ISSN 1884-0760

GRACE TECHNICAL REPORTS

"Putback" is the Essence of Bidirectional Programming

Sebastian Fischer Zhenjiang Hu Hugo Pacheco

GRACE-TR 2012-08 December 2012

CENTER FOR GLOBAL RESEARCH IN
ADVANCED SOFTWARE SCIENCE AND ENGINEERING
NATIONAL INSTITUTE OF INFORMATICS
2-1-2 HITOTSUBASHI, CHIYODA-KU, TOKYO, JAPAN

WWW page: http://grace-center.jp/

Bidirectional transformations, programs with a forward and a backward transformation that maintain consistency between input and output, are routinely written in ways that do not let programmers specify their behavior completely. Several bidirectional programming languages exist to aid programmers in writing bidirectional transformations with increased maintainability but decreased expressiveness.

Such languages allow programmers to write bidirectional transformations as one program for both directions, which is easier to maintain than separate programs for each direction. However, the maintainability provided by existing bidirectional languages comes at the cost of expressiveness because the ambiguity of synchronization is solved by default strategies which are hidden from programmers. The programmers' inability to influence synchronization strategies has led to the proposal of a vast number of approaches that consider tailor-made synchronization strategies for particular applications.

In this paper, we argue that such ambiguity is essential for bidirectional transformation and advocate that the synchronization strategy should not be hidden from programmers but considered from the start. We propose a novel approach to specifying so called well-behaved bidirectional programs by their backward transformations, capable of expressing all aspects of a bidirectional transformation completely, while retaining maintainability.

Soundness of our approach results from a systematic analysis of the laws describing well-behaved bidirectional transformations based on existing mathematical concepts. We show that well-behaved bidirectional transformations are uniquely determined by their backward transformations and corresponding forward transformations can be obtained for free.

“Putback” is the Essence of Bidirectional Programming

Sebastian Fischer Zhenjiang Hu Hugo Pacheco

December 2012

The further backward you can look,
the further forward you can see.
– Winston Churchill

1 Introduction

Bidirectional transformation (BX for short) [Czarnecki et al., 2009, Hu et al., 2011], originated from the *view updating* mechanism in the database community [Bancilhon and Spyratos, 1981, Dayal and Bernstein, 1982, Gottlob et al., 1988], has been recently attracting a lot of attention from researchers in the communities of programming languages and software engineering since the pioneering work on a combinatorial language for bidirectional tree transformation [Foster et al., 2007]. Bidirectional transformation provides a novel mechanism for synchronizing and maintaining the consistency of information between input and output, and has seen many interesting applications, including the synchronization of replicated data in different formats [Foster et al., 2007], presentation-oriented structured document development [Hu et al., 2008], interactive user interface design [Meertens, 1998] or coupled software transformation [Lämmel, 2004].

1.1 Bidirectional Transformation (BX)

A *bidirectional transformation* consists of a pair of transformations: the *forward* transformation is used to produce a target view from a source, while the *backward* transformation is used to “put back” modifications on the view to the source. To allow the forward transformation to discard information when producing a view, the backward transformation is supplied the original source in addition to the updated view. This situation is depicted in Figure 1 where, as is customary, the forward transformation is called *get* (also know as view function) and the backward transformation is called *put* (shorthand for “putback”).

Example 1.1. *As a simple example for a bidirectional transformation consider a forward function getFirst that selects the first component of a pair and a corresponding backward*

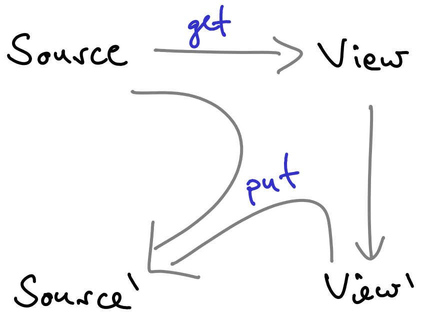

Figure 1: Forward and Backward Directions in Bidirectional Transformations

function putFirst that updates the first component and retains the second component from the original pair.

$$\begin{aligned} &\mathit{getFirst}\ (x,y) &&= x \\ &\mathit{putFirst}\ (x,y)\ z &&= (z,y) \end{aligned}$$

Not every combination of *get* and *put* functions forms a reasonable bidirectional transformation. The definitions need to fit together in the sense that one constitutes the opposite direction of the other. Put formally, the *get* and *put* functions of a bidirectional transformation should be *well-behaved* in the sense that they satisfy the following GetPut and PutGet laws.

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

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

The GetPut property requires that not changing the view shall be reflected as not changing the source, while the PutGet property requires all changes in the view to be completely reflected to the source so that the changed view can be computed again by applying the forward transformation to the changed source.

1.2 Bidirectional Programming

Bidirectional programming is to develop well-behaved bidirectional transformations in order to solve various synchronization problems.

A straightforward approach to bidirectional programming is to write two unidirectional transformations. Although this ad-hoc solution provides full control over both forward and backward transformations and can be realized using standard programming languages, it scales badly for non-trivial transformations and easily becomes expensive and error-prone: not only do we have to write two transformations instead of a single one, but the two transformations must be shown to satisfy the well-behavedness laws. In addition, it is a maintenance problem if a modification to one of the transformations requires a redefinition of the other transformation as well as a new proof of the laws.

To ease bidirectional programming and to enable maintainable bidirectional programming, it is preferable to write just a single program that can denote both transformations, which has motivated the following two methods:

- *Bidirectionalization of Forward Transformation.* This is to allow users to write the forward transformation in a familiar (unidirectional) programming language, and derive a suitable backward transformation through *bidirectionalization* techniques [Keller, 1986, Larson and Sheth, 1991, Xiong et al., 2007, Matsuda et al., 2007, Voigtländer, 2009, Hidaka et al., 2010].
- *Design of Domain-specific BX Languages.* This is to instruct users to write a program in a particular *bidirectional programming language* [Foster et al., 2007, Bohannon et al., 2006, 2008, Pacheco and Cunha, 2010, Hofmann et al., 2011, 2012, Pacheco et al., 2012], from which both transformations can be derived.

What both methods have in common is that one writes a forward transformation in a unidirectional language or a domain-specific BX language, and a corresponding good backward transformation can be automatically derived, which makes bidirectional programming easy and maintainable. Despite these advantages, there is an impractical assumption that

> for a forward transformation *get*, it is sufficient to derive a "suitable" *put* that can be combined to form a well-behaved bidirectional transformation.

In general a *get* function may not be injective, so there may exist many possible *put* functions that can be combined with *get* to form a bidirectional transformation. Moreover, the most "suitable" or "best" backward transformation may be hard to find and to justify. There is no clear consensus on the best requirements even for well-studied domains [Buneman et al., 2008].

To understand the inherent ambiguity in possible "putback" functions, consider, as an example, the following forward transformation.[1]

$$\begin{array}{l} \mathit{getEvens} :: [\mathit{Int}] \to [\mathit{Int}] \\ \mathit{getEvens}\ \mathit{ns} = \mathit{filter}\ \mathit{even}\ \mathit{ns} \end{array}$$

It returns those elements of a list of numbers which are even.

$$\mathit{evens}\ [2,3,5,6] = [2,6]$$

What is its corresponding *put*? If 6 is eliminated from the result, leaving the view list [2], there are many well-behaved ways of putting back this elimination to the source. For

[1] We use Haskell syntax for our examples as we discuss further in Section 3.1. The *filter* function is predefined and selects all elements from a list that satisfy the given predicate – in this case *even*, which is also predefined.

example, we could delete the number 6 from the source $[2, 3, 5, 6]$ to obtain the updated source $[2, 3, 5]$ or change 6 to n obtaining $[2, 3, 5, n]$ where n is an arbitrary odd number. Keller [1986] argues that many of these "putback" functions are incomparable, and that there is no reasonable approach to say which is the best.

