Towards an algebraic theory of bidirectional transformations

Perdita Stevens

Laboratory for Foundations of Computer Science
School of Informatics
University of Edinburgh

Abstract. Bidirectional transformations are important for model-driven development, and are also of wide interest in computer science. In this paper we present early work on an algebraic presentation of bidirectional transformations. In general, a bidirectional transformation must maintain consistency between two models, either of which may be edited, and each of which may incorporate information not represented in the other. Our main focus here is on lenses [2, 1, 3] which provide a particularly well-understood special case, in which one model is an abstraction of the other, and either the abstraction or the full model may be edited. We show that there is a correspondence between lenses and short exact sequences of monoids of edits. We go on to show that if we restrict attention to invertible edits, *very well-behaved* lenses correspond to split short exact sequences of groups; this helps to elucidate the structure of the edit groups.

1 Introduction

Fundamental to the idea of graph transformations is the idea that a change in one structure can correspond to a change in another in a precise sense. This fundamental idea appears in different guises in many areas of informatics; the guise most familiar to the present author is that of bidirectional model transformations, as they appear in the OMG's Model-Driven Architecture (or, as it is now usually more suggestively called, Model-Driven Development) initiative. A bidirectional transformation R between two classes of models, say M and N, incorporates a precise notion of what it is for $m \in M$ to be consistent with $n \in N$:

$$R \subseteq M \times N$$

It also specifies how, if one model is changed, the other can be changed so as to restore consistency. The forward transformation

$$\overrightarrow{R} : M \times N \longrightarrow N$$

takes a pair of models (m, n) which are not (necessarily) consistent. Leaving m alone, it calculates how to modify n so as to restore consistency. It returns this calculated n' such that $R(m, n')$. Symmetrically,

$$\overleftarrow{R} : M \times N \longrightarrow M$$

explains how to roll changes made to a model from N back to a change to make to a model from M.

For practical reasons, it is preferable that all three elements of the transformation – R, $\overrightarrow{R}$ and $\overleftarrow{R}$ – be expressed in one text; but this will not be essential to our semantic treatment here.

This basic framework is flexible enough to explain a wide range of languages for bidirectional transformations, including for example the OMG's QVT-R (Queries Views and Transformations – Relations) language. That language is discussed in [5], as are the postulates that a bidirectional transformation may be expected to satisfy. The reader is referred to that paper for details. In brief, the two main postulates are *correctness* and *hippocraticness*. Correctness has already been mentioned: it states that the forwards and backwards transformations really do restore consistency, e.g. that the returned n' above really does satisfy $R(m, n')$. Hippocraticness ("first do no harm") states that the transformation must not modify a pair of models which is already consistent (not even by returning a different consistent model). Correctness and hippocraticness go a long way to ruling out "silly" transformations, but something else still seems to be required. In [5] a third postulate, *undoability* is proposed, but this is arguably too strong.

The crucial point to notice is that there may be a genuine choice about how consistency is restored. In the absence of defensible way to define which is the "best" option, we want that choice to be in the hands of the person who designs the transformation. Given $m \in M$, there may be many $n' \in N$ such that $R(m, n')$. Given a model n such that $R(m, n)$ does not hold, the designer of the transformation $\overrightarrow{R}$ should be able to choose which of the possible n' will be returned. Although it may be that our transformation language imposes some limitations, for it to be practically useful it will have to permit considerable choice.

Thus far, our framework, like those typically used in graph transformations, is completely symmetric in M, N. Neither model is necessarily an abstraction of the other: each may contain information which is not contained in the other. We will begin with this general situation, but later we shall specialise to the particular case where N is an abstraction of M. This is the situation studied by the Harmony group and reported on in a series of papers including [2, 1]. Much of the present paper can reasonably be seen as "just" a translation into algebraic language of that work, sometimes with generalisation, sometimes with restriction. At the end of the paper we will discuss why this may be a useful undertaking; at the very least, it is hoped that it may amuse the algebraically-inclined reader.

The rest of this paper is structured as follows. In Section 2 we introduce some important equivalence relations that a bidirectional transformation imposes on the sets of models it relates. In Section 3 we discuss *edits* and introduce some basic algebraic ideas. Section 4 shows how to construct a short exact sequence of monoids (or groups) from appropriate lenses, while Section 5 shows how to go the other way, from a suitable sequence to a lens. Finally Section 6 concludes and briefly mentions future work.

A recent survey of bidirectional transformation approaches is found in [6]; these are many, so this paper will not attempt to summarise again, but will stick to the technical focus.

2 Transformations and equivalences

Let R (comprising, by abuse of notation, a consistency relation R, a forward transformation $\overrightarrow{R}$ and a backward transformation $\overleftarrow{R}$) be a transformation which is correct and hippocratic.

We will always assume that there is a trivial or content-free element of each set of models; for example, we will write the trivial element of M as Ω_M. If M is a set of models defined by a metamodel, this might be the model containing no model elements, if that is a member of M, i.e. permitted according to the metamodel. However, it might not be literally empty; if for example all models in M are required to contain some basic model elements, then Ω_M will contain these and nothing else. We will assume that $R(\Omega_M, \Omega_N)$.

Definition 1. *The equivalence relations $\overrightarrow{B}$ and $\overleftarrow{B}$ on M, and $\overrightarrow{F}$ and $\overleftarrow{F}$ on N, are defined as follows:*

- $m \sim_{\overrightarrow{B}} m' \Leftrightarrow \forall n \in N. \overrightarrow{R}(m,n) = \overrightarrow{R}(m',n)$
  (Intuitively, this says "m and m' do not differ in any way that is visible on the N side". The reader familiar with lenses will recognise that this generalises $\sim_g$.)
- $m \sim_{\overleftarrow{B}} m' \Leftrightarrow \forall n \in N. \overleftarrow{R}(m,n) = \overleftarrow{R}(m',n)$
  (Intuitively, "the only *differences between m and m' are those visible on the N side, so that they become indisinguishable after any synchronisation with an element of N". The reader familiar with [1] will recognise that this generalises $\sim_{max}$, the* coarsest *equivalence with respect to which a lens is quasi-oblivious.)*

and dually,

- $n \sim_{\overrightarrow{F}} n' \Leftrightarrow \forall m \in M. \overrightarrow{R}(m,n) = \overrightarrow{R}(m,n')$
- $n \sim_{\overleftarrow{F}} n' \Leftrightarrow \forall m \in M. \overleftarrow{R}(m,n) = \overleftarrow{R}(m,n')$

We can also, in the obvious way give versions of these definitions which are parameterised on subsets of M, N, respectively, the above then being given by plugging in the largest available set, getting the finest available equivalence relations. We do not need any of the coarser equivalences in this paper, however.

