BX, including a small selection of relevant design patterns. Finally, we briefly consider one approach for verification of BX, which applies to a specific approach to BX implementation and design. The paper concludes with a discussion on future challenges and perspectives on engineering of BX.

2 State of the Art

This section addresses some of the important state of the art in MDE approaches to BX, focusing on three specific elements: important BX *scenarios* that have been identified in the literature; important *languages* that have been influential in research in BX – in this case, we focus on QVT; and important *tools* that implement aspects of BX and that are based on MDE technology. We do not consider non-MDE approaches to BX in this brief review, and we also exclude TGG approaches because these are covered in detail by Anjorin's chapter [6] in this volume.

2.1 BX Scenarios

A number of recurring scenarios of use for BX have appeared in the MDE literature. Many of the MDE tools and languages that we discuss in the sequel have been designed to address these scenarios.

1. *Round-trip engineering*, i.e., generating code from models, modifying the code by hand, and then regenerating the models to reflect changes made in the code. A BX approach would, conceptually, aim to apply the principle of least change and minimise the number of modifications necessary to the original model, instead of regenerating the entire model after each change. Research in MDE related to incremental transformation is also addressing this scenario.
2. *Collaborative modelling*, wherein multiple stakeholders are editing the same model simultaneously. In practice, what often happens is that each stakeholder has a local copy (or view) of the source model, and their changes are reflected back on the master/source copy at specified points of time.
3. *Synchronisation*, e.g., synchronising documents and code, like assurance cases and source code. This is related to round-trip engineering but synchronisation can involve model management operations other than transformations.
4. *Reflection*, for example, reflecting the results of some kind of analysis on a source model. A concrete instance of this was investigated in the MADES project[1] where a UML MARTE model was transformed into a variety of formal models (UPPAAL, TRIO) to support analysis, and some of the results of the analysis were reflected in the MARTE models. This is an interesting example of a BX as the backwards transformation is generating a view of the target model which needs to be synchronised with the source model.

2.2 Standard MDE Languages for BX: QVT

While there are tools and approaches, based on MDE technology (like Eclipse EMF) for supporting BX, most of these approaches are strongly influenced by a significant standardised language for transformation: the OMG's Query, View and Transformations (QVT) standard [7]. QVT is a family of languages that were first envisaged in 2002 upon issue of an OMG request for proposals to support aspects of the OMG's Model-Driven Architecture standard. A number of replies were received, and the first version was submitted and approved in 2005. The most recent version, QVT 1.3, was released in June 2016.

QVT, as mentioned, is a family of languages. These languages are meant to support transformation and querying of MOF models; transformations and queries can be used in turn to generate views. The basic architecture of QVT is illustrated in Fig. 1. The QVT architecture builds on other OMG languages, particularly MOF but also the Object Constraint Language (OCL), from which QVT acquires its expression and collection manipulation facilities.

The Relations language provides mechanisms for the declarative specification of the relationships between MOF models. It supports in turn complex object

[1] http://www.mades-project.org/.

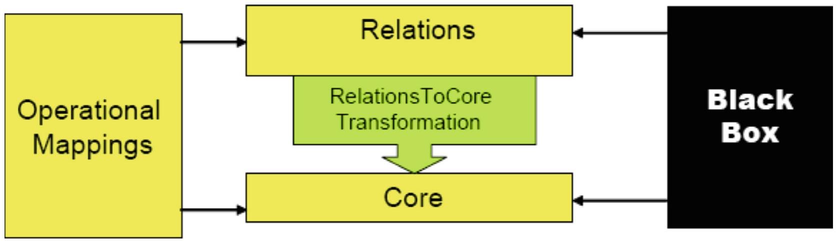


Fig. 1. QVT architecture [7]

pattern matching, and implicitly creates trace classes and their instances to record what occurred during the execution of a transformation. Assertions can also be made; for instance, relations can assert that other relations also hold between particular model elements matched by their patterns. As illustrated in the figure, the intention is that Relations specifications can be translated in to the QVT Core language, along with a set of trace models, which in total provide a formal semantics for QVT Relations. Though this is the intention of Relations, it has been shown – e.g., by Stevens [8] – that there are programs that can be expressed in Relations that cannot be translated to Core.

QVT Core, by contrast, is a small yet expressive language that only supports pattern matching over a flat set of variables by evaluating conditions over those variables against a set of models. It is intended to be semantically equivalent to QVT Relations, but equivalent QVT Relations programs are liable to be more concise than the QVT Core programs.

The Operational Mappings (sometimes called QVT Operations, or QVT-o) is an operational model transformation language that extends Relations with imperative constructs. Of all the QVT languages, it is QVT-o that has received the most use and attention.

The abstract syntax of the Relations language is illustrated in Fig. 2. The abstract syntax can be interpreted as follows: a QVT Relations program contains a set of rules which are relations. Relations are made up of patterns, and are applied to a set of typed model parameters. In particular, these relations can be interpreted in forward and backwards directions – that is, Relations is a BX language by design.

An example of the concrete syntax of Relations is shown in Listing 1.1. This example gives a relation that is part of the classic object-relational mapping, in this case used to map persistent classes in an object oriented program to a table. The example includes three parts: a domain (a set of patterns which defines the variables and constraints that model elements bound to those variables must satisfy – i.e., the bindings for the relation); the *when* clause (the conditions under which the relation must hold); and the *where* clause (the condition that must be satisfied by all model elements participating in the relation). The interpretation of when-clauses in the Eclipse QVT implementation is that these are preconditions, and where-clause are postconditions. Both of these clauses may contain arbitrary OCL expressions.

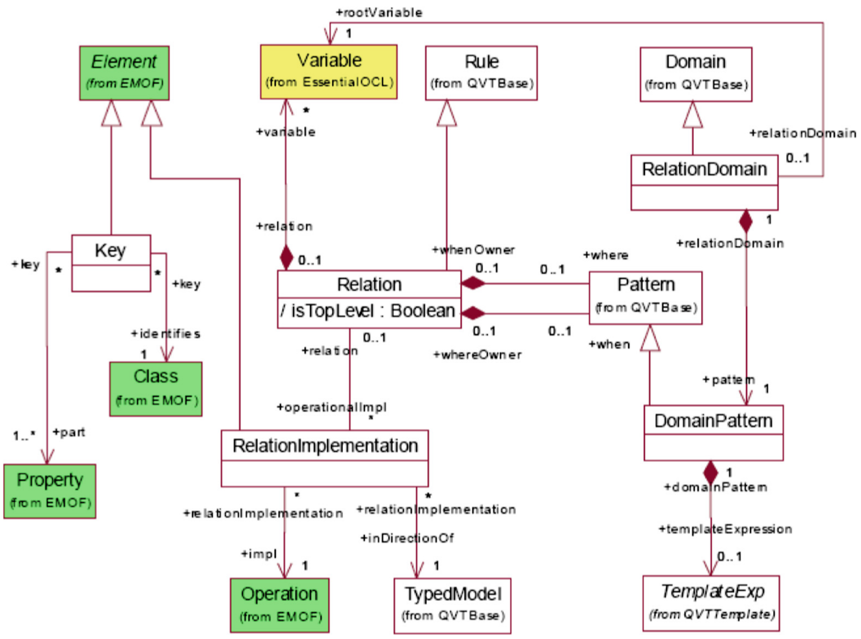


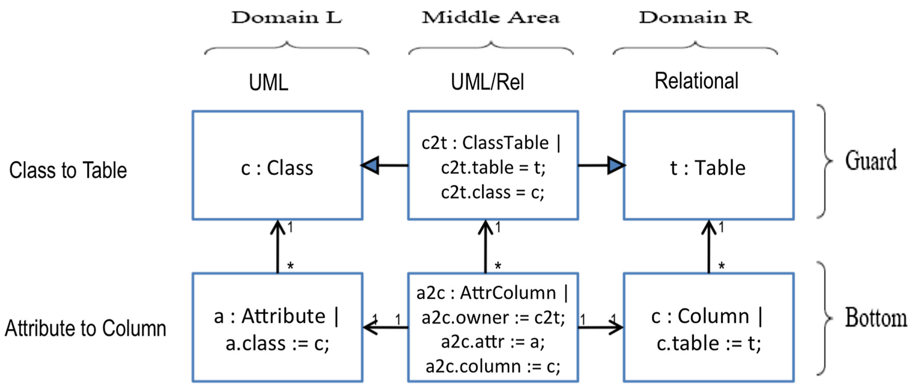
Fig. 2. QVT Relations abstract syntax

In this particular example, the domain clauses establish which model elements in a UML and a RDBMS model are of interest (they satisfy the predicate part of the domain clauses), and the when and where clauses are defined elsewhere by other relations.

The BX capabilities of QVT can also be illustrated by an example from QVT Core. Figure 3 illustrates an example of a single mapping rule in QVT Core. This is a *checking* example, which is used to check that particular patterns are satisfied by models. Once again, this is an example involving relations between a UML class model and a database model. The top part of the diagram (labelled Class to Table) defines the *c2t* relation, which relates a class to a table. The bottom pattern is evaluated using variable values of a valid binding (a valid pair of class and table) from the top pattern. In effect, the top part of the mapping rule defines a guard which restricts the scope of the bottom part of the rule.

```
relation ClassToTable /* map persistent class to table */
{
  domain uml c:Class {
      namespace = p:Package {},
      kind = 'Persistent',
      name = cn
  }
  domain rdbms t: Table {
      schema = s:Schema {},
      name = cn,
      column = cl:Column {
        name = cn + '_tid',
        type = 'NUMBER'},
        primaryKey = k:PrimaryKey {
          name = cn + '_pk',
          column = cl }
   }
   when {
     PackageToSchema(p,s);
   }
   where {
      AttributeToColumn(c,t);
    }
}
```

Listing 1.1. An example of QVT Relations



Domain L
Middle Area
Domain R
UML
UML/Rel
Relational
Class to Table
c : Class
c2t : ClassTable | c2t.table = t; c2t.class = c;
t : Table
Guard
Attribute to Column
a : Attribute | a.class := c;
a2c : AttrColumn | a2c.owner := c2t; a2c.attr := a; a2c.column := c;
c : Column | c.table := t;
Bottom
1
*

Fig. 3. QVT Core: mapping rule example

The mapping rule is directionless; it can be executed either way, i.e., checking a database table against a UML class, or checking a UML class against a database table.

The QVT standard is currently being further developed, both through the OMG standardisation efforts, but also through work on Eclipse QVT, an implementation of the different QVT languages. We briefly discuss the status of Eclipse QVT, and other MDE tools for BX, in the next subsection.