It is this unavoidable ambiguity of *put* that makes bidirectional programming difficult to be used in practice due to possibly unpredictable behaviors, and that has led to the boom of current bidirectional frameworks that propose to answer the needs of particular bidirectional applications [Czarnecki et al., 2009, Hu et al., 2011].

1.3 Putback-based Bidirectional Programming

So far, bidirectional programming has been focused on writing the forward transformation (or a bidirectional program that resembles writing the forward transformation), and then trying to derive a suitable (but less predictable) backward transformation that embodies a specific way to solve the ambiguity of update translation (i.e., an update strategy) [Keller, 1986, Barbosa et al., 2010, Pacheco et al., 2012]. Nevertheless, whatever update strategy is given, it does not resolve the ambiguity problem because there is in general no best update strategy [Buneman et al., 2008].

In this paper, we argue that the update strategy should be considered from the start, and propose a novel putback-based approach to bidirectional programming: writing *put* and deriving *get* (i.e., specifying the intended backward transformation that best suits particular purposes, and deriving the forward transformation.) The new approach attains the advantages of writing a single program to specify a bidirectional transformation to enable easy maintenance. It also enjoys an important feature that, in sharp contrast to bidirectional programming based on *get* where *get* is not sufficient to determine *put*, bidirectional programming based on *put* has the potential to describe all intentions of bidirectional transformations, since there is only one *get* that can be combined with a given *put* to form a well-behaved bidirectional transformation.

There are two major difficulties in constructing a framework for putback-based bidirectional programming. One is to clarify sufficient and necessary conditions on the *put* function such that it can be used to describe all intentions of a well-behaved bidirectional transformation while guaranteeing existence and uniqueness of the corresponding *get* function. The other difficulty is how to automatically derive the unique *get* from *put* in practice. Our main technical contributions can be summarized as follows.

- We clarify necessary conditions on putback functions of well-behaved bidirectional transformations in Section 3 and use these conditions to characterize different classes of BXs in Section 4.

- In the same section, we formally prove that for a given *put* function that satisfies the identified necessary conditions the corresponding *get* function is unique in each considered class of BXs. Hence, the *get* function is redundant and can be derived automatically, retaining maintainability and full control over the update strategy.

- In Section 5, we describe a prototype implementation of deriving *get* from *put* in the functional logic programming language Curry [Hanus (editor), 2012] and

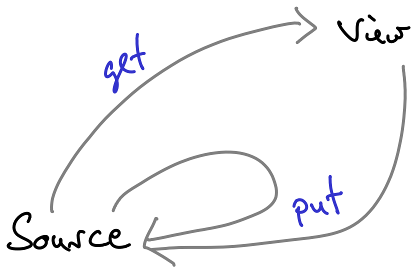

Figure 2: The GetPut law

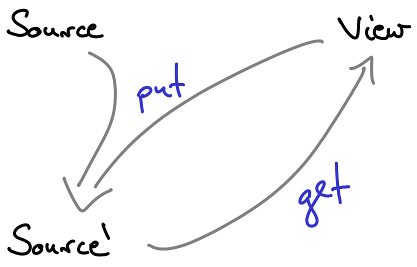

Figure 3: The PutGet law

demonstrate how to program in putback style using programs that implement different update strategies for transformations resembling database queries.

We begin in Section 2 by discussing the considered classes of BXs more formally and provide corresponding intuitions using examples that we also use to show the ambiguity of backward transformations for given forward transformations. We discuss related work in Section 6 and provide our conclusions together with possibilities for future work in Section 7.

2 Classes of Bidirectional Transformations

In this section we review different classes of bidirectional transformations, discussing their laws both equationally and intuitively. Example transformations in this section are minimalistic to highlight the essential differences between different classes of BXs.

2.1 Well-behaved BXs

Foster et al. [2007] call bidirectional transformations that satisfy the GetPut and PutGet laws *well-behaved*.

Definition 2.1 (GetPut law)**.** *The GetPut law describes a property of calling the forward function get before the backward function put.*

$$put\ s\ (get\ s) = s$$

It is depicted in Figure 2.

The put function should yield an unmodified source when passing the view obtained by get unchanged along with the original source. This property captures the intuition that if no view update takes place the source should not be updated either.

Definition 2.2 (PUTGET law)**.** *The* PUTGET *law describes a property of calling the backward function put before the forward function get.*

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

It is depicted in Figure 3.

When passing a source obtained by put to get then get should yield the same view that was passed to put. This property captures the intuition that view updates are reflected in the source type by put and can be observed by get.

The bidirectional transformation defined in Example 1.1 is well-behaved because it satisfies the GETPUT and PUTGET laws. To verify the GETPUT law, consider the following equations.

$$\begin{aligned} & \mathit{putFirst}\ (x, y)\ (\mathit{getFirst}\ (x, y)) \\ = \quad & \{ \text{ definition of } \mathit{getFirst} \ \} \\ & \mathit{putFirst}\ (x, y)\ x \\ = \quad & \{ \text{ definition of } \mathit{putFirst} \ \} \\ & (x, y) \end{aligned}$$

So, indeed, when we update the first component of a pair with its own first component, the pair does not change.

To verify the PUTGET law, consider the equations below.

$$\begin{aligned} & \mathit{getFirst}\ (\mathit{putFirst}\ (x, y)\ z) \\ = \quad & \{ \text{ definition of } \mathit{putFirst} \ \} \\ & \mathit{getFirst}\ (z, y) \\ = \quad & \{ \text{ definition of } \mathit{getFirst} \ \} \\ & z \end{aligned}$$

So, indeed, if we query the first component of a pair with an updated first component, we get back the value used for updating.

Example 2.3 (Change counter)**.** *As another example for a well-behaved bidirectional transformation, consider the combination of getFirst with the following backward function.*

$$\mathit{putFirstCount}\ (n, c)\ m = \textbf{if}\ n == m\ \textbf{then}\ (m, c)\ \textbf{else}\ (m, c + 1)$$

This function increments the second component of a pair whenever we change its first component.

To verify the GETPUT law, we can check the following equations.

$$\begin{aligned} & \mathit{putFirstCount}\ (n, c)\ (\mathit{getFirst}\ (n, c)) \\ = \quad & \{ \text{ definition of } \mathit{getFirst} \ \} \\ & \mathit{putFirstCount}\ (n, c)\ n \\ = \quad & \{ \text{ definition of } \mathit{putFirstCount} \ \} \end{aligned}$$

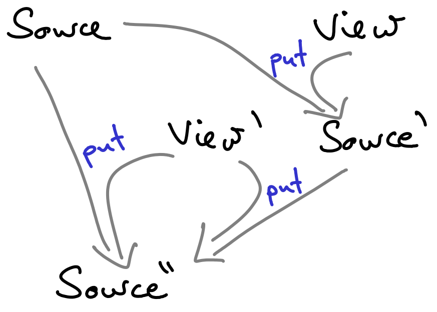

Figure 4: The PutPut law

$$
\begin{aligned}
& \textbf{if } n == n \textbf{ then } (n, c) \textbf{ else } (n, c+1) \\
= \quad & \{ \text{ reduction to } \textbf{then} \text{ branch } \} \\
& (n, c)
\end{aligned}
$$

Because we do not change the first component during the update, the counter is not incremented.