Thus, the transformation defines two different equivalences on M (and dually on N). Of course, any element $m \in M$ can then be viewed as a representative of its equivalence class $[m]_{\sim_{\overrightarrow{B}}}$, or as a representative of its other equivalence class $[m]_{\sim_{\overleftarrow{B}}}$. These are the co-ordinates of m in the sense that m is uniquely defined by its two classes; this was already remarked in the case of lenses in [1]:

Lemma 1. *Let $m_1, m_2 \in M$. If both $m_1 \sim_{\overrightarrow{B}} m_2$ and $m_1 \sim_{\overleftarrow{B}} m_2$ then $m_1 = m_2$.*

Proof. We have that for any $n \in N$, $\overrightarrow{R}(m_1, n) = \overrightarrow{R}(m_2, n)$ and $\overleftarrow{R}(m_1, n) = \overleftarrow{R}(m_2, n)$.

Pick $n \in N$ such that $R(m_1, n)$ (which is possible by the existence of trivial elements and correctness of $\overrightarrow{R}$; we could take $n = \overrightarrow{R}(m_1, \Omega_N)$). Then $\overrightarrow{R}(m_1, n) = n$ by hippocraticness, so $\overrightarrow{R}(m_2, n) = n$ by assumption, so $R(m_2, n)$ by correctness. Then $\overleftarrow{R}(m_1, n) = m_1$ by hippocraticness; but also $\overleftarrow{R}(m_1, n) = \overleftarrow{R}(m_2, n)$ by assumption, and the latter is m_2 by hippocraticness, so $m_1 = m_2$. □

A useful picture to bear in mind – although, of course, since M need not be finite or even countable, it is only an informal idea – is of the elements of M laid out on a grid whose columns represent $\sim_{\overrightarrow{B}}$-equivalence classes and whose rows represent $\sim_{\overleftarrow{B}}$-equivalence classes. We have just shown that no square on the grid can contain more than one element of M. In general, not every square need be occupied; indeed, the equivalence classes might have different cardinalities.

The *closure* of M with respect to transformation R, denoted by $\bar{M}$, is the cartesian product of the two sets of equivalence classes, which "contains" M: informally, the set of all squares in the grid. We will have $M = \bar{M}$ in the special case that R is an *undoable* transformation (again, this corresponds to a remark in [1] for lenses).

Since we are, so far, in the completely symmetric case of general bidirectional transformations, the same remarks and result apply to N. In the special case of lenses, which we shall come to, the grid for N, which in that setting is a strict abstraction of M, is degenerate, since then $\sim_{\overrightarrow{F}}$ is universal and $\sim_{\overleftarrow{F}}$ is trivial. In the even more special case of a bijective transformation (or oblivious lens, in the terminology of [2]), the grid for M is also degenerate.

3 Edits and algebraic basics

We will assume that the reader is familiar with the standard notions of group, monoid, mono-, epi- and isomorphisms of groups and monoids, subgroup, submonoid, and normal subgroup. Other definitions from algebra will be reproduced, marked (Standard).

In order to discuss how transformations behave it is useful to have a notion of an *edit*: a way in which a model is changed by its user. When an edit has been done on a model, restoring consistency between it and another model is a matter of performing the "corresponding" edit on the other model. The task of a transformation is then to specify what it means for an edit on one structure to correspond to an edit on the other structure.

The notion of an edit, though, is a little trickier than at first appears. What is an "edit" on a model? Intuitively, it is a thing you can do to an model, changing it into another model. Doing nothing is certainly an edit; edits can be undone; two edits can be done in sucession. Can the same edit be done on *any* model from a given model set; in other words, if we model an edit as an endofunction

$$g : M \longrightarrow M$$

should it be total? It is easy to come up with reasonable examples we might want to model that are ("add a class with a new name in the top level package") but also easy to come up with examples that at first sight are not ("delete the class called `Customer`"). We can get around the problem of edits which are not obviously total by decreeing that if an edit is not naturalistically applicable to a given model, then it should leave the model unchanged ("delete the class called `Customer` if there is one, otherwise do nothing"). In this way, we can model only total edits without imposing any real restriction.

To say that doing nothing is an edit is simply to say that the identity function is an edit. Then to say that edits can be composed is to say that the set of edits is a *monoid*. We will give the definition in order to set up some notation.

Definition 2. *(Standard) A set G provided with an operation*

$$* : G \times G \longrightarrow G$$

(written infix, e.g. $g_1 * g_2$, and in practice normally omitted: $g_1 g_2$) is a monoid if

1. *G contains an identity element, written 1_G, such that for any $g \in G$*

$$1_G * g = g * 1_G = g$$

2. *$*$ is associative: that is, for any g_1, g_2, $g_3 \in G$, we have*

$$(g_1 * g_2) * g_3 = g_1 * (g_2 * g_3)$$

Given a set S, we will often be interested in the monoid of all endofunctions $S \longrightarrow S$, in which the operation is function composition and the identity is the identity, "do nothing" function. We will write this $M(S)$.

What about the fact that edits can be undone? It is tempting to say that this means we have a group of edits, but this is premature. To say that an edit can be undone in the sense that a modelling tool will allow simply means that the tool will retain enough information to reverse any change that the user makes. It does not mean that there will necessarily be an edit g^{-1} which always undoes the effect of edit g, regardless of which model it was applied to. For example, in the case of our edit "delete the class called `Customer` if there is one, otherwise do nothing", there is no inverse, because the edit is not injective.

Definition 3. *(Standard) Let G be a monoid. If in addition G has inverses; that is, for any $g \in G$ there is an element $g^{-1} \in G$ such that*

$$g^{-1} * g = g * g^{-1} = 1_G$$

then G is a group. In that case, inverses are necessarily unique.

Let us pause to observe that an edit can be total without being invertible, and vice versa. For example,

- "delete everything" is total, but not invertible
- "delete package `P` and everything in it" is neither total nor invertible
- "add 1 to constant `MAX`" is not total, but it is invertible where defined
- "swap `true` and `false` wherever they occur" is both total and invertible (as it happens, it is self-inverse).

Given any monoid M of endofunctions on a set S, we will sometimes be interested in the set of all invertible – that is, bijective – elements of M, which of course forms a group. We will write this $G(M)$. Then in particular $G(M(S))$ is the full permutation group on S.