2.3 Tools

In this section we briefly outline some of the key tools, based on MDE technologies and principles, that either support or claim to support BX. As mentioned earlier, we exclude approaches based on triple graph grammars as these are covered elsewhere.

Medini. Medini[2] claims to be a reasonably complete implementation of QVT Relations, but is currently unsupported. It is an EMF based transformation engine but also has a non-commercial licensed editor and debugger. While it uses the QVT Relations syntax, it intentionally departs from the semantics of the OMG standard (e.g., how it supports deletion of elements, that it does not provide a checkonly mode). As such, we prefer not to label Medini as an implementation of QVT, but as a tool that is inspired by QVT.

ModelMorf. ModelMorf is a proprietary tool from Tata Consulting Services[3]. It also claims to faithfully implement the QVT Relations standard, but research by Stevens [8] shows that it does not fully implement the semantics specified in the standard. By some measures, it is more faithful than Medini, but it is still not a full implementation of QVT.

JQVT. jQVT[4] is a QVT-like engine that is defined on top of the Java type system instead of using EMF. In turn, it uses Xbase (a partial programming language written in Xtext which compiles to Java and includes powerful features such as closures) instead of OCL for expressions. In essence, jQVT is a Java embedding of QVT; the jQVT engine generates native Java code from jQVT scripts. Of note is that it does provide support for bidirectional transformations. As of early 2016 jQVT was still being maintained.

Echo. Echo[5] is an open-source EMF-based tool for model repair and transformation that exploits the Alloy model finder to determine models that satisfy relations. It provides an implementation of the QVT Relations syntax, but the semantics intentionally departs from the OMG specification. Echo is also bidirectional.

JTL. The Janus Transformation Language (JTL)[6] is a by-design bidirectional language with a QVT-like syntax, which propagates changes made in one model to the other. If a change made to one model makes the second model inconsistent, an approximation ("closest match") is calculated using answer set programming. As such, there can be several solutions to a transformation problem and the results provided by JTL may need to be constrained further.

Eclipse QVT. Substantial engineering effort is being put into the development of Eclipse QVT, a project that aims to support the full OMG QVT specification

[2] http://projects.ikv.de/qvt/wiki.
[3] Archived copy available at https://web.archive.org/web/20120323171429/http://www.tcs-trddc.com/trddc_website/ModelMorf/ModelMorf.htm.
[4] https://sourceforge.net/projects/jqvt/.
[5] http://haslab.github.io/echo/.
[6] http://jtl.di.univaq.it/.

(though with Ecore instead of MOF models). Currently, QVT Operations is well supported and active as part of the Eclipse M2M project. QVT Relations (in Eclipse terms, QVT Declarative) and QVT Core are work-in-progress. As work on these projects is ongoing and their status is changing regularly, we refer the reader to the Eclipse MMT project website[7] for the latest information. As of this writing, the intention with Eclipse QVT is that the Oxygen release in June 2017 will provide full support for QVT Relations.

Bidirectionalisation. There have been several approaches to so-called *bidirectionalisation* of transformations. In these approaches, a forward transformation (from source to target) is written and the backward transformation is calculated or computed automatically. Examples of this approach include that of Hoisl [9]. The GRoundTram approach of Sasano [10] is another example.

For further details, and a more in-depth classification of MDE approaches to BX, the interested reader is referred to Hidaka et al.'s excellent survey of BX [11].

3 Requirements Engineering for BX

In this section we will consider techniques and tools for requirements engineering for BX. We will motivate the benefits of considering requirements for BX in general, before discussing some of the general questions to be addressed when building a BX. These questions will help us motivate a discussion on the general properties of BX (which may be the source of constraints on requirements for a BX), as well as examples of functional and non-functional requirements for BX. This is followed by a broad overview of requirements engineering processes for BX, which leads in to a discussion on MDE languages suitable for requirements engineering for BX.

3.1 Motivation

Requirements engineering is the process of identifying, documenting and maintaining requirements in systems engineering. The typical tasks involved in requirements engineering are:

– *identification:* where new requirements to address a problem are clarified
– *analysis:* where the requirements are assessed to ensure they accurately capture what is needed for the system under consideration, and conflicts between stakeholders are resolved
– *specification:* where the requirements are documented in a precise (but not necessarily formal) way
– *validation:* where the requirements are checked to ensure they are consistent and address stakeholder needs

[7] https://projects.eclipse.org/projects/modeling.mmt.

– *maintenance:* where the requirements are considered for update as the system under consideration is constructed, deployed and changed.

BX are software systems and as such will benefit from a clear understanding of requirements; for large or complicated BX, there may be benefits to following a rigorous requirements engineering process as well. In particular, an understanding of requirements for BX can help in mapping BX problems to tools that are suitable for implementation (and vice versa). An understanding of requirements for BX can also help in contrasting different potential solutions in terms of their tradeoffs in how they satisfy requirements.

3.2 Questions and Properties for BX

A typical first phase of requirements engineering is *identification*, where engineers attempt to determine what requirements a software system should exhibit. This in turn may help determine properties or constraints that the ultimate system will satisfy. There are numerous ways in which requirements can be identified, e.g., via stakeholder interview, by reviewing existing similar systems, by following questionnaires or checklists, or by using testing techniques to derive requirements. Based on Tehrani et al.'s work [12], we suggest some general questions that could be addressed when constructing a BX, the answers to which could help derive requirements.

1. What needs to be transformed into what? Alternatively – and declaratively – what kind of consistency needs to be maintained?
2. What mechanisms can be used for building the BX? (i.e., theory, tools, techniques)
3. What are the application domains for the BX?
4. What are the specific characteristics of the BX (e.g, what patterns are appropriate to use)?
5. What are the quality requirements (e.g., performance) for the BX?
6. What are the success criteria for the BX?

Questions 3, 4 and 6 are possibly the most opaque. Question 3 is designed to help identify constraints on the scope of use for the BX, e.g., will the BX be used in developing hard real-time systems, or interactive systems? Question 4 is designed to help identify functional requirements, e.g., should the BX be parameterised, should it be interactive? This in turn may help identify suitable patterns that can be used in specifying or designing the BX. Question 6 is the "stopping condition": how will we know if we have successfully solved the BX problem?

BX exhibit various properties (such as least-change, or determinism). When considering requirements for a BX, there are general properties that may be of interest, particularly in determining constraints that the ultimate BX must satisfy. Some examples are:

– Size: is the BX small (e.g., a single reversible refactoring) or large (e.g., a reversible code generator)?

- Level of automation: is the BX meant to be fully automated, or involve a human-in-the-loop?
- Visualisation: how is the BX, the results of executing the BX, and the input to the BX presented to users?
- Level of industry application: to what extent is the BX to be deployed in an industrial context?
- Maturity level: should the BX be implemented in a tool? Should the BX be a theoretical construct?

Understanding the relative importance of these properties will be helpful in deciding on what theory or tool to choose for defining a BX.

3.3 Functional and Non-functional Requirements

In the classical requirements engineering literature, functional requirements specify what a system must, could or should provide. Non-functional (or behavioural) requirements specify criteria against which we can judge the quality of a system. In a requirements document, functional and non-functional requirements are typically presented separately, with suitable tests given that can be used to assess the coverage and completeness of fulfilment of requirements.

There has been little published research on examples of requirements for transformations in general, let alone BX, but based on some of [12,13] we can propose some examples for BX. We start with functional requirements. For simplicity of presentation, we assume that a BX under development is defined between two models (a source and a target).

- *Correctness:* a BX that is correct will restore consistency between inconsistent models after its execution. Operationally, when the BX is run in the forward direction, the target model must be well formed (defined in terms of conformance to the target metamodel and any corresponding constraints). Similarly, when the BX is run in the reverse direction, the source model must be well formed. It is interesting to observe that the terminology used in the BX community for correctness differs from that used in the requirements engineering community.
- *Inconsistency tolerance:* the BX should be able to support incomplete or inconsistent models, e.g., temporarily inconsistent models. This reflects the practical situation wherein a BX *gradually* re-establishes consistency over a sequence of steps.
- *Modularity:* it should be possible to compose BX into new transformations.
- *Traceability:* a BX should support the generation of trace-links (sometimes called a correspondence model) between source and target models, as well as between the steps of a transformation chain.
- *Change propagation:* a BX should provide support for propagating changes from one model to the other model.
- *Incrementality:* a BX should make it possible to update a model based only on the changes made to the other model (that is, the parts of the model that do not change are not used to make changes to the other model).

– *Uniqueness:* a BX could support the ability to generate a unique solution to the problem of ensuring consistency between two models.
– *Termination:* it should be possible to support the definition of terminating BX transformation executions.
– *Style:* a BX should be expressible in a particular style, i.e., declarative, operational or hybrid.

Note the wording of these requirements; we have used the words *must*, *should* and *could* to indicate the degree of importance or criticality of each type of requirement. As this suggests – and as is reinforced by [11] – there is substantial variability in what BX provide (and also how they are implemented).

Non-functional requirements, recall, specify criteria against which we can judge the quality of a BX. As is the case for functional requirements for BX, there is limited research on non-functional requirements. Some examples have been proposed by [13], and we list a selection here.

– *Extensibility*: the extent to which the BX can be extended to support new functional requirements or a change in scope.
– *Usability*: is the BX judged to be usable by stakeholders?
– *Robustness*: can the BX manage invalid models (i.e., that do not conform to the metamodels involved in the BX), or deal with errors in models?
– *Interoperability*: can the BX be combined and used together with non-BX tools (e.g., other MDE tools and operations, such as model comparisons or mergings)?

Clearly, more research on requirements for BX is needed. As our experience with building BX grows, and our understanding of what constitutes a useful BX scenario increases, our ability to elaborate sensible functional and non-functional requirements for BX will improve.

3.4 Requirements Engineering Processes for BX

In this section we outline typical stages of a requirements engineering process for BX and highlight the key artefacts and stakeholders that will be involved. We discuss elicitation in some detail, and evaluation briefly. This leads in to the next section where we give an overview of some of the key specification techniques that can be used within a requirements engineering process for BX.

Typical requirements engineering literature [14], identifies the following generic phases in requirements engineering:

– *Domain analysis and elicitation:* Identify who are your stakeholders. From these stakeholders, gather information on the system domain and system requirements.
– *Evaluation and negotiation:* Identify imprecision, conflicts, omissions and redundancies in the informal requirements identified in the previous phase. Resolve these (if possible and appropriate) via negotiation and consultation.

– *Specification:* Document the formal requirements in a specification (we will consider this for BX in more detail later). The specification is often the basis for a contract between developers and customers.
– *Validation and Verification:* Check the specification for consistency, completeness and acceptability to stakeholders.