To verify the PutGet law, we can proceed as follows.

$$
\begin{aligned}
& \mathit{getFirst}\ (\mathit{putFirstCount}\ (n, c)\ m) \\
= \quad & \{ \text{ definition of } \mathit{putFirstCount} \} \\
& \mathit{getFirst}\ (\textbf{if } n == m \textbf{ then } (m, c) \textbf{ else } (m, c+1)) \\
= \quad & \{ \text{ reduction of } \mathit{getFirst} \text{ in both branches } \} \\
& m
\end{aligned}
$$

Regardless of whether the counter is incremented, the first component of an updated pair is the value used for updating.

2.2 Very well-behaved BXs

Foster et al. [2007] call well-behaved bidirectional transformations that also satisfy the PutPut law *very well-behaved.*

Definition 2.4 (PutPut law)**.** *The* PutPut *law describes a property of calling the backward function put twice with different views.*

$$put\ (put\ s\ v')\ v = put\ s\ v \qquad \text{PutPut}$$

It is depicted in Figure 4.

When put is called twice in a row then the result should be the same as the result of the second call with the updated source replaced by the original. This property captures the intuition that view updates are independent of each other.

The bidirectional transformation defined in Example 1.1 is very well-behaved, as can be verified by the following equations.

$$\begin{aligned}
& \mathit{putFirst}\ (\mathit{putFirst}\ (x, y)\ z_1)\ z_2 \\
= \quad & \{ \text{ definition of } \mathit{putFirst} \ \} \\
& \mathit{putFirst}\ (z_1, y)\ z_2 \\
= \quad & \{ \text{ definition of } \mathit{putFirst} \ \} \\
& (z_2, y) \\
= \quad & \{ \text{ definition of } \mathit{putFirst} \ \} \\
& \mathit{putFirst}\ (x, y)\ z_2
\end{aligned}$$

The first update (using the value z_1) does not influence the second update (using z_2) because updates completely overwrite the effect of previous updates.

The bidirectional transformation defined in Example 2.3 is not very well-behaved because the PutPut law is violated if two updates in a row change the first component of the pair more often than the second update alone.

$$\begin{aligned}
& \mathit{putFirstCount}\ (\mathit{putFirstCount}\ (42, 0)\ 43)\ 42 \\
= \quad & \{ \text{ definition of } \mathit{putFirstCount} \ \} \\
& \mathit{putFirstCount}\ (43, 1)\ 42 \\
= \quad & \{ \text{ definition of } \mathit{putFirstCount} \ \} \\
& (42, 2) \\
\neq \quad & \\
& (42, 0) \\
= \quad & \{ \text{ definition of } \mathit{putFirstCount} \ \} \\
& \mathit{putFirstCount}\ (42, 0)\ 42
\end{aligned}$$

Different updates are not independent because their effect on the counter cannot be overwritten.

Example 2.5 (Maintaining difference)**.** *As a second example of a bidirectional transformation that is very well-behaved consider the function getFirst together with the following backward function that maintains the original difference of the components of a pair when updating the first component.*

$$\mathit{putFirstDiff}\ (x, y)\ z = (z, z + y - x)$$

When updating the first component, the second is also changed such that the difference between the components remains the same. Here are some example calls to clarify this behavior.

$$\begin{aligned}
& \mathit{putFirstDiff}\ (1, 2)\ 3 = (3, 3 + 2 - 1) = (3, 4) \\
& \mathit{putFirstDiff}\ (1, 1)\ 1 = (1, 1 + 1 - 1) = (1, 1) \\
& \mathit{putFirstDiff}\ (3, 2)\ 1 = (1, 1 + 2 - 3) = (1, 0)
\end{aligned}$$

The following equations show that the GetPut, PutGet, and PutPut laws hold, respectively, for the bidirectional transformation defined in Example 2.5.

$$
\begin{aligned}
& \mathit{putFirstDiff}\ (x, y)\ (\mathit{getFirst}\ (x, y)) \\
= \quad & \{ \text{ definition of } \mathit{getFirst} \ \} \\
& \mathit{putFirstDiff}\ (x, y)\ x \\
= \quad & \{ \text{ definition of } \mathit{putFirstDiff} \ \} \\
& (x, x + y - x) \\
= \quad & \{ \text{ associativity and inverse of addition } \} \\
& (x, y)
\end{aligned}
$$

$$
\begin{aligned}
& \mathit{getFirst}\ (\mathit{putFirstDiff}\ (x, y)\ z) \\
= \quad & \{ \text{ definition of } \mathit{putFirstDiff} \ \} \\
& \mathit{getFirst}\ (z, z + y - x) \\
= \quad & \{ \text{ definition of } \mathit{getFirst} \ \} \\
& z
\end{aligned}
$$

$$
\begin{aligned}
& \mathit{putFirstDiff}\ (\mathit{putFirstDiff}\ (x, y)\ z_1)\ z_2 \\
= \quad & \{ \text{ definition of } \mathit{putFirstDiff} \ \} \\
& \mathit{putFirstDiff}\ (z_1, z_1 + y - x)\ z_2 \\
= \quad & \{ \text{ definition of } \mathit{putFirstDiff} \ \} \\
& (z_2, z_2 + z_1 + y - x - z_1) \\
= \quad & \{ \text{ associativity and inverse of addition } \} \\
& (z_2, z_2 + y - x) \\
= \quad & \{ \text{ definition of } \mathit{putFirstDiff} \ \} \\
& \mathit{putFirstDiff}\ (x, y)\ z_2
\end{aligned}
$$

Example 2.5 follows a general pattern to construct very well-behaved bidirectional transformations by maintaining a constant complement [Bancilhon and Spyratos, 1981] of the view in invocations of the *put* function. In fact, Foster et al. [2007] prove that view updating under constant complement captures every very well-behaved BX.

2.3 Bijective BXs

Foster et al. [2007] call bidirectional transformations that satisfy the StrongGetPut law in addition to the PutGet law *bijective*.

Definition 2.6 (StrongGetPut law)**.** *The* StrongGetPut *law is a stronger version of the* GetPut *law for well-behaved bidirectional transformations.*

$$\mathit{put}\ s'\ (\mathit{get}\ s) = s \qquad\qquad \textsc{StrongGetPut}$$

When put is called on the result of get it should yield the same source that was given to get initially, regardless of what source is used for the update. Together with the PutGet *law (see Definition 2.2) the* StrongGetPut *law captures the intuition that there is a one-to-one correspondence between sources and views implemented by the get and put functions.*

The bidirectional transformation defined in Example 1.1 is not bijective because there is no one-to-one correspondence between pairs and their first components. To verify that the StrongGetPut law is violated, consider the following inequality.

$$
\begin{array}{ll}
 & \mathit{putFirst}\ (1,2)\ (\mathit{getFirst}\ (3,4)) \\
= & \quad \{\ \text{definition of } \mathit{getFirst}\ \} \\
 & \mathit{putFirst}\ (1,2)\ 3 \\
= & \quad \{\ \text{definition of } \mathit{putFirst}\ \} \\
 & (3,2) \\
\neq & \\
 & (3,4)
\end{array}
$$

Unlike (very) well-behaved BXs, bijective BXs do not allow to discard information when computing a view from a source. Examples 2.3 and 2.5 also do not define bijective bidirectional transformations because they use the same forward function as Example 1.1 which discards information.

3 Preliminary observations on putback functions

We now introduce notation and review mathematical concepts that play a role when we characterize the different classes of BXs in Section 4. While doing so, we observe necessary conditions on *put* functions implied by laws for bidirectional transformations.