Let us also note that if our transformation engine only sees models, before and after edits, it does not have access to information about what edit the user was doing, in the sense that we do not find out what s/he would have done on a different model; we only see what was done in one instance. Since the user *may* be thinking of a permutation, the transformation certainly has to behave sensibly in that case. Thus let us proceed, for now not committing ourselves to whether we have a group or only a monoid of edits.

Definition 4. *(Standard) Let G be a group or a monoid. The action of G on a set M is a function*

$$\cdot : G \times M \longrightarrow M$$

such that for any g, h, m

1. $1_G \cdot m = m$
2. $(gh) \cdot m = g(h \cdot m)$

We normally omit the dot and just write gm.

If G is a group, i.e. has inverses, it is easy to see that $g^{-1}n = m$ iff $gm = n$. This is why any group action on a set is a permutation action.

3.1 Lenses

We now switch to the restricted setting of lenses, in which one of the models is a strict abstraction of the other. We will use the notation of [1].

The basic premise is that we have two (maybe structured) sets, C and A, connected by an abstraction function $get : C \longrightarrow A$. We consider c and a to be consistent iff $get\ c = a$.

The *get* function, as well as specifying consistency, also provides the forwards transformation. Because of the restricted framework there is no choice, in the sense that the forward transformation is completely determined by the consistency relation: given c, there is a unique consistent a. Thus a lens l corresponds to a special bidirectional transformation R in which $R(c, a)$ holds iff $a = get\ c$, and $\overrightarrow{R}(c, a) = get\ c$ (note that in this special case $\overrightarrow{R}$ ignores a).

We will also need the two equivalence relations on C denoted $\sim_{\overrightarrow{B}}$ and $\sim_{\overleftarrow{B}}$ above, which as remarked are called $\sim_g$ and $\sim_{\max}$ in [1]. In the special case of lenses, we will refer to these equivalences as $\sim_A$ and $\sim_L$ respectively, for reasons which will become apparent. Thus $c_1 \sim_A c_2$ iff $\mathit{get}\ c_1 = \mathit{get}\ c_2$, while $c_1 \sim_L c_2$ iff for every $a \in A$ we have $\mathit{put}\ a\ c_1 = \mathit{put}\ a\ c_2$.

Where the lens designer has a genuine choice is in the *put* function, which corresponds to the backward transformation. A lens also provides

$$\mathit{put} \quad : A \longrightarrow C \longrightarrow C$$

Note that in [2] lenses are for technical reasons not required to be total on their domains, in order that a language of lenses can be defined using recursion; the lenses eventually written by a lens programmer will be total. In this paper, where we consider only semantic issues and do not concern ourselves with the language in which lenses are defined, we are only considering total lenses.

We will, as remarked in the symmetric setting, always assume that there is a trivial or content-free element of C, written Ω_C, and similarly for A. We require $\mathit{get}\ \Omega_C = \Omega_A$ (thus ensuring that Ω_C and Ω_A are consistent, as required) and we derive a function

$$\begin{aligned} \mathit{create}\ &: A \longrightarrow C \\ &\quad a \longmapsto \mathit{put}\ a\ \Omega_C \end{aligned}$$

To complete the definition, lenses are (in this paper) required to satisfy two basic *lens laws*, as follows.

Definition 5. *(adapted from [2]) Let C and A be sets, containing trivial elements Ω_C and Ω_A respectively. A lens from C to A consists of a pair of functions, $\mathit{get} : C \to A$ and $\mathit{put} : A \to C \to C$, such that the following conditions hold:*

$$\begin{aligned} \mathit{get}\ \Omega_C &= \Omega_A && \\ \mathit{put}\ (\mathit{get}\ c)\ c &= c && \text{GetPut} \\ \mathit{get}\ (\mathit{put}\ a\ c) &= a && \text{PutGet} \end{aligned}$$

Note that the lens law CreateGet from [1], viz that for any $a \in A$ we have $\mathit{get}\ (\mathit{create}\ a) = a$, follows from the definition and PutGet.

In general $\mathit{create}\ (\mathit{get}\ c)$ need not of course be c (it could be something else in the same $\sim_A$ equivalence class), but we do have (and will later use):

Lemma 2. *$\mathit{create}\ \Omega_A = \Omega_C$*

Proof. By definition $\mathit{create}\ \Omega_A = \mathit{put}\ \Omega_A\ \Omega_C = \mathit{put}\ (\mathit{get}\ \Omega_C)\ \Omega_C$, which is Ω_C by GetPut. □

This framework is equivalent to a restricted version of the model transformation framework in which the right hand model is required to be an abstraction of the left hand model, and transformations are required to be correct and hippocratic but not undoable. The curious thing is how little those two conditions alone actually restrict the transformation writer: there is an enormous amount of choice about what the put function should do, and many such choices will be in no way defensible as "sensible" behaviour. Formally:

Lemma 3. *Let get* $: C \longrightarrow A$ *be a surjective function, and let* $f_c : A \longrightarrow C$ *be a family of injective functions, one for each* $c \in C$*. Then provided only that* $f_c(get\ c) = c$ *for each* $c \in C$*, get together with the function put defined by* $put\ a\ c =_{def} f_c(a)$ *is a lens.*

Proof. GetPut is true by assumption. Suppose PutGet were violated, so that we have some a, c with $get\ (put\ a\ c) = a' \neq a$. But then $f_c(a') = f_c(get\ (put\ a\ c)) = f_c(get\ f_c(a)) = f_c(a)$ by assumption, contradicting injectivity of f_c. □

Basically, the lens laws force *put* to behave correctly if putting back an abstract element against the concrete element of which it is an abstraction – it is not allowed to break things if nothing has changed – but once any modification has been made in the abstract view, all bets are off. The corresponding issue in the model transformation framework is that hippocraticness requires a transformation not to fix something that isn't broken, but as soon as it is broken in even a trivial detail, the transformation is allowed to do whatever it wants. This is intuitively all wrong: we generally want a tiny change to one model to cause a tiny change to another, or at the very least, only certain enormous changes will seem reasonable! The question of how this should best be captured in a language framework is still open. As discussed in [5], we currently have no entirely satisfactory candidate condition. See also the discussion in [2]. Most convincing, although for some applications too strong, is the law called PutPut in [2]: it states (modulo totality) that for any $a, a' \in A$ and $c \in C$,

$$put\ a'\ (put\ a\ c) = put\ a'\ c \qquad \textsc{PutPut}$$

Definition 6. *(from [2]) A lens is called* very well-behaved *if it satisfies* PutPut*.*

4 Building sequences from lenses

Suppose we are given a lens: that is, sets C and A, each with their trivial element, with functions *get* and *put* satisfying the lens laws (and derived function *create*). In this section we will show how to represent this lens algebraically.