This is generic, applicable to any kind of software or systems engineering. What might a requirements engineering process for BX look like? Tehrani et al. [12] propose a process for transformations, which is depicted in Fig. 4.

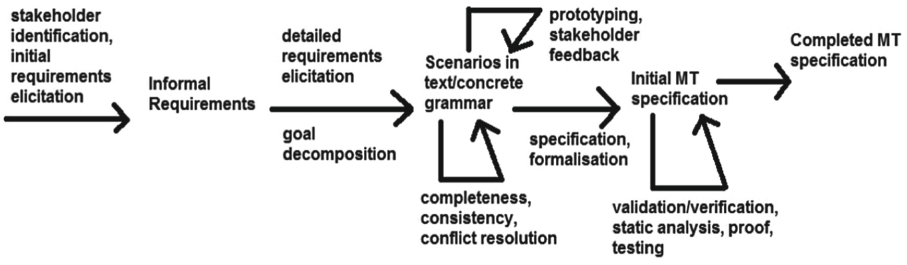

stakeholder identification, initial requirements elicitation
Informal Requirements
detailed requirements elicitation
goal decomposition
Scenarios in text/concrete grammar
prototyping, stakeholder feedback
completeness, consistency, conflict resolution
specification, formalisation
Initial MT specification
validation/verification, static analysis, proof, testing
Completed MT specification

Fig. 4. A transformation requirements engineering process [12]

(It is worth emphasising that the process shown in Fig. 4 is for transformations in general, not specifically for BX.) There are some points to note about the above process.

– The process is generic for the most part, and resembles the steps that are typically carried out for software systems.
– An interesting aspect is the use of scenarios as a concrete mechanism for driving the development of a requirements specification. In the context of BX this suggests that identifying and capturing more (and more detailed) BX scenarios will be very helpful in improving our understanding of BX requirements engineering.
– The process distinguishes between local and global requirements, as is often done in systems engineering. A local requirement may pertain to a particular transformation component (e.g., that correspondences are defined between elements of particular types), whereas a global requirement may apply to an entire transformation (e.g., a performance requirement, that a measure of complexity is reduced by running a BX, or a safety requirement).

3.5 Elicitation

Elicitation is an important first step in any requirements engineering process. What techniques might be applicable for BX? Many of the traditional elicitation techniques appear to be directly applicable to BX problems with little change, as argued by Tehrani et al. [12]. For example, a classic elicitation technique is observation (an ethnographic method): observing an existing – possibly manual – BX technique or process could provide sensible requirements for an automated process. Consider a scenario wherein a BX is to be defined between an Excel spreadsheet and a SysML requirements diagram[8]. A manual BX process between the two might involve (a) making changes to cells in an Excel column; (b) switching to a SysML editor; and (c) modifying attributes in a SysML class model. This might indicate to a requirements engineer that there is a sequence of steps that should be implemented in a BX.

Another technique that can be used for elicitation is the *unstructured interview*, where open-ended questions are asked about the problem domain or the current (BX) process. This can be useful for identifying transformation goals, e.g., "ensure that the source and target models are inconsistent for no more than 10 ms". In carrying out an unstructured interview regarding a transformation, Tehrani [12] suggests some generic open-ended questions that may be useful to consider; we have extended their questions with some of our own, based on our experience in the MONDO project[9].

– Is there a size range for the source and target models? This may suggest to the engineer the type of infrastructure that may be useful for the project (e.g., EMF to represent models).
– Does the encoding for the BX matter? For example, for very large scale models it may be necessary to consider binary formats.
– Are there any assumptions that are made about the source or target models? For example, are they always available? Are they read-only? Write-only? Are there confidentiality restrictions?

Along with unstructured interviews there are *structured* interviews, which involve asking pre-selected questions about the domain and the BX, perhaps based around a checklist linked to a requirements pattern catalogue. For example, a checklist of questions may be divided into parts, one focusing on questions related to global functional requirements (e.g., is hippocraticness important, is semantics preservation important?) and another related to local non-functional requirements (e.g., should this rule satisfy a specific time bound?).

A final elicitation technique that we mention is *scenario-based analysis*, where scenarios are used to capture different requirements transformation processing cases. The benefit of using scenarios is that they are concrete: scenarios are usually presented in a concrete scenario language, often supplemented with sketches of sample models. For example, for BX we might specify a scenario for introducing or removing a pattern to change an object-oriented design. The forward transformation scenario could include a concrete example of introducing the pattern into an existing design.

[8] This is a sanitised version of a real problem encountered by the author.
[9] http://www.mondo-project.org/.

3.6 Evaluation

Once we have elicited requirements for BX through any of the techniques described previously, we have a set of informal statements of what the BX must or should provide. These statements may be inconsistent, and ideally we should be identify this before we formalise the BX requirements in a specification. There is little to no published research on evaluation techniques for BX requirements. We may find some inspiration in the general requirements engineering literature. For example, one approach used for requirements evaluation is prototyping, i.e., engineers build a prototype (paper, mock-up, simulation) of a solution in order to help identify or reconcile inconsistencies. It is unclear whether the expense of building a BX prototype is less than building a BX in the first place (because, for example, a BX prototype could be constructed using standard BX tools, or could be constructed as a paper prototype). Another approach that is sometimes used is goal-oriented analysis, but it is as of yet unclear how goal-oriented techniques apply to the definition of BX. There are significant open questions relating to how we evaluate requirements for BX.

3.7 MDE Languages for Requirements Engineering for BX

In this section we move from a mostly abstract discussion on requirements engineering for BX and focus on the more concrete topic of languages that can be used to support requirements engineering for BX. There has been some work in this area – i.e., on different MDE languages and tools for specifying transformation requirements – though there is still very limited experience of specifying requirements for BX in the specific. Here, we will focus on presenting details of one approach – *trans*ML – which is a family of languages that can be used for engineering model transformations. *trans*ML can, as we will show, be used to specify different aspects of the requirements for a BX. We will also use *trans*ML in the next section to specify different facets of the architecture and design of a BX. For an alternative approach to specifying requirements for transformations, based on mind-maps, the interested reader is referred to the DSL-Maps approach [15].

*trans*ML [16], by way of introduction, is a family of MDE languages to support the lifecycle of transformation development, from requirements through to implementation. It is technology agnostic, and can be used with any transformation implementation language (there is published experience of using *trans*ML with QVT, EOL, ETL and ATL [16]). The overall architecture of *trans*ML – that is, the set of languages and their inter-relationships – is depicted in Fig. 5. The parts of *trans*ML relevant to this section are the Requirements language (at the top) and the languages to support Analysis (Simple Scenarios and Formal Specification).

We focus on the requirements language and those languages of *trans*ML that support analysis in this section. The former is used primarily to support the description of the results of elicitation. The latter are used to support detailed specification.

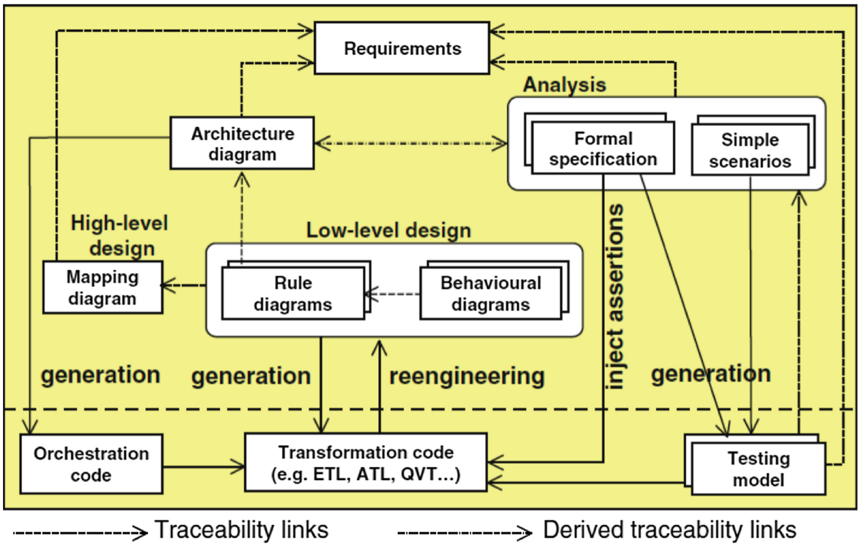


Fig. 5. *trans*ML architecture; boxes represent languages (or sets of languages) and arrows represent dependencies, typically traceability links [16]

To support description of the results of elicitation, *trans*ML provides a diagrammatic representation of (BX) requirements that is derived from SysML requirements diagrams. Such representations can be produced using any of the aforementioned techniques for elicitation. Because *trans*ML is an MDE language, it is defined using metamodels. The *trans*ML requirements metamodel is shown in Fig. 6.

The requirements metamodel is very simple, but defines an expressive requirements language for BX. The language explicitly supports hierarchical decomposition of requirements, as well as classification, refinement, and traceability. Of particular note is the *ReqSource* element, which identifies where a requirement arises, i.e., in the source of a transformation, the target of a transformation, or from the transformation itself (it is generated by the transformation).

We illustrate the requirements metamodel with two examples, the first from Guerra et al. [16] which shows an example requirements model for a unidirectional transformation (Fig. 7), and the second which shows an example for a BX (Fig. 8). In both cases, the examples involve transformations from and between object-oriented and database models. We observe that different concrete syntaxes are used in each example. The first concrete syntax is based on SysML, whereas the second is a box-and-arrow domain-specific requirements language which makes use of elements of UML (particularly dependencies and stereotypes).