3.1 Currying, point-free style, and infix operator sections

We denote function application by juxtaposition (as in the functional programming language Haskell [Marlow (editor), 2010]). For example, the application of a function *get* to a source argument s is written as follows.

$$\mathit{get}\ s$$

We write functions with multiple arguments in so-called curried style, i.e., instead of taking (a tuple of) multiple arguments directly, multi-argument functions take one argument and yield a function for the remaining arguments. As is conventional, function application associates to the left and, hence, the application of a function *put* to a source argument s and a view argument v is written as follows.

$$\mathit{put}\ s\ v = (\mathit{put}\ s)\ v$$

The higher-order functions *curry* and *uncurry* translate between the curried and uncurried versions of a function.

$$
\begin{array}{ll}
\mathit{curry}\ f\ x\ y & = f\ (x,y) \\
\mathit{uncurry}\ f\ (x,y) & = f\ x\ y
\end{array}
$$

For example, we can obtain the uncurried version of a *put* function by calling *uncurry* and then pass the arguments as a pair.

$uncurry\ put\ (s, v) = put\ s\ v$

Sometimes, we give type annotations for functions, again in Haskell notation. If sources have type S and views are of type V then curried and uncurried *put* functions have the following types, respectively.

$$\begin{aligned} &put && :: S \to (V \to S) \\ &uncurry\ put && :: (S, V) \to S \end{aligned}$$

As is conventional, the function-type constructor associates to the right, so the type $S \to (V \to S)$ can be written $S \to V \to S$, for short.

We write *id* for the identity function on arbitrary types. To highlight a specific argument (and result) type of *id* we sometimes write it as a subscript, such as id_S for the identity function on a type S.

We write a dot to denote function composition. For example, the composition of a partially applied *put* function with a *get* function, applying *get* to the result of *put s*, is written as follows.

$$get \cdot put\ s = \lambda v \to get\ (put\ s\ v)$$

Lambda abstractions, like the one above, define anonymous functions.

Binary functions, such as *put*, can be enclosed in backquotes to use them as infix operators. Furthermore, partial applications of infix operators can be formed for the first and second argument using parenthesized *infix operator sections* where only the left (i.e., first) or right (i.e., second) argument is provided. The following examples, demonstrate the use of such notation.

$$\begin{aligned} &s\ \text{`}put\text{`}\ v = put\ s\ v \\ &(s\text{`}put\text{`}) = \lambda v \to put\ s\ v = put\ s \\ &(\text{`}put\text{`}v) = \lambda s \to put\ s\ v \end{aligned}$$

Proposition 3.1 (Point-free laws for BXs)**.** *We can now rephrase some of the laws for bidirectional transformations reviewed in Section 2 in so-called point-free style, i.e., without mentioning some arguments of functions.*

1. *The* PutGet *law (cf. Definition 2.2) states the following equation for all sources s.*

   $$get \cdot put\ s = id$$

2. *The* PutPut *law (cf. Definition 2.4) states the following equation for all views v and v'.*

   $$(\text{`}put\text{`}v) \cdot (\text{`}put\text{`}v') = (\text{`}put\text{`}v)$$

3. *The* StrongGetPut *law (cf. Definition 2.6) states the following equation for all sources s.*

   $$put\ s \cdot get = id$$

$\square$

Reformulating laws in point-free style will help identify necessary conditions on putback functions using standard mathematical terminology.

3.2 Mathematical propositions

This subsection recalls basic mathematical concepts used subsequently and relates them to the previously introduced notation.

Definition 3.2 (Injectivity and left inverse)**.** *A function $f :: A \to B$ is* injective *if and only if there is a function $g :: B \to A$ such that $g \cdot f = id_A$. In this case, g is called* left inverse *of f.*

Intuitively, an injective function maps different arguments to different results. As a consequence, the result type is "at least as big as" the argument type.

Definition 3.3 (Surjectivity and right inverse)**.** *A function $f :: A \to B$ is* surjective on *B if and only if there is a function $g :: B \to A$ such that $f \cdot g = id_B$. In this case, g is called* right inverse *of f.*

Intuitively, a surjective function yields every value in its result type for some argument. As a consequence, the argument type is "at least as big as" the result type.

Proposition 3.4 (Injectivity and Surjectivity in BXs)**.** *Certain laws for bidirectional transformations impose injectivity and surjectivity requirements on bidirectional transformations.*

1. *The* PutGet *law (cf. Proposition 3.1) implies that get is surjective on the view type V and "put s" is injective for all sources s. If the* PutGet *law holds then, for all sources s, get is a left inverse of "put s" and "put s" is a right inverse of get.*

2. *The* StrongGetPut *law (cf. Proposition 3.1) implies that get is injective and "put s" is surjective on the source type S for all sources s. If the* StrongGetPut *law holds then, for all sources s, get is a right inverse of "put s" and "put s" is a left inverse of get.*

$\square$

Definition 3.5 (Bijectivity and inverse)**.** *A function f is* bijective *if and only if it is injective and surjective. In this case, there is exactly one left inverse g of f. It is also a right inverse of f and, therefore, called* inverse, *for short. We write f^{-1} for the inverse of a bijective function f.*

Intuitively, a bijective function defines a one-to-one correspondence between its argument and result type. As a consequence, both have the same cardinality.

Proposition 3.4 justifies why bidirectional transformations that satisfy both the PutGet as well as the StrongGetPut law are called bijective, because in this case *get* is bijective and *put s* is its inverse for all sources *s*. This shows that *put* functions in bijective bidirectional transformations can ignore their source argument, such that they trivially satisfy the PutPut law (cf. Definition 2.4). Therefore, every bijective bidirectional transformation is also very well behaved.

It is worth considering the differences between the properties defined in Definitions 3.2–3.5 for multi-argument functions in curried and uncurried style. For example, for a given *put* function we can distinguish the following notions of injectivity.

1. *put* is injective
2. *put s* is injective for all source values *s*
3. *uncurry put* is injective

We can build an intuition for the differences between these properties by considering the cardinalities of source and view types, say $|S|$ and $|V|$ respectively.

The cardinality of the argument type S of *put* is $|S|$ and the cardinality of its result type $V \to S$ is $|S|^{|V|}$. Therefore, it is easy to give an injective *put* function. In fact, the constant *put* function defined as follows is an example.

$$put\ s\ v = s$$

On the other hand, the argument type (S, V) of *uncurry put* has cardinality $|S| \cdot |V|$ and its result type S has cardinality $|S|$. So, for finite source and view types with $|V| > 1$ there is no *put* function such that *uncurry put* is injective.

Property 2 lies in-between properties 1 and 3. For source types that are "at least as big as" the view type, there are *put* functions that satisfy property 2. The constant *put* function, however, is a counter example because all views are mapped to the same source.

We can make similar observations for surjectivity instead of injectivity of multi-argument functions.

Proposition 3.6 (Surjectivity of *uncurry put*)**.** *The* GetPut *law (cf. Definition 2.1) implies that "uncurry put" is surjective on the source type.*

Proof. If the GetPut law holds then the function "$p^r = \lambda s \to (s, get\ s)$" is a right inverse of "*uncurry put*":

$$\begin{array}{ll} & uncurry\ put\ (p^r\ s) \\ = & \{\ \text{definition of } p^r\ \} \\ & uncurry\ put\ (s, get\ s) \\ = & \{\ \text{definition of } uncurry\ \} \\ & put\ s\ (get\ s) \end{array}$$

$$
\begin{array}{ll}
= & \{\ \textsc{GetPut law}\ \} \\
& s
\end{array}
$$

□

Our final mathematical concept that turns out helpful for characterizing the different classes of BXs introduced in Section 2 is idempotence.