Now, fundamentally what we want to do is to say what edit on one model corresponds to an edit on the other, and we want to do this in such a way that composition of edits is respected and obviously so that doing nothing on one model corresponds to doing nothing on the other.

Lenses, however, do not come equipped with a notion of edit: we have to add that. What should the edits on C be? Our first thought might be to use the whole monoid of functions from C to itself: but in fact, we will need a *compatibility condition* in order for *get*, which is supposed to be an abstraction function, to work as one. The condition is that for any g in our monoid of edits, and for any $c, c' \in C$:

$$\textit{get}\ c = \textit{get}\ c' \Rightarrow \textit{get}\ gc = \textit{get}\ gc' \qquad \textsc{Compat}$$

– in other words, an edit should *act* on $C/\sim_A$. Let $\Pi_C \subseteq M(C)$ be the set of all functions from C to itself that satisfy this compatibility condition. It is easy to see that Π_C is itself a monoid, and that it acts transitively on C, which reassures us that it is expressive enough to model anything the user does to an actual model. In fact, an element g of Π_C is defined by:

1. a function $\bar{g} : C/\sim_A \longrightarrow C/\sim_A$, together with
2. for each $[c] \in C/\sim_A$, a function $g_{[c]} : [c] \longrightarrow [gc]$.

Essentially the compatibility condition says that the abstraction embodied in *get* respects the edits which are allowed, in the sense that if two concrete states look the same in the abstract view before a concrete edit, they will also look the same in the abstract view after the edit. Although this may repay further study, it seems a plausible requirement, in order for there to *be* a notion of edit on the abstract domain which is compatible with the notion of edit on the concrete domain.

Lemma 4. *Any lens induces a monoid homomorphism*

$$\mu : \Pi_C \longrightarrow M(A)$$

defined as

$$(\mu g)(a) = \textit{get}\ (g(\textit{create}\ a))$$

Proof. We have to show that μ preserves identity and composition. Considering identity:

$$\begin{aligned}(\mu 1_G)(a) &= \textit{get}\ (1_G(\textit{create}\ a)) && \\ &= \textit{get}\ \textit{create}\ a && \text{by definition of } 1_G \\ &= a && \text{by } \textsc{CreateGet}\end{aligned}$$

For composition, we have to show that for any g_1, g_2, a

$$(\mu(g_1 g_2))(a) = (\mu g_1)(\mu g_2)(a)$$

By definition of μ we have:

$$(\mu(g_1 g_2))(a) = \textit{get}\ ((g_1 g_2)(\textit{create}\ a))$$

and

$$(\mu g_1)(\mu g_2)(a) = \mathit{get}\ (g_1(\mathit{create}\ \mathit{get}\ (g_2(\mathit{create}\ a))))$$

By CreateGet,

$$\mathit{get}\ g_2(\mathit{create}\ a) = \mathit{get}\ \mathit{create}\ \mathit{get}\ (g_2(\mathit{create}\ a))$$

So by the compatibility condition,

$$\mathit{get}\ g_1 g_2(\mathit{create}\ a) = \mathit{get}\ g_1 \mathit{create}\ \mathit{get}\ (g_2(\mathit{create}\ a))$$

which is what we had to prove.

□

Let us write K and H for the kernel and image of μ, respectively.

Lemma 5. *H acts transitively on A.*

Proof. Given a_1, a_2 in A, we need to show that there is some $g \in \Pi_C$ such that $(\mu g)a_1 = a_2$. Let c be any element of C such that $\mathit{get}\ c = a_2$. Since the action of Π_C on C is transitive, there is some $g \in \Pi_C$ such that $g(\mathit{create}\ a_1) = c$. Then $(\mu g)a_1 = a_2$ as required. □

Lemma 6. *If $g_1 \Omega_C = g_2 \Omega_C$ then $\mu g_1 \Omega_A = \mu g_2 \Omega_A$*

Proof. Suppose $g_1 \Omega_C = g_2 \Omega_C$. Then $(\mu g_1)(\Omega_A) = \mathit{get}\ (g_1(\mathit{create}\ \Omega_A)) = \mathit{get}\ (g_1 \Omega_C)$ by Lemma 2. Similarly, $(\mu g_2)(\Omega_A) = \mathit{get}\ (g_2 \Omega_C)$, and we are done by hypothesis. □

Next, consider the *function* (in the absence of PutPut it is not necessarily a homomorphism, as we shall discuss):

$$\lambda : H \longrightarrow \Pi_C$$

given by

$$(\lambda h)(c) = \mathit{put}\ h(\mathit{get}\ c)\ c$$

This is the function that captures how to "put back" information introduced by a user editing an abstract model, to give a corresponding edit on the concrete model.

Lemma 7. *λ is well-defined.*

Proof. We have to show that for any $h \in H$, the function λh is an element of Π_C: that is, that it satisfies the compatibility condition. Suppose $get\ c_1 = get\ c_2$. Then

$$get\ ((\lambda h)c_1) = get\ (put\ h(get\ c_1)\ c_1) = h(get\ c_1)$$

by definition of λ and PutGet. Similarly, $get\ ((\lambda h)c_1) = h(get\ c_2)$, so we are done by assumption. $\square$

Lemma 8. *$\mu\lambda$ is the identity on H.*

Proof. We have to show that for any $h \in H$ we have $\mu(\lambda h) = h$. Let a be any element of A.

$$\begin{aligned}(\mu(\lambda h)\ a &= get\ ((\lambda h) create\ a)\\ &= get\ (put\ h(get\ create\ a)\ create\ a)\\ &= get\ (put\ h(a)\ create\ a) && \text{by CreateGet}\\ &= h\ a && \text{by PutGet}\end{aligned}$$

$\square$

Thus, the function λ is a right inverse for the epimorphism μ.