The top-level requirement (OO2DB Transformation) in Fig. 7 is decomposed into the set of requirements below (i.e., No Redefined Attributes, Classes,

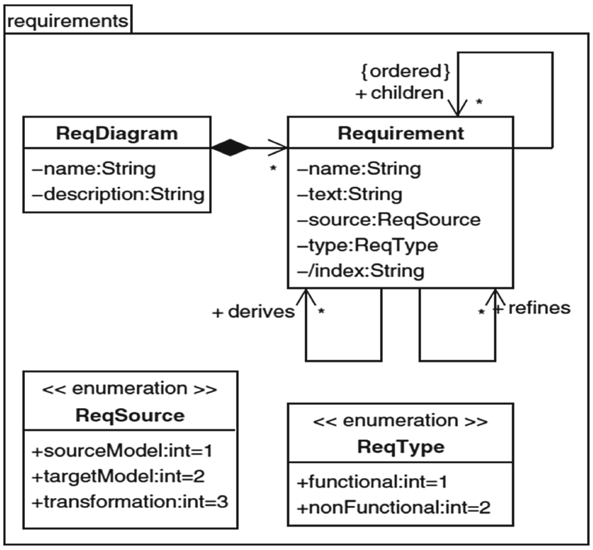


Fig. 6. *trans*ML requirements metamodel [16]

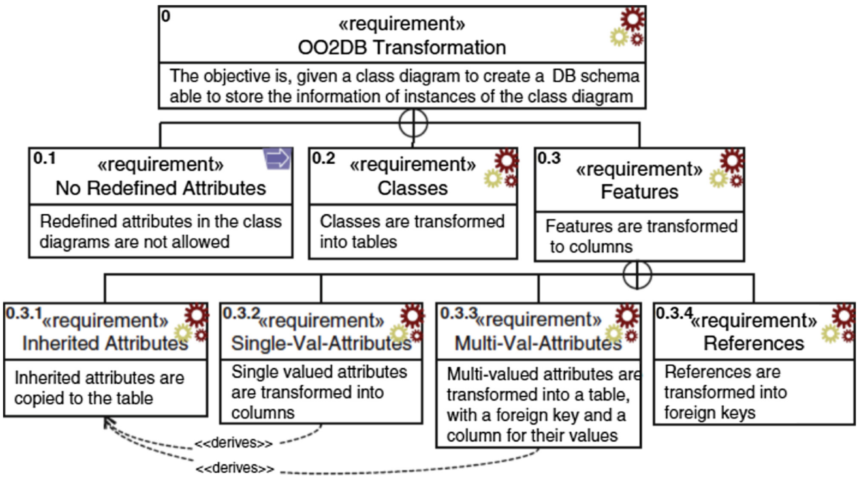

0 «requirement» OO2DB Transformation
The objective is, given a class diagram to create a DB schema able to store the information of instances of the class diagram
0.1 «requirement» No Redefined Attributes
Redefined attributes in the class diagrams are not allowed
0.2 «requirement» Classes
Classes are transformed into tables
0.3 «requirement» Features
Features are transformed to columns
0.3.1 «requirement» Inherited Attributes
Inherited attributes are copied to the table
0.3.2 «requirement» Single-Val-Attributes
Single valued attributes are transformed into columns
0.3.3 «requirement» Multi-Val-Attributes
Multi-valued attributes are transformed into a table, with a foreign key and a column for their values
0.3.4 «requirement» References
References are transformed into foreign keys
<<derives>>
<<derives>>

Fig. 7. *trans*ML requirements model example (SysML-like concrete syntax) [16]

Features). The Features requirement is further decomposed in the last level of the diagram. Note that derived requirements are also noted, i.e., that the Inherited Attributes requirement is derived from the Single-Val-Attributes and Multi-Val-Attributes requirements.

The example in Fig. 8 illustrates a requirements specification for a BX. It has a similar structure to the previous example for a unidirectional transformation.

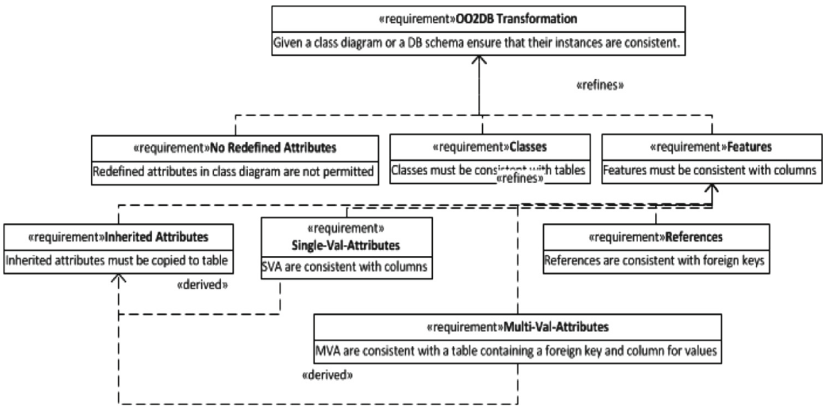

«requirement»OO2DB Transformation
Given a class diagram or a DB schema ensure that their instances are consistent.
«refines»
«requirement»No Redefined Attributes
Redefined attributes in class diagram are not permitted
«requirement»Classes
Classes must be consistent with tables
«requirement»Features
Features must be consistent with columns
«refines»
«requirement»Inherited Attributes
Inherited attributes must be copied to table
«requirement» Single-Val-Attributes
SVA are consistent with columns
«requirement»References
References are consistent with foreign keys
«derived»
«requirement»Multi-Val-Attributes
MVA are consistent with a table containing a foreign key and column for values
«derived»

Fig. 8. *trans*ML requirements model example (box-and-arrow concrete syntax)

The main difference is in the expression of the individual requirements, which are expressed in terms of consistency relationships rather than transformation features.

Both of these examples are informal, in the sense that they rely substantially on natural language, and are the result of applying elicitation techniques; they may contain imprecision or inconsistencies, which may be resolved by analysis. *trans*ML supports two sets of languages for requirements analysis: a simple scenario language, and a formal specification language for requirements.

The simple scenario language of *trans*ML supports description of *concrete cases* for transformation, i.e., how examples are meant to be related by the BX. *trans*ML is applicable to both models or fragments of models, the latter of which is essential for incremental development and for working with large monolithic models. An example of a transformation case (i.e., a scenario) for part of an object-oriented to database BX is shown in Fig. 9.

On the left side of the example is the object-oriented model fragment, consisting of a class with a multi-valued attribute; on the right side is a database model fragment, consisting of two tables containing columns and foreign keys. This is an example of a BX scenario involving a class with a multi-valued attribute and a consistent database model that resolves the multi-valued aspect using a foreign key (there are other solutions).

The second *trans*ML language for requirements analysis supports formal specification of requirements; it is used to specify what a transformation has to do. It captures correctness properties and specifies restrictions on the models involved in the BX (for example, the consistency relations specified in the BX may only be applicable when the source or target models obey various constraints). The *trans*ML formal specification language supports all of this via use of declarative patterns, a concept taken from triple graph grammars. Patterns express allowed and permitted relations between elements from the involved models. The pattern language itself is expressive and can include conditions on attribute values as well as constraints.

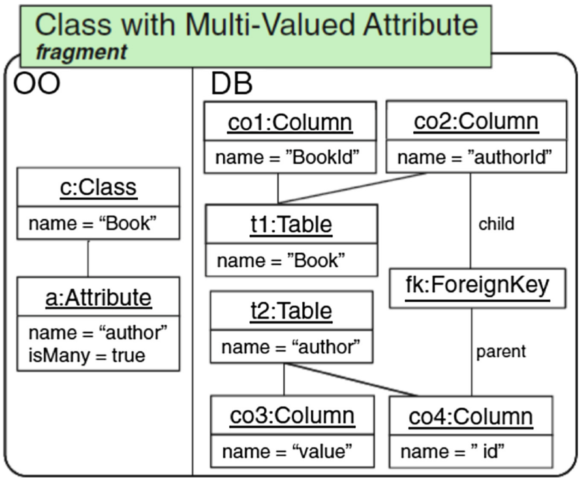


Fig. 9. *trans*ML scenario (example case) [16]

The metamodel for the *trans*ML formal specification language for requirements is depicted in Fig. 10.

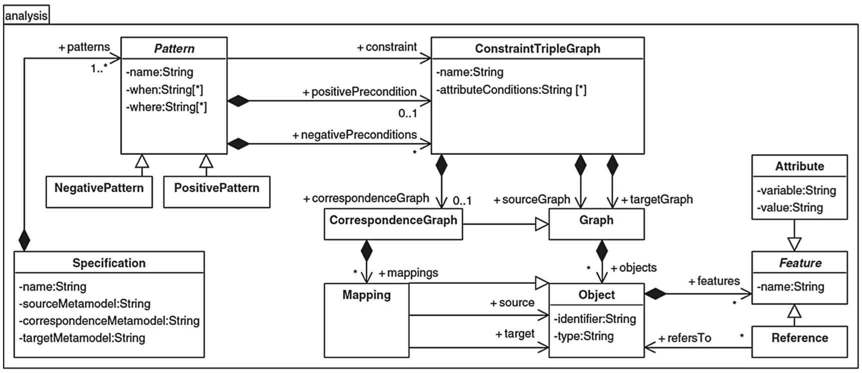


Fig. 10. *trans*ML formal specification language metamodel [16]

A requirements specification (the Specification element in Fig. 10) is made up of a number of patterns. A pattern may be a positive or a negative precondition, which are similar to both the *when*-clauses of QVT Relations, as well as triple graph grammar's negative application conditions. The Constraint Triple Graph element encodes these clauses, and also include correspondence graphs (which is effectively traceability information) as well as links to source and target graphs.

An example of a pattern for a BX is shown in Fig. 11.

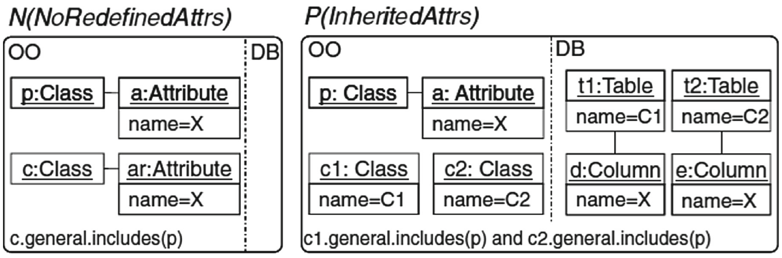


Fig. 11. *trans*ML example pattern [16]

The example pattern is, once again, taken from the object-oriented to database BX example that we have used several times before. In this example, the left side of the diagram is a negative pattern: it checks for the existence of two classes c and p such that p is an ancestor or c, while both have an attribute with the same name (X). On the right is a positive pattern: it expresses the inherited attribute property (in this case, the inherited attribute named X is mapped into two columns in the database model). More detailed examples of patterns and specifications can be found in the paper on *trans*ML [16].

In the next section we consider the next phases of the BX engineering lifecycle, focusing on architecture and design; we will explore further aspects of *trans*ML for supporting these phases.

4 Architecture and Design

In this section we motivate and present the flavour of an approach for developing the architecture and design of a BX, including MDE languages that can be used to capture detailed designs of BX, as well as techniques for expressing and applying design patterns for BX. What we present here builds on the techniques introduced in the last section, where we used *trans*ML to capture requirements for BX. We omit an end-to-end example, instead aiming to focus on touching on a variety of techniques that can be used to engineer BX solutions.

As discussed earlier, large and complicated BX are similar to large and complicated software systems: they involve many parts (e.g., transformation components, rules) with complicated inter-relationships and dependencies. Many BX have sophisticated behaviour which can be difficult to interpret from their concrete syntax. They are also difficult to engineer correctly. Large software systems are usually not monolithic: they are built as a set of interrelated components. Arguably, BX should be constructed in the same way.