Definition 3.7 (Idempotence). *A function $f :: A \to A$ is called* idempotent *if the following equation holds.*

$$f \cdot f = f$$

This notion is useful to observe that well-behaved BXs satisfy a weaker version of the PUTPUT law (cf. Proposition 3.1).

Proposition 3.8 (Idempotence of (‘*put*‘*v*)). *In a well-behaved bidirectional transformation, i.e., which satisfies the* GETPUT *and* PUTGET *laws, the function* (‘*put*‘*v*) *is idempotent for all views v. This idempotence requirement can also be expressed as the following equation, called* PUTTWICE *law [Foster, 2009].*

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

Proof. The following equations use the PUTGET and GETPUT laws to derive the claimed idempotence requirement.

$$
\begin{array}{ll}
& ((`put`v) \cdot (`put`v))\ s \\
= & \{\ \text{function composition}\ \} \\
& (s\ `put`\ v)\ `put`\ v \\
= & \{\ \text{prefix notation}\ \} \\
& put\ (put\ s\ v)\ v \\
= & \{\ \textsc{PutGet law}\ \} \\
& put\ (put\ s\ v)\ (get\ (put\ s\ v)) \\
= & \{\ \textsc{GetPut law}\ \} \\
& put\ s\ v \\
= & \{\ \text{infix operator section}\ \} \\
& (`put`v)\ s
\end{array}
$$

□

The PUTPUT law is stronger than idempotence of (‘*put*‘*v*) because it requires a similar equation for compositions of *put* applied to different views v and v' instead of compositions of *put* applied to the same view. Example 2.3 shows that the PUTPUT law is stronger than idempotence of (‘*put*‘*v*) because the bidirectional transformation defined there is well-behaved, but not very.

4 Characterizing BXs in terms of the putback function

We now characterize different classes of BXs introduced in Section 2 based on necessary conditions observed in Section 3. From this characterization we will be able to identify exactly, for each introduced class of BXs, which part of the definition of a bidirectional transformation is redundant due to the corresponding laws.

As a preliminary observation, note that for bijective BXs the *put* function is uniquely determined by the *get* function and vice-versa. We can define

$$put\ s = get^{-1}$$

for all sources s to derive a unique *put* from *get*. We can also define

$$get\ s = (put\ s)^{-1}\ s$$

to derive a unique *get* from *put*.

For (very) well-behaved BXs the *get* function does not give rise to a unique *put* function. Examples 1.1 and 2.5 are two different very well-behaved BXs with the same *put* function. Hence, (very) well-behaved BXs cannot be specified completely by only providing a *get* function.

In the remainder of this section, we show that (very) well-behaved BXs can be specified completely by giving their *put* function. No additional conditions on *put* functions are required apart from necessary conditions observed in Section 3.

4.1 Characterizing well-behaved BXs

Our first theorem shows that we can replace each of the defining laws for well-behaved BXs by necessary conditions observed previously.

Theorem 4.1 (Characterizing well-behaved BXs). *The following propositions are equivalent.*

1. *The* GetPut *and* PutGet *laws hold.*
2. *The* GetPut *law holds, (‘put‘v) is idempotent for all views v, and “put s” is injective for all sources s.*
3. *The* PutGet *law holds, (‘put‘v) is idempotent for all views v, and “uncurry put” is surjective on the source type.*

Proof. We show the implications $1 \Rightarrow 2 \Rightarrow 1 \Rightarrow 3 \Rightarrow 1$.

$1 \Rightarrow 2$ The idempotence requirement follows from Proposition 3.8. The injectivity requirement is a direct consequence of the PutGet law (cf. Proposition 3.4).

$2 \Rightarrow 1$ To conclude the PutGet law

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

for all sources s and views v, let p^l be a left inverse of $put\ (put\ s\ v)$. Then, verify the following equations using the GetPut law and Proposition 3.8.

$$
\begin{array}{ll}
 & get\ (put\ s\ v) \\
= & \{\ \text{definition of } id\ \} \\
 & id\ (get\ (put\ s\ v)) \\
= & \{\ p^l \text{ is left inverse of } put\ (put\ s\ v)\ \} \\
 & p^l\ (put\ (put\ s\ v)\ (get\ (put\ s\ v))) \\
= & \{\ \text{GetPut law}\ \} \\
 & p^l\ (put\ s\ v) \\
= & \{\ \text{idempotence of } (`put`v), \text{Proposition 3.8}\ \} \\
 & p^l\ (put\ (put\ s\ v)\ v) \\
= & \{\ p^l \text{ is left inverse of } put\ (put\ s\ v)\ \} \\
 & id\ v \\
= & \{\ \text{definition of } id\ \} \\
 & v
\end{array}
$$

$1 \Rightarrow 3$ The idempotence requirement follows from Proposition 3.8. Surjectivity of "*uncurry put*" follows from Proposition 3.6.

$3 \Rightarrow 1$ To conclude the GetPut law, let p^r be a right inverse of "*uncurry put*", and for a source s define "$(s', v) = p^r\ s$" such that "$put\ s'\ v = s$". Then, verify the following equations using the PutGet law and Proposition 3.8.

$$
\begin{array}{ll}
 & put\ s\ (get\ s) \\
= & \{\ put\ s'\ v = s\ \} \\
 & put\ (put\ s'\ v)\ (get\ (put\ s'\ v)) \\
= & \{\ \text{PutGet law}\ \} \\
 & put\ (put\ s'\ v)\ v \\
= & \{\ \text{idempotence of } (`put`v), \text{Proposition 3.8}\ \} \\
 & put\ s'\ v \\
= & \{\ put\ s'\ v = s\ \} \\
 & s
\end{array}
$$

□

We cannot replace both defining laws for well-behaved BXs by the employed necessary conditions on the *put* function without losing the connection to the *get* function. However, those necessary conditions on *put* functions *are* sufficient in the sense that they give rise to a unique *get* function such that the resulting bidirectional transformation is well-behaved. The following Theorem 4.2 shows that *put* functions satisfying the necessary conditions used in Theorem 4.1 characterize well-behaved BXs, i.e., well-behaved BXs are uniquely determined by their putback function.

Theorem 4.2 (Uniqueness of *get* for well-behaved *put*)**.** *Assume a put function that satisfies all of the following propositions.*

1. *(‘put‘v) is idempotent for all views v.*
2. *“put s” is injective for all sources s.*
3. *“uncurry put” is surjective on the source type.*

Then the following propositions are also satisfied.

(a) For every source s there is exactly one view v such that put s v = s.

(b) There is exactly one get function such that the resulting BX is well-behaved.

Proof. (a) Regarding the existence of v, choose for all s a source s' and a view v such that $s = put\ s'\ v$ according to (3). Then, the following equations hold because of Proposition 3.8.

$$\begin{array}{ll} & put\ s\ v \\ = & \{\ s = put\ s'\ v\ \} \\ & put\ (put\ s'\ v)\ v \\ = & \{\ \text{idempotence of } (`put`v),\ \text{Proposition 3.8}\ \} \\ & put\ s'\ v \\ = & \{\ s = put\ s'\ v\ \} \\ & s \end{array}$$

A view v satisfying the equation $put\ s\ v = s$ is unique because “*put s*” is injective according to (2).

(b) Regarding the existence of *get* define

$$get\ s = v, with\ v\ such\ that\ s = put\ s\ v$$

according to (a). Then, we can verify the GetPut law as follows.

$$\begin{array}{ll} & put\ s\ (get\ s) \\ = & \{\ \text{definition of } get\ \} \\ & put\ s\ v, with\ v\ such\ that\ s = put\ s\ v \\ = & \{\ s = put\ s\ v\ \} \\ & s \end{array}$$

From the second proposition of Theorem 4.1 we can now conclude that the resulting BX is well-behaved.