Although in general λ may not be a monoid homomorphism, it does behave as such on the identity:

Lemma 9. $(\lambda 1_H) = 1_{\Pi_C}$

Proof. For any $c \in C$, GetPut gives $(\lambda 1_H)c = c$. $\square$

Later, we shall want:

Lemma 10. *For all $g \in \Pi_C$ and for all $h \in H$, we have*

$$h(\mu g)\Omega_A = (\mu g)\Omega_A \Rightarrow (\lambda h)g\Omega_C = g\Omega_C$$

Proof. Expanding the definitions, we may assume $h(\mu g)\Omega_A = (\mu g)\Omega_A$, that is, that $h(get\ (g(create\ \Omega_A)) = get\ (g(create\ \Omega_A)$. Since $create\ \Omega_A = \Omega_C$ (Lemma 2), our assumption becomes $h(get\ (g\Omega_C) = get\ (g\Omega_C$. Now using this, we have to show that $(\lambda h)g\Omega_C = g\Omega_C$. Again expanding the definition, $(\lambda h)g\Omega_C = put\ (h(get\ (g\Omega_C)))\ (g\Omega_C) = put\ (get\ (g\Omega_C))\ (g\Omega_C)$ by assumption, which is $g\Omega_C$ by GetPut. $\square$

To sum up what we have done so far, we need two more standard definitions:

Definition 7. *(Standard) A sequence of groups or monoids*

$$... \to G_{i-1} \overset{\lambda_i}{\to} G_i \overset{\lambda_{i+1}}{\to} G_{i+1} \to ...$$

is exact if for each i,

$$img\ \lambda_i = ker\ \lambda_{i+1}$$

– that is, the elements of G_i which are the images under λ_i of elements of G_{i-1} are exactly those elements of G_i which are mapped by λ_{i+1} to the identity element of G_{i+1}.

Definition 8. *(Standard) A short exact sequence is an exact sequence of length 5, whose ends are trivial:*

$$1 \to K \to G \to H \to 1$$

Therefore we may rephrase what we have shown so far as

Proposition 1. *Let l be a lens from C to A, consisting of functions put and get. Let Π_C be the monoid of endofunctions on C which satisfy* Compat. *Then*

$$1 \to K \to \Pi_C \xrightarrow{\mu} H \to 1$$

is a short exact sequence of monoids, where the monoid homomorphism μ is defined by

$$(\mu g)(a) = get\ (g(create\ a))$$

Moreover, the function $\lambda : H \longrightarrow \Pi_C$ defined by $(\lambda h)(c) = put\ h(get\ c)\ c$ is a right inverse for μ.

However, the usual reason in algebra for considering short exact sequences is that they often encode useful information about the structures in them; unfortunately, in the case of general monoids, they are not so informative. The rephrasing above is suggestive, but not yet very useful. In order to go further, we have to restrict the setting. There are two obvious ways to do this: we can consider only very well-behaved lenses (those which satisfy PutPut), and/or we can restrict attention to invertible edits. Let us consider the first of these restrictions first.

4.1 Very well-behaved lenses

It turns out that insisting that the lens be very well-behaved corresponds exactly to insisting that λ be a monoid homomorphism.

Lemma 11. *If* PutPut *holds then λ is a monoid homomorphism.*

Proof. We have already shown (Lemma 9) that λ preserves identity; now we have to show that it preserves composition.

$$\begin{aligned}
(\lambda h_1)(\lambda h_2)\ c &= (\lambda h_1)(put\ h_2(get\ c)\ c) \\
&= put\ (h_1(get\ (put\ h_2(get\ c)\ c))\ (put\ h_2(get\ c)\ c) \\
&= put\ (h_1(get\ (put\ h_2(get\ c)\ c))\ c) && \text{by PutPut} \\
&= put\ (h_1(h_2(get\ c))\ c && \text{by PutGet} \\
&= \lambda(h_1 h_2)\ c
\end{aligned}$$

□

Lemma 12. *If λ is a homomorphism then* PutPut *holds.*

Proof. Let $a, a' \in A$ and $c \in C$. We have to show that

$$put\ a'\ (put\ a\ c) = put\ a'\ c$$

By transitivity, we can pick h such that $a = h(get\ c)$ and then h' such that $a' = h'a = h'h(get\ c)$. Then $put\ a\ c = \lambda hc$ and $put\ a'\ c = \lambda(h'h)c = \lambda h' \lambda hc$ since λ is a homomorphism. Thus

$$RHS = \lambda h' \lambda hc = \lambda h'(put\ a\ c) = put\ h'(get\ (put\ a\ c))\ (put\ a\ c)$$

which is $put\ (h'a)\ (put\ a\ c)$ by PutGet, which is the LHS by choice of h'. □

This is a very interesting correspondence, particularly in view of the difficulty, mentioned earlier, in choosing an appropriate condition to complement the basic lens laws and ensure "sensible" behaviour. The fact that PutPut corresponds to so basic an algebraic phenomenon as homomorphism is encouraging. Let us now consider the restriction to invertible edits.

4.2 Invertible edits

Recall that for any monoid M, $G(M)$ is the collection of invertible elements of M, which forms a group. Using exactly the same definition as above, we can define $\mu : G(\Pi_C) \longrightarrow G(M(A))$.

However, it turns out that the development we did for monoids will fail in two ways if we try to use an arbitrary lens in conjunction with considering only invertible edits. Firstly, the action of $G(\Pi_C)$ will not necessarily be transitive on C, because if c_1 and c_2 are in $\sim_A$ equivalence classes of different cardinalities, then no invertible element of Π_C can map c_1 to c_2. A consequence of this is that, if we restrict to invertible edits but still consider an arbitrary lens, there might be cases were we could not handle the situation in which a user modified a model c_1, turning it into c_2, and the changes were rolled through to a corresponding model. Since the original lens, which is independent of any notion of edit, can roll through any change a user might make, our algebraic framework would then be failing to describe the full behaviour of the lens. Secondly, our function λ might not be well-defined, since if it is not a monoid homomorphism, it might map an invertible edit to one which is not invertible.

Both of these problems are solved if we assume, for the remainder of the section, that l is a very well-behaved lens, so that PutPut holds. (This may not be the only way to proceed, however.) The development done for monoids now goes through smoothly, using

Lemma 13. *(from [1]) If l is a very well-behaved lens, then there is a bijection between C and the cartesian product $C/{\sim_L} \times C/{\sim_A}$.*

In particular, all the equivalence classes in $C/{\sim_A}$ have the same cardinality: according to the informal grid picture we suggested before, there is exactly one element of C occupying every square of the rectangular grid whose columns are labelled by elements of $C/{\sim_A}$ and whose columns are labelled by elements of $C/{\sim_L}$.

Thus an element g of $G(\Pi_C)$ is defined by:

1. a permutation $\bar{g} : C/{\sim_A} \longrightarrow C/{\sim_A}$, together with
2. for each $[c] \in C/{\sim_A}$, a bijection $g_{[c]} : [c] \longrightarrow [gc]$.

First, we need to check that for any $g \in G(\Pi_C)$, the endofunction μg is indeed invertible. This is immediate from the fact that μ is a monoid homomorphism. Recall that any monoid homomorphism between groups is a group homomorphism. Finally we have to check that λ remains well-defined when restricted. Since λ is a monoid homomorphism, the image of any invertible element is invertible, so it is a group homomorphism.