Nevertheless, architecture for BX – and transformations in general – can be complicated. Some of the issues are as follows.

– *Components:* what are appropriate component models for BX? For software systems we have a reasonable understanding of what a component in a software architecture is, how it may be implemented, and how it can be precisely

combined with other components. Our understanding of components for BX and transformations in general is underdeveloped. Most transformation languages offer a notion of a *rule*, and some languages have a notion of *module*, but richer and deeper understanding (e.g., of ports, protocols, and architectural styles) is missing.
- *Relationships:* what are appropriate relationships that can be defined between BX components? For software systems we have a comprehensive library of component connectors (e.g., protocols, buffers, compositions, containments) that can be deployed; a similar understanding for BX is not yet available.
- *Interoperability:* a key aspect of software architecture is what it provides in terms of interoperation with external systems. For BX, the question is: how can a BX be integrated with other components or architectures, e.g., code generators, verification tools, etc.

We will now present an approach to transformation architecture embodied in *trans*ML and present several small examples of both BX architecture and unidirectional transformation architecture. We then describe an approach for detailed design for transformations.

4.1 BX Architecture in *trans*ML

In Sect. 3 we introduced the *trans*ML approach and explained its support for requirements specification (including scenarios and formal requirement specification). As illustrated in Fig. 5, *trans*ML provides support for expressing transformation architectures and designs.

Architecture in *trans*ML is embodied in a traditional architectural modelling approach: an architecture is a set of components and connectors that interact via directional interfaces. Component types are given in terms of metamodels, or event types (for supporting event-driven architectures or for events generated by sensors) or other components (to support higher-order transformations). The component model is general in the sense that it can be used to represent transformations, black-box components (e.g., non-transformation or non-MDE components), or actors (e.g., human users).

The *trans*ML metamodel for architectures is illustrated in Fig. 12. It is worth noting the *direction* attribute on the Interface element; components of BX may both generate and receive information via interfaces.

Constraints on interfaces can be used to impose a concept of contract, e.g., to restrict expected inputs and outputs, but also to support conformance checking.

Figure 13 shows an example of a unidirectional transformation architecture, using a simple component-based concrete syntax from UML. This example illustrates a transformation-centric view, i.e., the components in the architecture are themselves transformations. This can be contrasted with a type-centric architecture, shown in Fig. 14, where the components are types (or metamodels). In both cases, the example architecture is for a chain of transformations between an object-oriented model and SQL code.

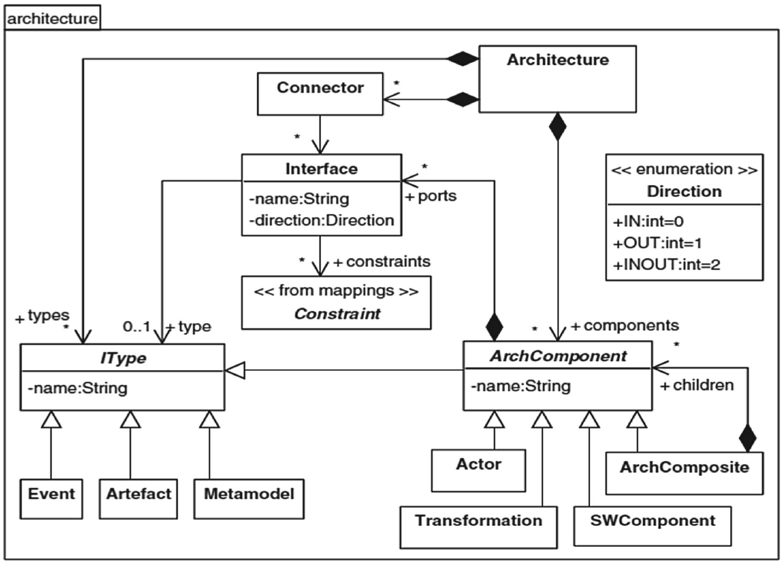


Fig. 12. *trans*ML architecture metamode [16]

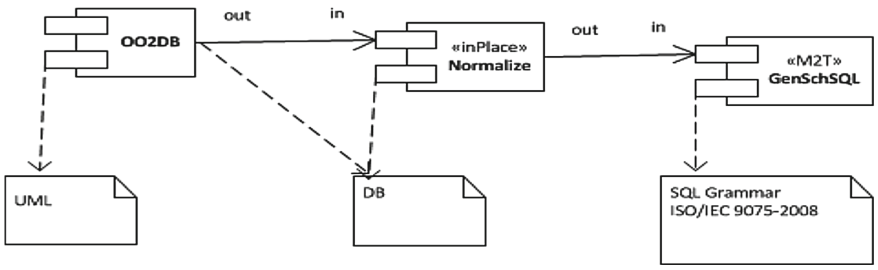


Fig. 13. *trans*ML architecture example (transformation-centric, undirectional)

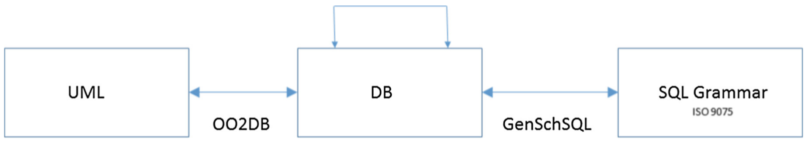


Fig. 14. *trans*ML architecture example (type-centric, bidirectional)

In the above example, firstly a unidirectional OO2DB transformation is executed (taking a UML model as input and producing a DB model as output). Then, a normalising update-in-place transformation is executed on the DB model. Finally, a model-to-text transformation is executed on the DB model, producing SQL code compliant to a specific grammar.

The type-centric view represents the individual transformations as relationships between components. We have extended this example to represent bidirectional transformations throughout: i.e., OO2DB, Normalise and GenSchSQL (the model-to-text transformation) could be executed in either direction. We could, of course, present the same BX in a transformation-centric style. In this case, the architecture in Fig. 13 would have bidirectional dependencies on the relevant input and output models, as depicted in Fig. 15 (in the figure we have circled the ports and connectors to highlight the bidirectionality of information flow).

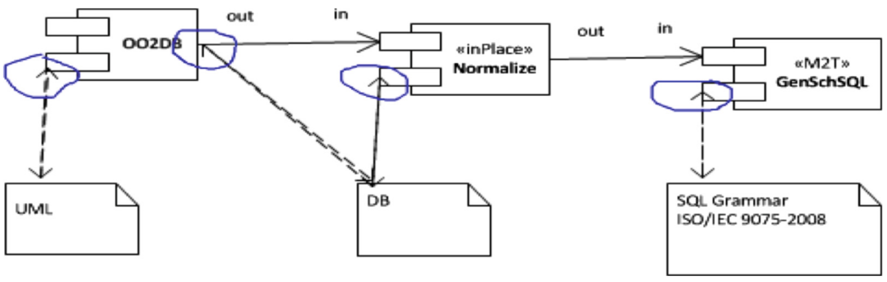


Fig. 15. *trans*ML architecture example (transformation-centric, bidirectional)

4.2 Design of BX

The architecture of a software system captures the key components and their interrelationships. In the case of a BX this includes the connections between transformation components, the ports through which components communicate, and restrictions and constraints on that communication. The engineering process for BX continues with design, which can be broken into two parts: *high-level design*, which focuses on capturing *what is transformed into what*; and *low-level design*, which focuses on capturing *how* the transformation is to be carried out. We briefly consider *trans*ML support for each aspect.

High-level design of a BX, once again, aims to capture what is transformed into what. To represent this, *trans*ML introduces a *mapping diagram*, inspired by triple graph grammars. These capture the mappings between arbitrary model elements involved in the transformation. However, mappings are not meant to be used as a implementation model – specifically, they are not meant to be used as a tracing mechanism to guide the execution of code (this, as we will soon see, is the purpose of the low-level design features of the *trans*ML family of languages).

The *trans*ML metamodel for mapping diagrams is illustrated in Fig. 16. Mappings have ends which are associated with modelling elements. Navigability is a property of mappings; BX will involve navigation to both source and target. Constraints can be attached to mappings in order to define conditions on when (part of) a mapping can hold.

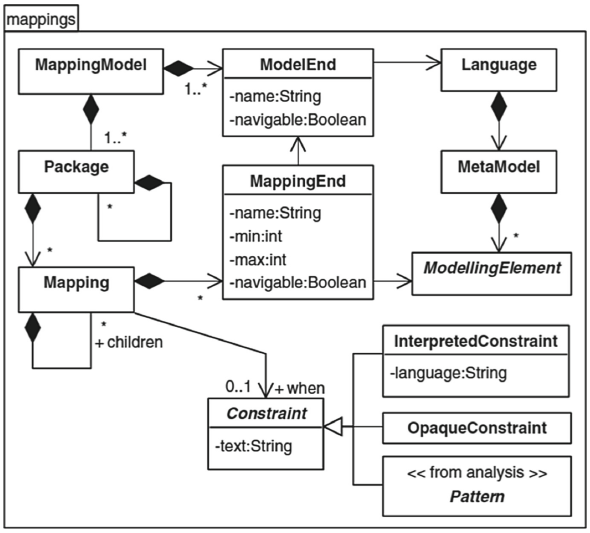


Fig. 16. *trans*ML mapping diagram metamodel [16]

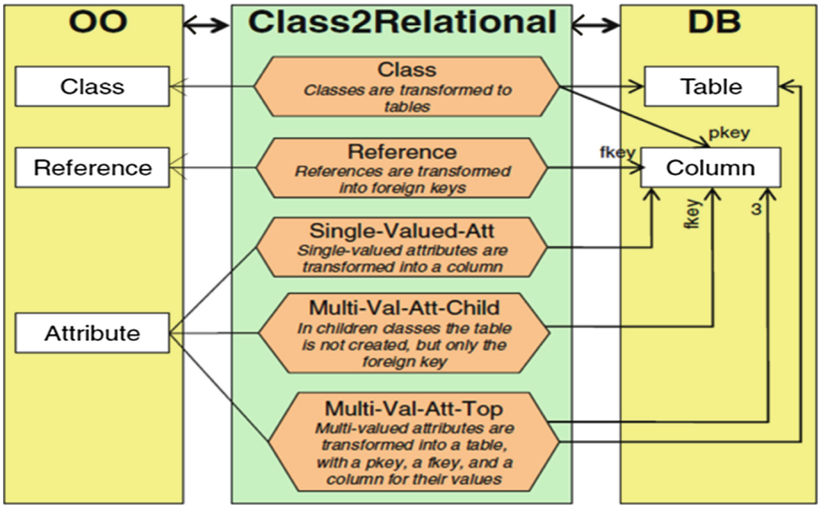

OO
Class2Relational
DB
Class
Classes are transformed to tables
Table
Reference
References are transformed into foreign keys
pkey
fkey
Column
Single-Valued-Att
Single-valued attributes are transformed into a column
fkey
3
Multi-Val-Att-Child
In children classes the table is not created, but only the foreign key
Attribute
Multi-Val-Att-Top
Multi-valued attributes are transformed into a table, with a pkey, a fkey, and a column for their values

Fig. 17. *trans*ML mapping example [16]

Figure 17 illustrates a mapping, for the OO2DBl BX. On the left of the diagram is a package containing key modelling elements of an OO model; on the right, a database model. In the centre are the mappings along with some informal English text explaining the purpose of each distinct mapping. Note the navigability of each rule; these can be executed from a DB model to an OO model, or vice versa.