Regarding the uniqueness of *get*, consider get' such that the resulting BX is well-behaved. Then, we can observe the following equations making use of (a) and the PutGet law.

$$
\begin{array}{ll}
 & get'\ s \\
= & \{\ \text{proposition } (a)\ \} \\
 & get'\ (put\ s\ v), with\ v\ such\ that\ s = put\ s\ v \\
= & \{\ \text{PUTGET law}\ \} \\
 & v, with\ v\ such\ that\ s = put\ s\ v \\
= & \{\ \text{definition of } get\ \} \\
 & get\ s
\end{array}
$$

□

This result shows that well-behaved BXs are characterized by their putback function, i.e., the *get* function is redundant for the purpose of specification.

4.2 Characterizing very well-behaved BXs

Very well-behaved BXs differ from well-behaved BXs only in the PUTPUT law (see Definition 2.4) which (as we have observed after Proposition 3.8) is a stronger version of the idempotence requirement on (‘*put*‘v) for all views v.

As a consequence, the following characterization of very well-behaved BXs is a direct consequence of Theorem 4.1.

Corollary 4.3 (Characterizing very well-behaved BXs)**.** *The following propositions are equivalent.*

1. *The* GETPUT*,* PUTGET*, and* PUTPUT *laws hold.*
2. *The* GETPUT *and* PUTPUT *laws hold and "put s" is injective for all sources s.*
3. *The* PUTGET *and* PUTPUT *laws hold and "uncurry put" is surjective on the source type.*

Proof. Direct consequence of Theorem 4.1 because the PUTPUT law (cf. Proposition 3.1) implies idempotence of (‘*put*‘v) for all views v. □

We get a similar result regarding the uniqueness of the *get* function for a given *put* function of a very well-behaved BX as we observed for well-behaved BXs.

Corollary 4.4 (Uniqueness of *get* for very well-behaved *put*)**.** *Assume a put function that satisfies all of the following propositions.*

1. *The* PUTPUT *law holds.*
2. *"put s" is injective for all sources s.*
3. *"uncurry put" is surjective on the source type.*

Then there is exactly one get function such that the resulting BX is very well-behaved.

Proof. The propositions imply those of Theorem 4.2 so there is a unique *get* function such that the resulting BX is well-behaved. It is also very well-behaved because the *put* function satisfies PUTPUT according to proposition (1). □

4.3 Partial BXs

So far, we have assumed *get* and *put* to be functions —in the mathematical sense— between a source type S and a view type V. This entails, specifically, that *get* and *put* are both total, i.e., yield a value for every argument in their respective domains. In the case of well-behaved BXs, totality of *get* means surjectivity of *uncurry put* and totality of *put* means surjectivity of *get*.

While it is possible, in principle, to define precise types that match the domain and range of arbitrary *get* and *put* functions exactly, in practice, it is often convenient to allow the source and view types of bidirectional transformations be larger and define *get* or *put* as partial functions between these larger types [Pacheco, 2012].

Example 4.5 (Partial BX to access the head of a non-empty list)**.** *As an example for a partial bidirectional transformation, consider the following definitions.*

$$\begin{array}{ll} \mathit{getHead}\ (x : \_\ ) & = x \\ \mathit{putHead}\ (\_ : \mathit{xs})\ x & = x : \mathit{xs} \end{array}$$

In Haskell, $(x : \mathit{xs})$ *denotes a list containing at least one element – x being the first and xs containing all remaining ones.*

Here are some example calls, that demonstrate the behavior of the forward and backward functions using an alternative list notation.

$$\begin{array}{ll} \mathit{getHead}\ [1,2,3] = 1 & \\ \mathit{getHead}\ [\,] & \text{-- } \mathit{fails} \\ \mathit{putHead}\ [1,2]\ 3\ = [3,2] & \\ \mathit{putHead}\ [\,]\quad 1 & \text{-- } \mathit{fails} \end{array}$$

If we define the source type for this transformation to be the type $[\mathit{Elem}]$ of lists of elements of type *Elem* and the view type to be *Elem* then *getHead* and *putHead* do not form a well-behaved BX. The GetPut and PutGet laws are violated if we use the empty list as a source value.

$$\begin{array}{l} \mathit{putHead}\ [\,]\ (\mathit{getHead}\ [\,]) \neq [\,] \\ \mathit{getHead}\ (\mathit{putHead}\ [\,]\ 1)\ \neq 1 \end{array}$$

Also, note that one of the conditions on the *put* function used in Theorem 4.2 is violated: "*uncurry putHead*" is not surjective because $\mathit{put}\ s\ v \neq [\,]$ for all s and v.

In Example 4.5, however, there is an easy way out. If we define the source type to be the type of *non-empty* lists then *putHead* does satisfy all conditions of Theorem 4.2 and *getHead* is the unique forward function forming a well-behaved BX with *putHead*.

In general, we may assume such precise types even if we do not express them in a programming language. For example, in Section 5 we will define a bidirectional transformation whose view type consists of people from Tokyo. While we do not model the type of people from Tokyo in our programming language of choice, we can still use

Theorem 4.2 to derive a corresponding forward function automatically via the same reasoning we applied to Example 4.5.

Alternatively, we could reformulate our definitions such that *get* and *put* are partial functions and bidirectional laws are partial equalities. This relaxation still preserves all the above theorems (excluding surjectivity of *get* and of *uncurry put*) as long as *get* and *put* are *safe* [Pacheco, 2012], in the sense that *get* is defined for the image of *put* and *put* is defined for the image of *get*. For space and readability reasons, we refrain from restating our results with partial functions.[2]

5 Put-based programming of BXs

Our main result, Theorem 4.2, states that in a well-behaved bidirectional transformation the definition of *get* is redundant given a definition of *put* that satisfies certain properties necessary for well-behavedness. In this section, we argue for specifying a well-behaved BX by defining a *put* function. We show how to use the functional-logic programming language Curry [Hanus (editor), 2012] to derive a corresponding *get* function automatically and show, using several examples, how to define *put* functions for well-behaved BXs.

5.1 Using Curry to derive *get* from *put*

We can use the built-in search facilities of the functional-logic programming language Curry to derive the *get* function of a well-behaved bidirectional transformation from their *put* function automatically. While search is probably not the most efficient way to obtain *get* from *put* it is sufficient for the demonstration purposes of this section.

Syntactically, Curry is an extension of basic Haskell. Semantically, an important difference is the handling of pattern matching – especially in presence of multiple rules. In Curry, pattern-match failure is handled silently and more than one defining rule of a function (or more accurately: operation)[3] may be applied nondeterministically. For example, the following program splits a given list in two parts at an arbitrary position.

```
split xs       = ([], xs)
split (x : xs) = (x : ys, zs)
  where
    (ys, zs) = split xs
```

The defining rules of *split* are overlapping but, unlike in Haskell, not only the first matching rule is applied but all matching rules are applied nondeterministically. For example, there are three possible results of the call *split* [1, 2]:

```
([1,2], [])
([1], [2])
([], [1,2])
```

[2]An Alloy [Jackson, 2012] model with precise partial definitions and a lightweight proof of Theorem 4.2 can be downloaded from `http://www.di.uminho.pt/~hpacheco/publications/Put.als`.

[3]Because of nondeterminism, the input-output relation of a Curry operation does not necessarily describe a function.

Implementations of Curry usually allow to observe all nondeterministic results interactively and are free to present them in an arbitrary order.