We will now write G instead of $G(\Pi_C)$.

Now that we are considering groups rather than just monoids, let us return to our short exact sequence. The crucial standard result is

Lemma 14. *(Standard) Let*

$$1 \to K \to G \to H \to 1$$

be a short exact sequence of groups. Then $K \trianglelefteq G$ and $G/K \simeq H$; we say that G is an extension of H by K.

That is, the short exact sequence tells you how G is in a certain sense built from its substructure K together with the extending structure H. Notice, though, that H need not embed in G, i.e., there need not be any group monomorphism from H to G; if these are edit structures, there need not be a systematic way to regard an edit done on an abstract model as an edit done on the concrete model. That is, we cannot necessarily express the edits that can be done on the concrete domain, in terms of edits done on the abstract domain together with other information. Algebraically, this is because – *in general* – a short exact sequence does not necessarily *split*.

Definition 9. *(Standard) A short exact sequence of groups*

$$1 \to K \to G \xrightarrow{\sigma} H \to 1$$

is said to split if there exists a group monomorphism $\lambda : H \longrightarrow G$ which composes with σ to the identity on H:

$$\forall h \in H \sigma(\lambda h) = h$$

In that case, λ is said to split the sequence, and $G \simeq K \rtimes H$.

Definition 10. *(Standard) Let G be a group, with subgroups $K \trianglelefteq G$ and $H \leq G$. G is the (internal) semi-direct product $K \rtimes H$ if:*

– $KH = G$
– $K \cap H = 1_G$

In this case, we observe that

– *every element g of G can be written uniquely as the product $g = kh$ of elements $k \in K$ and $h \in H$;*

- $(kh)^{-1} = (h^{-1}k^{-1}h)h^{-1}$ *(note that $h^{-1}k^{-1}h \in K$ by normality of K);*
- $(k_1h_1)(k_2h_2) = (k_1h_1k_2h_1^{-1})(h_1h_2)$ *(noting again that this is the product of an element of K and one of H, by normality).*

The product is direct if in addition H and K commute.

Our restriction to very well-behaved lenses gives us that λ is a monoid and hence a group homomorphism, which is exactly what is needed to ensure that the short exact sequence splits. That is, we have (summarising)

Theorem 1. *Let l be a very well-behaved lens from C to A, consisting of functions put and get. Let G be the group of invertible endofunctions on C which satisfy* COMPAT. *Then*

$$1 \to K \to G \xrightarrow{\mu} H \to 1$$

is a short exact sequence of groups, where the group homomorphism μ is defined by

$$(\mu g)(a) = get\ (g(create\ a))$$

Moreover, the function $\lambda : H \longrightarrow G$ defined by $(\lambda h)(c) = put\ h(get\ c)\ c$ is a right inverse for μ, and a group homomorphism. Therefore it splits the sequence and we have an isomorphism

$$K \rtimes H \simeq G$$

We can now discuss the action of G on C in terms of what K, H do to $C/{\sim_L}$ and $C/{\sim_A}$.

Lemma 15. *The subgroup λH of G acts, trivially, on $C/{\sim_L}$: that is, for any $c \in C$ and $h \in H$, $(\lambda h)c \sim_L c$.*

Proof. We have to show that for any $a \in A$,

$$put\ a\ ((\lambda h)c) = put\ a\ c$$

The LHS is by definition $put\ a\ (put\ (h(get\ c)\ c)$ which is the RHS by PUTPUT. □

To put it another way, $\lambda(H)$ stabilises the $\sim_L$-equivalence classes.

In particular, $\lambda(H) \leq G$ acts on $create\ A \subseteq C$ just as H acts on A.

Let us identify A with the set $create\ A$ and take this as the transversal of $\sim_A$; and let L be the set $put\ \Omega_A\ C$, and take these elements as the transversal of $\sim_L$. Observe that (in this restricted setting) $create\ a \sim_L create\ b$ for any $a, b \in A$, and also $get\ (put\ \Omega_A\ c) = get\ (put\ \Omega_A\ d)(= \Omega_A)$ for any $c, d \in C$, so we can picture the elements of $create\ A$ laid out as the bottom row and the elements of L in the left-hand column of our grid, respectively. We can identify C with $L \times A$ via the bijection $c \mapsto (put\ \Omega_A\ c, create\ (get\ c))$.

Let us from now on elide λ and regard H as a subgroup of G via λ. [1]

In terms of our informal grid, elements of H stabilise the rows, permuting the elements of each row. Each row is permuted identically. Formally:

[1] That is, as usual we can safely elide the distinction between internal and external semi-direct products.

Lemma 16. *For any $h \in H$ and $(l, a) \in L \times A$ we have $h(l, a) = (l, ha)$.*

Proof. Let $h(l, a) = (l', a')$. We must have $l' = l$ by Lemma 15. By definition $h' = get\ \lambda hc$ where $c = (l, a)$ and $a = get\ c$; but this is $get\ (put\ (h(get\ c)\ c) = h(get\ c) = ha$ by PutGet as usual. □

Next we consider the role of K.

Lemma 17. *The normal subgroup K of G acts, trivially, on $C/{\sim_A}$: that is, for any $c \in C$ and $k \in K$, $kc \sim_A c$.*

To put it another way, K stabilises the $\sim_A$-equivalence classes. In particular, K acts on L. In terms of our informal grid, elements of K stabilise the columns, possibly permuting the elements of each column individually. Unlike H acting on rows, however, K does not necessarily do the same permutation on each column. Suppose for a moment that we are not given G with its action on $L \times A = C$, but instead are given just H and K, together with K's action on L (that is, on the left-hand column of the informal grid only) and H's action on A (that is, on the bottom row of the grid). We may ask, does this information determine the full action of G on $L \times A$? If not, to what extent does it constrain it? Since we know that H acts in the same way on every row, so its action on C is determined by its action on A, the interesting part is how K can act on a general element.

Lemma 18. *Let $k \in K$ and $(l, a) \in L \times A$. Then $k(l, a)$ can be written as $((hkh^{-1})l, a)$ where $h \in H$ satisfies $ha = \Omega_A$.*

Proof. Observe that

$$\begin{aligned}
k(l,a) &= h^{-1}(hkh^{-1})h(l,a) \\
&= h^{-1}(hkh^{-1})(l,ha) && \text{by Lemma 16} \\
&= h^{-1}((hkh^{-1})l,ha) && \text{since } ha = \Omega_A \text{ and } hkh^{-1} \in K \text{ by normality} \\
& && \text{so this is just the action of } K \text{ on } L \\
&= ((hkh^{-1})l,a) && \text{by Lemma 16 again}
\end{aligned}$$

as required. □

Putting these calculations together, we see that the action of G on C can be composed from the actions of H on A and of K on L, thus:

$$(kh)(l, a) = ((h_1 k h_1^{-1})l, ha)$$

where $h_1 a = \Omega_A$.