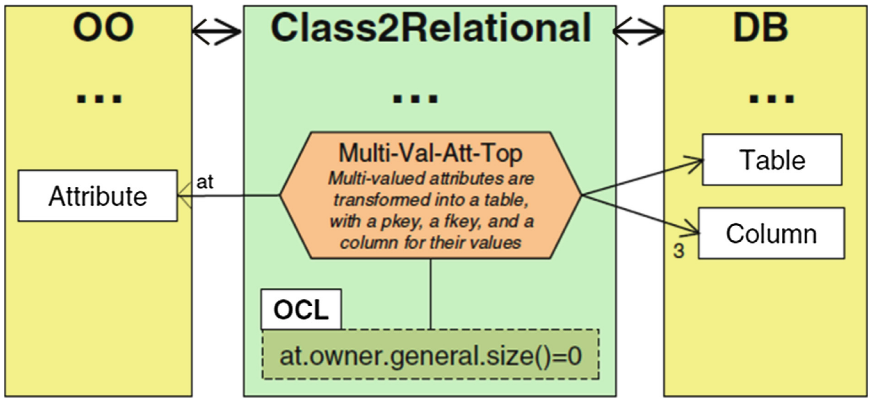

OO
...
Attribute
at
Class2Relational
...
Multi-Val-Att-Top
Multi-valued attributes are transformed into a table, with a pkey, a fkey, and a column for their values
OCL
at.owner.general.size()=0
DB
...
Table
Column
3

Fig. 18. *trans*ML mapping example (constraint) [16]

The next example, in Fig. 18, elaborates what is presented in Fig. 17 and imposes a constraint on the very last mapping, Multi-Val-Att-Top. The constraint, expressed in OCL, states that the owner of an attribute cannot have any parent classes; this is so that multi-valued attributes can be appropriately flattened into a table.

While high-level design is supported in *trans*ML via mapping diagrams, low-level design – which is where the transition to implementation begins – is supported by more detailed diagrams. Technically, low-level design *could* be supported by using a favourite BX programming language. But it may be preferable – for reasons of process – to maintain a degree of platform independence while still focusing on the essential aspects of BX development. As such, *trans*ML provides low-level design languages for capturing the *structure* of BX rules, control flow, and blocks. These are encapsulated in two diagrams: the *rule structure diagram* and the *rule behaviour diagram*.

The rule structure diagram (metamodel in Fig. 19) is used to refine a mapping diagram. A rule in such a diagram can contribute to the implementation of one or more mappings. Rules themselves may be unidirectional or bidirectional. Structure diagrams also allow for explicit or implicit (e.g., nondeterministic) capture of execution flow, via subclasses of the *Flow* metaclass. In particular, a set of rules can be placed inside a nondeterministic block, for example, as in graph transformation programs.

Effectively, rule structure diagrams capture the structure rules, execution flow and data dependencies. This is illustrated in Fig. 20, which shows a directional transformation from an object-oriented model to a database model. The structure in particular is tailored to a representation of rules in the Epsilon Transformation Language (ETL). There is a top-level rule (Class2Table) that is executed initially; its execution is followed by a block of rules that execute nondeterministically; these populate the structure of a database table (i.e., Reference2Column, SingleValuedAtt2Column, MultiValuedAtt2Table). Note that blocks can be a

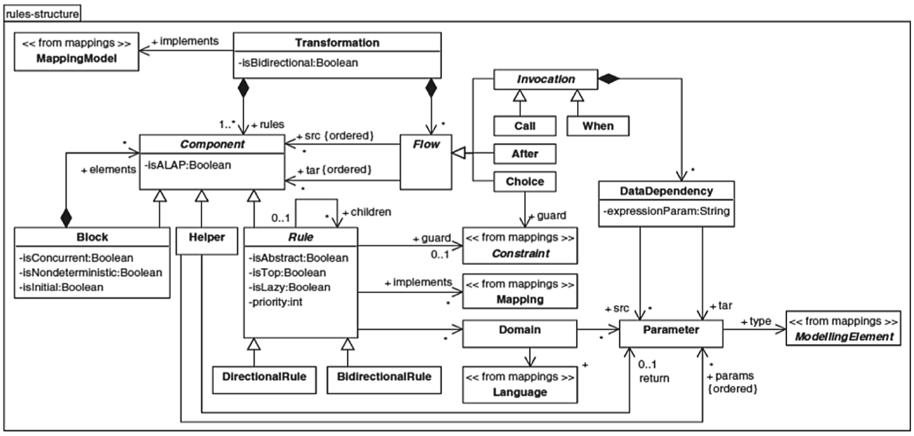


Fig. 19. *trans*ML rule structure diagram metamodel [16]

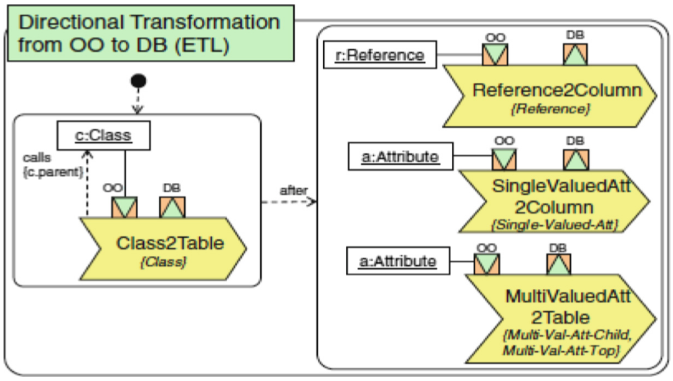


Fig. 20. *trans*ML rule structure diagram example [16]

useful mechanism for design, even if the ultimate implementation language does not support them (for example, ETL does not support blocks directly).

A second example is shown in Listing 1.2. In this case, a small domain-specific BX language is used to specify parts of a transformation between trees and graphs. The transformation is divided into two nondeterministic blocks; these blocks encapsulate bidirectional rules between elements of one model (e.g., Tree) and elements of a second model (e.g., Node).

Rule structure diagrams in particular need to take into account the choice of ultimate implementation language. This is because these diagrams capture execution flow, which is platform specific. For example, consider ETL: the execution flow model is such that each rule is executed once at each instance of input; by comparison, in a graph transformation language, execution is for "as long as possible", i.e., until a fix-point is reached. As such, a specific rule

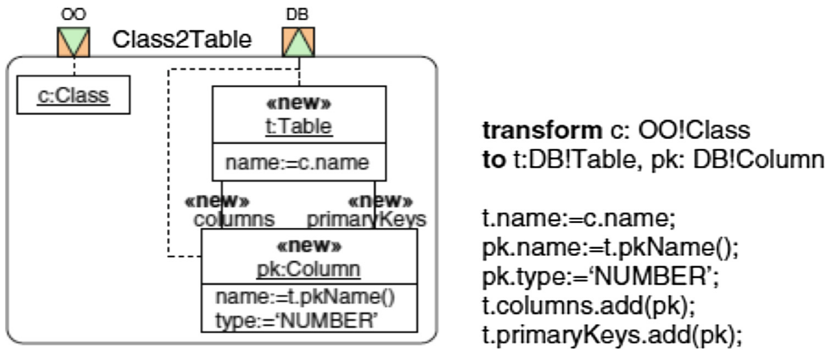
```
transformation Tree2Graph {
  nondeterministic RuleBlockForward {
     bidirectional Tree2Node { ... };
     bidirectional TreeEdge2GraphEdge {...};
   }

   nondeterministic RuleBlockBackward {
      bidirectional TreeLabelsfromNodeLabels {...};
      bidirectional TreeEdgesfromGraphEdges { ... };
    }
}
```

Listing 1.2. An example of a BX using blocks

structure diagram may be transformed easily to one implementation language, but not another. The metamodel for rule structures is, in our experience, sufficiently generic to capture a number of transformation implementation languages, but there may be specific features of specific implementation languages that we have not considered that are not easily supported.

The rule structure diagram treats rules as black boxes, ignoring their behaviour. As such, concepts such as attribute contribution, object creation, or link configuration will be ignored. These can all be specified using implementation languages such as ETL, but *trans*ML also provides a diagram for their specification: the rule behaviour diagram. This allows the behaviour of rules to be captured using an action language, or declarative graphical pre- and post-conditions, or object diagrams annotated with operations (similar in a sense to Catalysis snapshots). An example unidirectional rule behaviour diagram is shown in Fig. 21. On the left of the figure is a snapshot with annotations indicating creation of objects. On the right is the ETL program that would correspond to such a diagram.



```
transform c: OO!Class
to t:DB!Table, pk: DB!Column

t.name:=c.name;
pk.name:=t.pkName();
pk.type:='NUMBER';
t.columns.add(pk);
t.primaryKeys.add(pk);
```

Fig. 21. *trans*ML rule behaviour diagram example [16]

It should be noted that while we have broad and quite deep experience of using *trans*ML for engineering unidirectional transformations, we have much less experience of using it for engineering BX. Using some of the features of *trans*ML for capturing different aspects of BX may be a useful contribution to the BX community, as they provide platform-independent ways of specifying different features.

4.3 Design Patterns for BX

In this section we very briefly discuss several design patterns [17] for BX. Design patterns in general capture recurring design problems (e.g., in object-oriented design) and their solutions. Solutions generally need to be instantiated for particular problem concepts. Many different patterns have been developed and captured in the literature, including some for model transformations. In this section we present three examples, taken from Lano et al. [18] with some customisation for our context.