An alternative way to define *split* is by constraining free variables. Instead of encoding nondeterminism using overlapping rules, we can also induce search by calculating with unknown information. Here is an alternative definition of *split* following this approach.

```
split xs | xs == ys ++ zs = (ys, zs)
  where
    ys, zs free
```

This definition expresses in a guard that concatenating two unknown lists *ys* and *zs* using the predefined operation "++" should yield the argument list *xs*. The Curry implementation searches for instantiations of *ys* and *zs* that satisfy the guard and returns them as result of *split*. As there are, generally, multiple ways to instantiate *ys* and *zs* to satisfy the guard, the result of split is nondeterministic like with the previous definition.[4]

In order to define the *get* function of a bidirectional transformation based on a *put* function, we can use the same programming style as in the second definition of *split*. For convenience, we first define a type for bidirectional transformations between a source type s and a view type v.

```
type BX s v = s → v → s
```

The type *BX s v* defines bidirectional transformations as their *put* function. We provide a function *put* that just calls this function.

```
put :: BX s v → s → v → s
put bx s v = bx s v
```

The function *get* is defined based on *put* using the constraint given in Theorem 4.2.

```
get :: BX s v → s → v
get bx s | put bx s v == s = v
  where
    v free
```

These definitions allow to *define* bidirectional transformations by giving only the putback function but to *use* them in both directions. In the remainder of this section we show several examples of defining and using bidirectional transformations in this putback style.

5.2 Record field access

The most basic bidirectional transformations access fields of records similarly to Example 1.1. To demonstrate record field access, we define a type for people.

[4]Depending on the Curry implementation, constraint equalities need to be specified using a different operator. We use the Kiel Curry System KiCS2 [KiCS2 developers, 2012] which allows to apply standard equality to free variables.

```
data Person = Person Name City
data Name   = Hugo  | Sebastian | Zhenjiang
data City   = Braga | Kiel      | Tokyo
```

Example 5.1. *We define bidirectional transformations name and city to access the name and associated city of a person, respectively.*

```
name :: BX Person Name
name (Person _ c) n = Person n c
city :: BX Person City
city (Person n _) c = Person n c
```

Both definitions specify a bidirectional transformation by its putback function and we can issue calls in the Curry system KiCS2 to check that the derived forward function works as expected.

```
KiCS2⟩ get name (Person Hugo Braga)
Hugo
KiCS2⟩ get city (Person Zhenjiang Tokyo)
Tokyo
```

Of course, we can also call the putback function —directly or indirectly using *put*— as the following examples demonstrate.

```
KiCS2⟩ put city (Person Sebastian Tokyo) Kiel
Person Sebastian Kiel
KiCS2⟩ city (Person Sebastian Tokyo) Kiel
Person Sebastian Kiel
```

We can also use these bidirectional transformations in other Curry functions. For example, the following predicate checks whether a person is from a given city.

```
isFrom :: City → Person → Bool
isFrom c p = c == get city p
```

In the following, we define bidirectional transformations in analogy to database view updates using a database of people.

5.3 Database view updating

Historically, database view updating is a primary source of motivation for researching bidirectional transformations. We now demonstrate how different view update strategies described in the literature can be expressed in our *put*-based framework for bidirectional programming.

For the sake of simplicity, we regard database tables as sets of rows represented as lists sorted by a key identifying rows uniquely. For example, the following is a database of people where each person is identified by their name.

```
people     = [hugo, sebastian, zhenjiang]
hugo       = Person Hugo Braga
sebastian  = Person Sebastian Tokyo
zhenjiang  = Person Zhenjiang Tokyo
```

In order to update this database of people, the following function *mergePeople* will be helpful. It takes two tables of people (sorted by name) and merges them into a single (sorted) table of people. If entries with the same key are present in both tables then the entry in the second table overwrites the entry in the first.

```
mergePeople :: [Person] → [Person] → [Person]
mergePeople old new = merge (sorted old) (sorted new)
  where
    merge [ ] ps = ps
    merge (p : ps) [ ] = p : ps
    merge (p : ps) (q : qs)
      | get name p < get name q
        = p : merge ps (q : qs)
      | get name p == get name q
        = q : merge ps qs
      | get name p > get name q
        = q : merge (p : ps) qs
    sorted [ ]      = [ ]
    sorted (p : ps) = ascending p ps
    ascending p [ ] = [p]
    ascending p (q : qs)
      | get name p < get name q
        = p : ascending q qs
```

The function *mergePeople* restricts the tables passed as arguments to be sorted using the partial function *sorted* that ensures that its argument is a sorted list of people. The function *sorted* is the identity function on sorted tables but fails on lists of people that are not sorted by name (in mathematical terms, a coreflexive relation that is a subset of the identity relation).

Now, consider a database query that selects all people from a certain city. For example, selecting all people from *Tokyo* from the *people* database defined above would result in the following view of this database.

```
[Person Sebastian Tokyo, Person Zhenjiang Tokyo]
```

When adding new people to this view, reflecting it in the original database is straightforward: if a person with the same name already exists then change their city to *Tokyo*; if there is no person with that name in the original database then add it.[5] For deletion, however, there is no straightforward update strategy. When deleting a person from the view, we can either

1. delete it from the original database or
2. change their city to a city different from *Tokyo*.

Keller [1986] argues that both strategies may be reasonable depending on context, so a system computing an update strategy automatically solely based on the definition of the view function is insufficient. Using *put*-based bidirectional programming, both strategies above can be expressed in a straightforward way.

5.3.1 Reflecting deletions via deletions

The first strategy —deleting people from the original database that are not present in the updated view— can be implemented as follows:

- first, delete all people from the given city from the original database,
- then update the result by merging all people from the updated view.

Example 5.2 (Reflecting deletions via deletions)**.** *The following function peopleFrom implements a bidirectional transformation using the strategy described above.*[6]

```
peopleFrom :: City → BX [Person] [Person]
peopleFrom c source view =
    let elsewhere = filter (not · isFrom c) source
    in  mergePeople elsewhere (map ensureCity view)
  where
    ensureCity q | isFrom c q = q
```

The function *peopleFrom* uses *mergePeople* to merge the list *elsewhere* of people not from the given city in the original *source* with the list of people from the updated *view*. The function *mergePeople* ensures that both lists are sorted. Additionally, *peopleFrom* restricts updated views using the local function *ensureCity* which is a partial identity function on people from the given city. Restricting updated views is important for maintaining the injectivity requirement of Theorem 4.2, as we discuss below. As a

[5]In fact, these are not the only well-behaved ways to handle additions in a view. Changing the city of an existing person could be distinguished from deleting it and adding a new person if people would have additional properties besides their name and city. Also, adding people not from *Tokyo* would not violate well-behavedness if only performed when a person in the view is not present in the source. While our approach allows to define all such updates, we restrict ourselves to more reasonable strategies trying to keep updates minimal, which is consistent with update translation strategies used for database view updating.

[6]The predefined function *map* applies a given function to each element of a list.

consequence of this restriction, it is not possible to move people to a different city using this update strategy.

Conceptually, *peopleFrom c* is a bidirectional transformation between the source type of sorted lists of people and the view type of sorted lists of people from the city c. Note that specifying the view type in the type system would require a dependently typed programming language, i.e., one where the types of expressions can depend on the values of others. Curry is not dependently typed, so we define *peopleFrom c* as a partial function instead.

To verify that the *get*-function derived by our framework is unique and the resulting transformation is well-behaved, we need to check the following three conditions:

1. $\mathit{peopleFrom}\ c\ (\mathit{peopleFrom}\ c\ s\ v)\ v = \mathit{peopleFrom}\ c\ s\ v$ for all cities c, sorted lists s of people, and sorted lists v of people from the city c,

2. $\mathit{peopleFrom}\ c\ s$ is injective for all cities c and sorted lists of people s, and

3. $\mathit{uncurry}\ (\mathit{peopleFrom}\ c)$ is surjective on sorted lists of people for all cities c.