In general there is a genuine choice of element of H – or equivalently, a genuine choice of semidirect product, that is, of homomorphisms in our short exact sequence – so that the actions of H on A and K on L, devoid of information about how the two groups are connected, do not completely determine G with its action on C. We also need an oracle to make the necessary choices, or, equivalently, to be given the homomorphism μ which determines which of the various semidirect products of H with K is intended.

We should, however, observe two special cases. First, if the action of H on A is such that there is always a unique element h such that $ha = \Omega_A$, then each choice is unique, and the actions of H and K on A and L respectively will completely determine the action of G on C. Second, if G is actually the *direct* product $K \times H$ – that is, elements of K commute with elements of H – then the action of G on C simplifies to the pointwise action

$$(k, h)(l, a) = (kl, ha)$$

as expected, there is no choice to be made, and again the actions of K and H on L and A do completely determine the action of G on C. This means, informally, that in this special case an edit acts independently on the part of the concrete model from C that's retained in the abstract view A and on the part which is discarded by the abstraction.

5 Building lenses from sequences

To show that we really do have an alternative way of looking at this world, we now need to consider the other direction.

Suppose we are given a short exact sequence of monoids

$$1 \to K \to G \xrightarrow{\mu} H \to 1$$

in which G acts on a set C (equipped with a trivial element Ω_C) and H acts on a set A (equipped with trivial element Ω_A). We can read this as telling us how to translate edits on C to edits on A: that is, it already gives us a (unidirectional) model transformation.

If we are given, additionally, an injective function $\lambda : H \longrightarrow G$ such that $\mu\lambda$ is the identity function on H, we can regard this as a bidirectional transformation: it tells us how to translate edits in both directions.

However, it does not necessarily correspond to a lens. Fundamentally the issue is this. Lenses work in the absence of any intentional information about the edits a user has made to the models: the lens only sees the modified models. In principle, there is no reason why we should not define a different kind of bidirectional transformation that does take notice of *how* the user achieved their changes. Two different edits might have the same effect on a model in C (rsp. A), but their images under μ (rsp. λ) might legitimately have different effects on a corresponding model in A (rsp. C).

We will always require that the action of G on C and the action of H on A are transitive, so that there always is some edit that will take the current model to the desired modified model. Beyond this, different choices might represent different means of editing models.

For the rest of this section we restrict attention to those sequences from which lenses can be defined:

Definition 11. *A sequence of monoids as described above is lens-like if it satisfies the following two conditions:*

LL1: if $g_1\Omega_C = g_2\Omega_C$ then $\mu g_1\Omega_A = \mu g_2\Omega_A$
LL2: for all $g \in G$ and for all $h \in H$, we have

$$h(\mu g)\Omega_A = (\mu g)\Omega_A \Rightarrow (\lambda h)g\Omega_C = g\Omega_C$$

Note that any sequence which arises from a lens by the construction in Section 4 is lens-like, as expected, by Lemmas 6 and 10.

Given a lens-like sequence, we can define a get function as follows. Given $c \in C$, let $g \in G$ be any element such that $g\Omega_C = c$; then

$$\mathit{get}\ c =_{\text{def}} \mu g\Omega_A$$

Lemma 19. *get is well-defined*

Proof. There is at least one suitable choice of g by transitivity. LL1 suffices to show that different choices of g give the same result for *get c*. □

Lemma 20. *get and the group action of G on C satisfy the original compatibility condition* Compat.

Proof. Suppose *get* c_1 = *get* c_2. Expanding the definition, this means that we have elements of G, say g_1 and g_2, such that

$$g_1\Omega_C = c_1$$

$$g_2\Omega_C = c_2$$

$$\mu g_1\Omega_A = \mu g_2\Omega_A$$

We need to show, for an arbitrary $g \in G$, that *get* gc_1 = *get* gc_2. Expanding the definition again, we have elements of G, say g_1' and g_2', such that

$$g_1'\Omega_C = gc_1$$

$$g_2'\Omega_C = gc_2$$

and we need to show that $\mu g_1'\Omega_A = \mu g_2'\Omega_A$. Now,

$$\begin{array}{lll} g_1'\Omega_C & = gg_1\Omega_C & \text{rearranging the above, so} \\ \mu g_1'\Omega_A & = \mu gg_1\Omega_A & \text{by LL1} \\ & = (\mu g)(\mu g_1\Omega_A) & \text{since } \mu \text{ is a homomorphism} \\ & = (\mu g)(\mu g_2\Omega_A) & \text{by assumption} \\ & = (\mu(gg_2))\Omega_A & \text{since } \mu \text{ is a homomorphism} \\ & = (\mu g_2')\Omega_A & \text{by LL1, since } g_2'\Omega_C = gg_2\Omega_C \end{array}$$

□

Our definition of *put* involves making a choice, for each pair $a \in A$ and $c \in C$, of an element of the group H which has the desired effect; different choices may give different *put* functions, so our definition is parameterised on an oracle. This is not surprising, in the light of the many choices of put function discussed earlier (Lemma 3).

Suppose we have an oracle which, given arguments a and c, returns $h \in H$ such that $a = h(\mathit{get}\ c)$. (At least one such element is always guaranteed to exist by transitivity of the action of H on A.) Then

$$\mathit{put}\ a\ c =_{\mathrm{def}} \lambda hc$$

Theorem 2. *The put and get functions defined above comprise a lens (for any oracle).*

Proof. GetPut: we have to show that $\mathit{put}\ (\mathit{get}\ c)\ c = c$ for any c. By definition, the LHS is λhc for some h such that $\mathit{get}\ c = h(\mathit{get}\ c)$; that is, such that $(\mu g)\Omega_A = h(\mu g)\Omega_A$ where $g\Omega_C = c$. By LL2 we have $(\lambda h)g\Omega_C = g\Omega_C$; that is, the LHS is c, as required.
PutGet: we have to show that $\mathit{get}\ (\mathit{put}\ a\ c) = a$. Expanding the definitions, we have $\mathit{get}\ (\mathit{put}\ a\ c) = \mu g\Omega_A$ for some g such that $g\Omega_C = \mathit{put}\ a\ c$, which in turn is λhc for some h such that $a = h(\mathit{get}\ c) = h(\mu g'\Omega_A)$ for some g' such that $g'\Omega_C = c$.