Auxiliary Correspondence Model Pattern. A special kind of model transformation is a *merging* or *weaving*, where two or more models are combined into a single model. This weaving process can be carried out in batch mode or via a change propagation approach, where changes from the models being combined can be propagated to others. In doing so, most such transformations make use of a so-called *auxiliary correspondence model*. This is a design pattern: the auxiliary correspondence model defines auxiliary model elements and associations that link source and target elements. It can be used to record mappings performed by a BX and to propagate modifications when one model changes. The benefit of using such a pattern is that it separates concerns: the source and target models are kept separate from the connections that link their elements. In turn, these explicit links between source and target model can make it easier to check correctness and coverage in the transformation. The disadvantage of applying this pattern is that it requires maintenance of an additional model.

Unique Instantiation Pattern. This pattern focuses on improving the efficiency of transformations. In particular, it is applied to avoid duplicating model elements in either the source or the target of a BX. In particular, the pattern imposes a check that an element satisfying specified properties does not exist, before the element is actually created in the source or target. For example, in a QVT-Relations transformation that has applied this pattern, new elements will not be created if there are already elements that satisfy the relations specified; this is really at the heart of check-before-enforce mode in QVT-Relations. The benefit of using this pattern is that it can help ensure hippocraticness; the disadvantage is the test for existence, which can degrade BX performance. However, we note that other patterns, e.g., related to indexing [18] – and model indexing frameworks like Hawk – can help offset this.

Map Objects Before Links Pattern. This pattern is used to separate the relation between elements in source and target models from the relations between *links* in the models. A particular application of this pattern would be to structure a transformation wherein model elements are transformed before the relations between model elements (i.e., nodes before edges). Such an execution flow may be useful in cases where models may have self-associations or circular dependencies. The benefit of using this pattern is similar to that of the Visitor pattern [17] in object-oriented design: the specification of the transformation is modular and processing for a new type of association in a modelling language can be more easily handled. The disadvantage of using this pattern is that while edges (relations) are treated modularly, nodes (model elements) may not be, and if a new feature is added to a language, it may require significant restructuring to the transformation that has used this pattern.

4.4 Summary

In this section we have discussed different aspects of the architecture and design of BX, covering abstract architecture for transformations, through high-level and low-level design, including behaviour of individual transformation rules, as well as a selection of design patterns that can be used to help increase cohesion and decrease coupling in our BX. We will next briefly discuss an approach to verification of BX, focusing on use of mathematical techniques.

5 Verification

In this section we explore a specific approach to verifying bidirectional transformations. The approach we present is intended to be pragmatic, meant to be used with existing MDE tools and technologies. As such we do not consider issues such as soundness or completeness, though the mechanisms are present to prove conjectures related to these properties if so desired.

BX are challenging to implement on account of the inherent complicatedness (or complexity!) that they must encode. Model transformation languages supporting them often do so with conditions: some require that BX are bijective (e.g. BOTL [19]), whereas others require users to work with specific formalisms such as triple graph grammars (e.g. MOFLON [20]). Many modern transformation languages do not provide any support for BX (e.g. ATL [2]), meaning that users must express them as two unidirectional transformations. While this seems a practical workaround, the two transformations may diverge over time – that is, there are no guarantees that the two unidirectional transformations maintain the consistency relationship between the models.

A trade-off between the benefit (but complexity) of pure BX languages and the practicality (but possible incoherence) of unidirectional transformations can be achieved in Epsilon. Epsilon has languages supporting the specification of unidirectional transformations in either a rule-based (ETL), update-in-place

(EWL), or operational (EOL) [21] style. Furthermore, it provides an inter-model consistency language (EVL [22]) that can be used to express and evaluate constraints between models. With these languages, BX can be simulated by: (1) defining pairs of unidirectional transformations for separately updating the source and target models; and (2) defining inter-model constraints in EVL, the violation of which will trigger EWL transformations to restore consistency.

Although this process gives us a means of checking consistency and automatically triggering a transformation to restore it, we lack the important guarantee that BX give us: the compatibility of the transformations. It might be the case that after the execution of one transformation, the other does not actually restore consistency, leading to further EVL violations. Thus, how do we check for, and maintain, compatibility?

We aim to obtain the guarantees of BX without the need for BX languages. Instead, we can use *rigorous* proof techniques to verify that faked BX are consistency preserving, and thus indistinguishable to users from true BX. To this end, we propose to apply techniques from graph transformation verification. Given a faked BX in Epsilon, we will model the unidirectional transformations as graph transformation rules, and EVL constraints as nested graph conditions [23]. Then, by leveraging graph transformation proof calculi [24–26] in a weakest precondition style, we aim to automatically prove compatibility of the unidirectional transformations with respect to the EVL constraints.

5.1 Illustration

To illustrate the idea, consider yet again the OO2DB problem. Consistency between a typical OO and a typical DB model is defined in terms of a correspondence between the data in the models, e.g. every table n corresponds to a class n, and every column m corresponds to an attribute m. Figure 22 contains two simple models that are consistent in this sense (we omit the metamodels, but they are obvious).

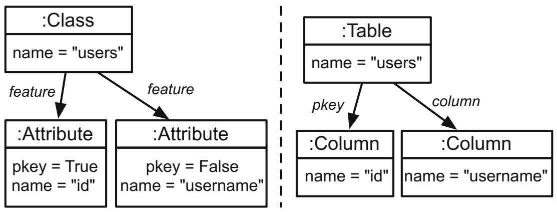


Fig. 22. Two consistent OO and DB models

Users of the models should be able to create new classes (or tables) whilst maintaining inter-model consistency. Upon the creation of a new class (resp. table), a table (resp. class) should be created with the same name to restore consistency. We can implement such a simple BX in Epsilon with a pair of

```
wizard AddClass {
 do {
   var c: new Class;
   c.name = newName;
   self.Class.all.first().contents.add(c);
 }}

wizard AddTable {
 do {
   var table: new Table;
   table.name = newName;
   self.Table.all.first().contents.add(table);
 }}
```

Listing 1.3. Example wizards for simulating BX

```
context OO!Class {
 constraint TableExists {
   check : DB!Table.all.select(t|t.name = self.name).size() > 0
}}

context DB!Table {
 constraint ClassExists {
   check : OO!Class.all.select(c|c.name = self.name).size() > 0
}}
```

Listing 1.4. Inter-model constraints

unidirectional transformations (one for updating the class diagram model, one for updating the relational database) and a set of EVL constraints. For the former, we can use the Epsilon Wizard Language (EWL) to define a pair of update-in-place transformations, `AddClass` and `AddTable` (for simplicity, here we assume the new class/table name `newName` to be pre-determined and unique, but Epsilon does support the capturing and sharing of such data between wizards).

Using the Epsilon Validation Language (EVL), we express inter-model consistency: that for every class n, there exists a table named n (and vice versa). If one of the constraints is violated, Epsilon can automatically trigger the relevant transformation to attempt to restore consistency. For example, after executing the transformation `AddClass`, the constraint `TableExists` will be violated, indicating that the transformation `AddTable` should be executed to restore consistency.

This example of a bidirectional transformation, simulated in Epsilon, is a simple one chosen to illustrate the concepts. Even what appears to be a simple BX can lead to more interesting (i.e. less symmetric) BX, e.g. manipulating inheritance in the class model.

5.2 Checking Compatibility

A critical difference between the simulated BX in the previous section and a true BX is the absence of guarantees about the compatibility of the transformations: upon the violation of `TableExists`, for example, does the execution of `AddTable` actually restore consistency? For this simple example, a manual inspection will confirm that the transformations are indeed compatible. But what about more intricate BX? And what about BX that evolve and change over time? For the Epsilon-based approach to be a convincing alternative to a BX language, it is imperative that the compatibility of the transformations can be checked, and that this can be done in a simple and automatic way. To this end, we propose to leverage and adapt some recent developments in the verification of graph transformations.

Graph transformation is a computation abstraction: the state of a computation is represented as a graph, and the computational steps as applications of rules (i.e. akin to string rewriting in Chomsky grammars, but lifted to graphs). Modelling a problem using graph transformation brings an immediate benefit in visualisation, but also an important one in terms of semantics: the abstraction has a well-developed algebraic theory that can be used for formal reasoning. This has been exploited to facilitate the verification of graph transformation systems, i.e. calculi for systematically proving specifications about graph properties before and after any execution of some given rules. In particular, we look to exploit work by Poskitt and Plump, who developed proof calculi for graph programs, separately addressing reasoning about programs and properties involving attribute manipulation [25,26].

Our example BX for the OO2RDBMS problem can be translated into graph programs and nested conditions, as given in Fig. 23. The programs P_S, P_T are the individual rules creating a class or table node labelled `newName` (here, $\emptyset$ denotes the empty graph, indicating that the rules can be applied without first matching any structure, i.e. unconditionally). The nested condition *evl*, given on the right, expresses that for every class (or table) node, there is a table (or class) node with the same name (note that $\mathtt{x}, \mathtt{y}$ are variables, and that the numbers indicate when nodes are the same down the nesting of the formula). Were the weakest liberal preconditions to be constructed, we would find:

$$\mathrm{Wlp}(P_S; P_T, evl) \equiv \mathrm{Wlp}(P_T; P_S, evl) \equiv evl.$$

Since $evl \Rightarrow evl$ is valid, both $\{evl\}\ P_S;\ P_T\ \{evl\}$ and $\{evl\}\ P_T;\ P_S\ \{evl\}$ must hold, and – assuming correctness of the abstractions – the original EWL transformations are therefore compatible with respect to the EVL constraints.

A key challenge with an approach such as this is what to do when the verification step fails, i.e., the implication above does not hold. We are exploring the use of the GROOVE tool[10] to generate counterexamples when verification fails, via exploring executions of the graph transformation rules.

[10] http://groove.sourceforge.net/groove-index.html.

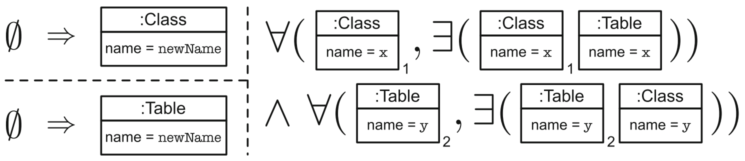

∅ ⇒ :Class name = newName
∅ ⇒ :Table name = newName
∀(:Class name = x 1, ∃(:Class name = x 1 :Table name = x))
∧ ∀(:Table name = y 2, ∃(:Table name = y 2 :Class name = y))

Fig. 23. Our CD2RDBM BX expressed as graph transformation rules and a nested condition

6 Conclusions and Perspectives

Bidirectional transformations must be engineered, as must unidirectional transformations and other programs that manipulate models in MDE. The state-of-the-art in engineering BX is piecemeal at the moment: there are some specific techniques for supporting different engineering phases – such as requirements engineering or design – but very coarse understanding of efficient and effective engineering lifecycles, and alternative process models. This paper attempts to capture some of the current thinking on engineering BX. It summarises some of the state-of-the-art in BX design and implementation, presents some approaches for requirements specification and analysis, and suggests some ideas for capturing the architecture of complicated BX, and the detailed design of BX in general. It also presents some ideas on an approach for verification of BX; this approach is pragmatic, in the sense that it is meant to be used within an engineering process and it acknowledges tradeoffs between completeness and soundness.

MDE for BX possesses some sound theory – such as delta lenses – and some pragmatic, if incomplete, tools (such as Eclipse QVT-Relations) but these are still siloed: the theory needs to inform the enhancement of tools, and the tools need to be used to test the corners of the theory. A good example of research that attempts to link BX theory and practice is that combining triple graph grammars and delta lenses (e.g., [27]), but more needs to be done. What is really needed is tools that *evidently* implement the theory in a systematic and audited way.

A key challenge in connecting theory with practical tools is the limitations in our theories of metamodelling. It is questionable whether we have a sound and complete understanding of a type theory for MDE and metamodelling, but this would underpin any attempts to link a theory of BX with the pragmatic tools supporting BX.

We mentioned tools for BX throughout this paper. The standardised tool in the MDE community is QVT-Relations; the Eclipse implementation is still under development. QVT-Relations has been criticised for being very complex, with substantial semantic ambiguity. The development of its Eclipse implementation is revealing some of these ambiguities, but this will only be convincing if supported by a sound theory, e.g., delta lenses. However, the gap between delta lens theory and QVT-Relations is substantial: changing QVT-Relations to conform with delta lenses may be difficult if not impossible; building a new BX that supports delta lens theory is possible, but it would not be QVT-Relations. It is difficult to see how connections between strong theory and MDE standards will play out.

It also remains to be seen whether we can develop a rich, compelling set of industrial scenarios for BX. In our substantial industrial experience of transformations and MDE, we have had only one precise requirement for a BX (across over 20 industrial projects and 13 years of experience), and that was for the results of various forms of analysis (e.g., failure analysis, performance analysis) to be reflected on source models after calculation. It is unclear if such scenarios benefit from the heavyweight machinery of BX. But it should also be noted that requirements for BX sometimes emerge as development proceeds and having ways in which transformations can be *extended* to become bidirectional may be useful.

In Sect. 5 we described an approach to BX that involved specification of inter-model consistency constraints between two models, and the definition of two separate but synchronised update-in-place transformations on the two models. When the constraints were violated and the models became inconsistent, the transformations would be triggered to re-establish consistency. This approach – two simple yet unidirectional transformations instead of a single bidirectional transformation – needs to be clearly related to the BX solution space: when is it more effective to use versus building a full BX?

Finally, we observe that many transformations developed in practice are operational (e.g., those written in EOL or subsets of ATL). As well, there are many model-to-text (or model-to-grammar) transformations that support code generation scenarios. How do these fit in to the BX space? Are they simply too hard to consider? Are there scenarios or types of transformations that simple *should not* (rather than *cannot*) be bidirectionalised? As a challenge, consider the EuGENia tool[11] which is a unidirectional model transformation written in EOL, which automatically generates three models needed by GMF to construct a graphical editor. These are generated by a transformation that takes as input a single annotated Ecore model. The transformation is defined entirely operationally, as we found that it would be too complex to implement using declarative rules (it is not a mapping transformation). Could EuGENia be turned into a bidirectional transformation? Our intuition is no (and, more pragmatically, we cannot see any reason why one would want to do so), but it would be interesting to explore what, fundamentally, makes an operational or hybrid transformation difficult to bidirectionalise.

Acknowledgements. Parts of this work were supported by the European Commission's 7th Framework Programme, through grant #611125 (MONDO). The author also acknowledges the support of Innovate UK and the Aerospace Technology via the SECT-AIR grant, and the EPSRC, for their support for the Summer School in Bidirectional Transformations. The author thanks Dimitris Kolovos, Chris Poskitt, Arend Rensink, Mike Dodds, Esther Guerra and Juan de Lara for many useful discussions and collaboration on the topics presented in this paper, and to the reviewers of this manuscript for the helpful suggestions and advice.

[11] https://eclipse.org/epsilon/doc/eugenia/.

References

1. Van Gorp, P., Engels, G. (eds.): ICMT 2016. LNCS, vol. 9765. Springer, Cham (2016). https://doi.org/10.1007/978-3-319-42064-6
2. Jouault, F., Allilaire, F., Bézivin, J., Kurtev, I.: ATL: a model transformation tool. Sci. Comput. Program. **72**(1–2), 31–39 (2008)
3. Kolovos, D.S., Paige, R.F., Polack, F.A.C.: The epsilon transformation language. In: Vallecillo, A., Gray, J., Pierantonio, A. (eds.) ICMT 2008. LNCS, vol. 5063, pp. 46–60. Springer, Heidelberg (2008). https://doi.org/10.1007/978-3-540-69927-9_4
4. Kolovos, D.S., Paige, R.F., Polack, F.A.C., Rose, L.M.: Update transformations in the small with the epsilon wizard language. J. Object Technol. **6**(9), 53–69 (2007)
5. Diskin, Z., Xiong, Y., Czarnecki, K.: From state- to delta-based bidirectional model transformations. In: Tratt, L., Gogolla, M. (eds.) ICMT 2010. LNCS, vol. 6142, pp. 61–76. Springer, Heidelberg (2010). https://doi.org/10.1007/978-3-642-13688-7_5
6. Anjorin, A.: An introduction to triple graph grammars as an implementation of the delta-lens framework. In: Gibbons, J., Stevens, P. (eds.) Bidirectional Transformations. LNCS, vol. 9715, pp. 29–72. Springer, Cham (2018)
7. OMG. MOF 2.0 QVT V1.3. Object Management Group (2016)
8. Stevens, P.: A simple game-theoretic approach to checkonly QVT relations. Softw. Syst. Model. **12**(1), 175–199 (2013)
9. Hoisl, B., Hu, Z., Hidaka, S.: Towards bidirectional higher-order transformation for model-driven co-evolution. In: Hammoudi, S., Pires, L.F., Filipe, J., das Neves, R.C. (eds.) MODELSWARD 2014. CCIS, vol. 506, pp. 153–167. Springer, Cham (2015). https://doi.org/10.1007/978-3-319-25156-1_10
10. Sasano, I., Hu, Z., Hidaka, S., Inaba, K., Kato, H., Nakano, K.: Toward bidirectionalization of ATL with GRoundTram. In: Cabot, J., Visser, E. (eds.) ICMT 2011. LNCS, vol. 6707, pp. 138–151. Springer, Heidelberg (2011). https://doi.org/10.1007/978-3-642-21732-6_10
11. Hidaka, S., Tisi, M., Cabot, J., Zhenjiang, H.: Feature-based classification of bidirectional transformation approaches. Softw. Syst. Model. **15**(3), 907–928 (2016)
12. Tehrani, S.Y., Zschaler, S., Lano, K.: Requirements engineering in model-transformation development: an interview-based study. In: Van Gorp, P., Engels, G. (eds.) ICMT 2016. LNCS, vol. 9765, pp. 123–137. Springer, Cham (2016). https://doi.org/10.1007/978-3-319-42064-6_9
13. Nalchigar, S., Salay, R., Chechik, M.: Towards a catalog of non-functional requirements in model transformation languages. In: Proceedings of the Second Workshop on the Analysis of Model Transformations (AMT 2013), Miami, FL, USA, 29 September 2013
14. IEEE 29148–2011. Systems and software engineering lifecycle processes requirements engineering (2011)
15. Pescador, A., de Lara, J.: DSL-maps: from requirements to design of domain-specific languages. In: Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering, ASE 2016, Singapore, 3–7 September 2016, pp. 438–443 (2016)
16. Guerra, E., de Lara, J., Kolovos, D.S., Paige, R.F., dos Santos, O.M.: Engineering model transformations with transML. Softw. Syst. Model. **12**(3), 555–577 (2013)
17. Gamma, E., Helm, R., Johnson, R., Vlissides, J.: Design Patterns: Elements of Reusable Object-Oriented Software. Addison-Wesley, Boston (1995)

18. Lano, K., Kolahdouz-Rahimi, S.: Model transformation design patterns. IEEE Trans. Softw. Eng. **40**(12), 1224–1259 (2014)
19. Braun, P., Marschall, F.: Transforming object oriented models with BOTL. In: GT-VMT 2002. ENTCS, vol. 4066, pp. 103–117. Elsevier (2003)
20. Amelunxen, C., Königs, A., Rötschke, T., Schürr, A.: MOFLON: a standard-compliant metamodeling framework with graph transformations. In: Rensink, A., Warmer, J. (eds.) ECMDA-FA 2006. LNCS, vol. 4066, pp. 361–375. Springer, Heidelberg (2006). https://doi.org/10.1007/11787044_27
21. Paige, R.F., Kolovos, D.S., Rose, L.M., Drivalos, N., Polack, F.A.C.: The design of a conceptual framework and technical infrastructure for model management language engineering. In: ICECCS 2009, pp. 162–171. IEEE Computer Society (2009)
22. Kolovos, D.S., Paige, R.F., Polack, F.A.C.: On the evolution of OCL for capturing structural constraints in modelling languages. In: Abrial, J.-R., Glässer, U. (eds.) Rigorous Methods for Software Construction and Analysis. LNCS, vol. 5115, pp. 204–218. Springer, Heidelberg (2009). https://doi.org/10.1007/978-3-642-11447-2_13
23. Habel, A., Pennemann, K.-H.: Correctness of high-level transformation systems relative to nested conditions. Math. Struct. Comput. Sci. **19**(2), 245–296 (2009)
24. Habel, A., Pennemann, K.-H., Rensink, A.: Weakest preconditions for high-level programs. In: Corradini, A., Ehrig, H., Montanari, U., Ribeiro, L., Rozenberg, G. (eds.) ICGT 2006. LNCS, vol. 4178, pp. 445–460. Springer, Heidelberg (2006). https://doi.org/10.1007/11841883_31
25. Poskitt, C.M.: Verification of graph programs. Ph.D. thesis, The University of York (2013)
26. Poskitt, C.M., Plump, D.: Hoare-style verification of graph programs. Fundamenta Informaticae **118**(1–2), 135–175 (2012)
27. Hermann, F., Ehrig, H., Orejas, F., Czarnecki, K., Diskin, Z., Xiong, Y., Gottmann, S., Engel, T.: Model synchronization based on triple graph grammars: correctness, completeness and invertibility. Softw. Syst. Model. **14**(1), 241–269 (2015)

Author Index