Since $g\Omega_C = \mathit{put}\ a\ c = (\lambda h)g'\Omega_C$, LL1 gives $\mu g\Omega_A = \mu((\lambda h)g')\Omega_A$. Since μ is a homomorphism, and $\mu\lambda h = h$, this is equivalent to

$$\mu g\Omega_A = h(\mu g')\Omega_A$$

But we had already showed that the LHS of this equation is $\mathit{get}\ (\mathit{put}\ a\ c)$ while the RHS is a. □

Theorem 3. *If this sequence was in fact constructed from a lens l as described in Section 4, then the lens we construct from the sequence is exactly l (and in particular, it does not then depend on our choice of oracle).*

Proof. We write $l.\mathit{get}$ and $l.\mathit{put}$ for the components of l, get and put for our constructed components.

$$\begin{aligned}
\mathit{get}\ c &= (\mu g)\Omega_A && \text{where } g\Omega_C = c\\
&= l.\mathit{get}\ (gl.\mathit{create}\ \Omega_A)\\
&= l.\mathit{get}\ (g\Omega_C)\\
&= l.\mathit{get}\ c
\end{aligned}$$

$$\begin{aligned}
\mathit{put}\ a\ c &= (\lambda h)c && \text{where } h(\mathit{get}\ c) = a\\
&= l.\mathit{put}\ (h\ l.\mathit{get}\ c)\ c && \text{by definition of } \lambda\\
&= l.\mathit{put}\ (h\ \mathit{get}\ c)\ c && \text{by the above}\\
&= l.\mathit{put}\ a\ c && \text{by choice of } h
\end{aligned}$$

□

Lemma 21. *If, further, λ is a group homomorphism, so that it splits the sequence, and in addition* either *of the following holds,*

1. *G and H are groups; or*
2. *we have the property that $h(get\ c) = h'(get\ c) \Rightarrow (\lambda h)c = (\lambda h')c$*

then the lens is very well-behaved.

Proof. We have to show that PutPut holds; that is, that for any $a, a' \in A$ and $c \in C$ we have

$$put\ a'\ (put\ a\ c) = put\ a'\ c$$

Expanding the definition, the LHS is $\lambda h'(\lambda hc)$ where $a' = h'(get\ (put\ a\ c)) = h'a$ by PutGet and $a = h(get\ c)$. Putting these together, we see that $a' = h'h(get\ c)$. Expanding the RHS, it is $\lambda h''c$ where $a' = h''(get\ c)$.

Since λ is a homomorphism, we have

$$LHS = \lambda h'(\lambda hc) = \lambda(h'h)c$$

If we can show that this is $\lambda h''c$, we are done. If condition 2. in the statement holds, it shows exactly this.

Alternatively, if G and H are groups, then LL2 gives us what we need, for then there is (by transitivity) some g such that $g\Omega_C = c$, and we have $(h''^{-1}h'h)(\mu g)\Omega_A = (\mu g)\Omega_A$, so by LL2, $c = g\Omega_C = \lambda((h''^{-1}h'h))g\Omega_C = \lambda(h''^{-1})(\lambda h')(\lambda h)g\Omega_C = \lambda(h''^{-1})(\lambda h')(\lambda h)c$. Multiplying on the left by $\lambda(h'')$ and using (again) the fact that λ is a homomorphism gives the result. □

6 Conclusions and further work

In this paper we have described an algebraic framework in which to think about bidirectional transformations between sets of models. We have focused on an important special case, where one of the models is an abstraction of the other, and we have shown how to translate key elements of the body of work on lenses into algebraic terms. The lens framework was invented with the pragmatic needs of transformation programmers in mind: yet, that it fits so neatly into the algebraic framework suggests that the choice of laws it embodies are canonical within its region of the transformation language design space.

Much remains to be done, especially in exploiting the algebraic framework to give new (and/or easier) insight into how edit structures and transformations can be composed, and to explore beyond the boundaries of the lens framework. On the other hand, within those boundaries, it would be interesting to incorporate the work on dictionary and skeleton lenses from [1] (where wreath products clearly have a role to play) and on lenses up to equivalences from [3]. Looking more widely, it is to be hoped that the algebraic approach will also be useful in integrating different approaches to bidirectional transformations, including those from the graph transformation community; this may shed light on the design space of bidirectional transformation languages and thus contribute, ultimately, to the development of more useful languages for model-driven development.

From a theoretical point of view, it would be interesting to widen the search for connections into the fields of topology and category theory, and to understand the connections with earlier work such as [4] better. Finally, a major area of future work is to understand the connections with graph grammars, especially triple graph grammars.

Acknowledgements I thank the PC chairs of ICGT, Reiko Heckel and Gabriele Taentzer, for the invitation to write this paper and for comments on a draft. I thank Benjamin Pierce for many helpful discussions, and the many people who have commented on my earlier work on transformations.

References

1. Aaron Bohannon, J. Nathan Foster, Benjamin C. Pierce, Alexandre Pilkiewicz, and Alan Schmitt. Boomerang: Resourceful lenses for string data. In *ACM SIGPLAN–SIGACT Symposium on Principles of Programming Languages (POPL), San Francisco, California*, January 2008.
2. J. Nathan Foster, Michael B. Greenwald, Jonathan T. Moore, Benjamin C. Pierce, and Alan Schmitt. Combinators for bidirectional tree transformations: A linguistic approach to the view-update problem. *ACM Transactions on Programming Languages and Systems*, 29(3):17, May 2007. Preliminary version presented at the *Workshop on Programming Language Technologies for XML (PLAN-X)*, 2004; extended abstract presented at *Principles of Programming Languages (POPL)*, 2005.
3. J. Nathan Foster, Alexandre Pilkiewicz, and Benjamin C. Pierce. Quotient lenses. To appear in Proceedings of International Conference on Functional Programming 2008. Available at http://www.cis.upenn.edu/ jnfoster/papers/quotient-lenses.pdf.
4. Georg Gottlob, Paolo Paolini, and Roberto Zicari. Properties and update semantics of consistent views. *ACM Trans. Database Syst.*, 13(4):486–524, 1988.
5. Perdita Stevens. Bidirectional model transformations in QVT: Semantic issues and open questions. In *Proceedings of 10th International Conference on Model Driven Engineering Languages and Systems, MODELS 2007, Nashville, USA, September 30 - October 5, 2007*, volume 4735 of *Lecture Notes in Computer Science*, pages 1–15. Springer, 2007.
6. Perdita Stevens. A landscape of bidirectional model transformations. In *Post-proceedings of GTTSE'07*, 2008. to appear.