The first condition is satisfied because the second application of *peopleFrom c* will remove the updated people and then re-add them, so updating twice using the same view is the same as updating only once.

The second condition is satisfied because *peopleFrom c* restricts updated views to people from the given city and is not defined for other views. Without this restriction, views that contain people from the original database not from the city c would map to the same updated source as views not containing them, violating the injectivity requirement. By deleting all people from the city c from the original source and ensuring that people in the updated view are from the city c, *peopleFrom c* ensures that updated views are mapped uniquely to updated sources.

The third requirement is satisfied because every database of people can be obtained using *peopleFrom c* by passing it as original source together with a view that includes all its people from the city c.[7]

Together, these conditions ensure that the *get* function derived for *peopleFrom c* is unique and the resulting BX is well-behaved. The following calls demonstrate the behavior of the derived view function and the defined update strategy.

```
KiCS2⟩ get (peopleFrom Tokyo) people
[Person Sebastian Tokyo, Person Zhenjiang Tokyo]
KiCS2⟩ put (peopleFrom Tokyo) people [zhenjiang]
[Person Hugo Braga, Person Zhenjiang Tokyo]
KiCS2⟩ put (peopleFrom Tokyo) people [put city hugo Tokyo, zhenjiang]
[Person Hugo Tokyo, Person Zhenjiang Tokyo]
```

[7]This requirement would not be necessary in a reformulation of Theorem 4.2 for partial functions, but it ensures that the domain of the forward function (i.e., the range of the uncurried backward function) is the "type" of *all* sorted lists of people, as intended.

The first call demonstrates the *get* function querying all people from Tokyo in the database defined earlier. The second call demonstrates deleting *sebastian* by deleting him from the updated view passed to the *put* function. The result contains *hugo* who is not from *Tokyo* and *zhenjiang* who was included in the updated view. The third call demonstrates deleting *sebastian* and moving *hugo* to *Tokyo* by adding him to the list of people from *Tokyo*.

5.3.2 Reflecting deletions via modifications

Instead of deleting people from the database that are not present in the updated view queried by city we can also move them to a different city. This strategy can be implemented as follows:

1. first move all people from the queried city to the new city in the original source,
2. then call the update strategy defined in Section 5.3.1 to overwrite all moved people who are present in the view, effectively moving only those who are not present.

Example 5.3 (Reflecting deletions via modifications)**.** *The following definition implements the strategy described above.*

```
peopleFromTo :: City → City → BX [Person] [Person]
peopleFromTo from to source view =
    let moved = map move source
    in  peopleFrom from moved view
  where
    move p | get city p == from = put city p to
           | otherwise          = p
```

Again, we need to verify the conditions of Theorem 4.2 to ensure that the resulting BX is well-behaved:

1. Putting an original *source* with the same *view* twice is the same as updating it only once because the underlying transformation *peopleFrom* satisfies this property for all sources, i.e., also for the modified source passed by *peopleFromTo*.
2. The injectivity requirement is also implied by the corresponding property for the underlying transformation.
3. Regarding surjectivity, note that, even though people in the original *source* are moved initially, they can be moved back by including all moved people in the *view* argument. So obtaining an arbitrary source as result of *peopleFromTo* is achieved in the same way as for *peopleFrom*.

Here are example calls that demonstrate the difference between the *peopleFrom* strategy and the *peopleFromTo* strategy.

$KiCS2\rangle$ *get* (*peopleFromTo Tokyo Kiel*) *people*
[*Person Sebastian Tokyo*, *Person Zhenjiang Tokyo*]
$KiCS2\rangle$ *put* (*peopleFromTo Tokyo Kiel*) *people* [*zhenjiang*]
[*Person Hugo Braga*, *Person Sebastian Kiel*, *Person Zhenjiang Tokyo*]
$KiCS2\rangle$ *put* (*peopleFromTo Tokyo Kiel*) *people* [*put city hugo Tokyo*, *zhenjiang*]
[*Person Hugo Tokyo*, *Person Sebastian Kiel*, *Person Zhenjiang Tokyo*]

The *get* function of the BX *peopleFromTo Tokyo Kiel* is the same as the *get* function of *peopleFrom Tokyo*. The difference is only in the *put* function which moves *sebastian* from *Tokyo* to *Kiel* when he is deleted from the *view* instead of deleting him from the *source*. Inserted people, like *hugo*, are inserted into the original *source* (or modified if they already exist) just like before.

5.3.3 Missing support for ensuring well-behavedness

Examples 5.1 through 5.3 show how to write bidirectional transformations in putback style. The definitions inspired by database queries follow a verbal description of update strategies and are implemented modularly by reusing common parts. We have argued informally that our definitions satisfy the necessary conditions for the transformations to be well-behaved and observed their behavior using example calls.

Ensuring well-behavedness can be tricky, however. As an example of a bidirectional transformation where well-behavedness is possible but not easily ensured, we discuss joins of database tables without showing their implementation.

Consider that we define datatypes for books associating titles with owners which allow the following definitions.

someBooks = [*Book The_Art_of_Computer_Programming Zhenjiang*,
Book The_Elements_of_Style Sebastian,
Book The_Lord_of_the_Rings Hugo]
somePeople = [*hugo*, *zhenjiang*]

The result of joining the tables *someBooks* and *somePeople* is the following list of pairs containing titles and cities.

[(*The_Art_of_Computer_Programming*, *Zhenjiang*, *Tokyo*),
(*The_Lord_of_the_Rings*, *Hugo*, *Braga*)]

Sebastian's book is not included because he is not listed in the table *somePeople*. Say, we want to update this join to include another book owned by *Sebastian* and at the same time remove *Hugo*'s book.

someJoin = [(*The_Art_of_Computer_Programming*, *Zhenjiang*, *Tokyo*),
(*To_Mock_a_Mockingbird*, *Sebastian*, *Tokyo*)]

When implementing a putback function for joins, we need to consider at least the following questions.

1. Should books like *The_Element_of_Style* that are not part of the updated view be included in the book table of the updated source?
2. What about the explicitly removed book *The_Lord_of_the_Rings*?
3. Can we always add such new books to the book table?
4. Can we always add new owners to the person table?

We will answer these questions in turn considering well-behavedness of the resulting transformation.

1. Books like *The_Elements_of_Style* that are not part of the view but of the book table in the original source need to be included in order to satisfy the GETPUT law. If we do not include such books in the updated book table, we could not update the source (*someBooks*, *somePeople*) with it's join and get back the same source. On the other hand, we need to delete such books from the updated source, if the view is updated with other books of the same owner for reasons discussed below.
2. For deleted books like *Hugo*'s we have different options. We can delete the book from the updated book table or delete the owner from the updated person table. When deleting a book from the book table, we need to distinguish this case from the previous case where we have to keep a book that is not part of the view because the owner is not in the person table. Deleting *Hugo* from the people table is admissible in this case because he has no other books. If there were other books owned by *Hugo* in the original book table, however, we could not delete him from the updated people table without violating the PUTGET law: when computing the join for the updated source, all other books owned by *Hugo* would be missing.
3. We can (and, because of the PUTGET law, have to) add new books to the book table.
4. We can add new owners to the person table if they do not have other books listed in the book table but need to be careful about adding owners of other books. For example, if we add *Sebastian* to the updated person table because his book *To_Mock_a_Mockingbird* should be included in the view, his other book *The_Elements_of_Style* would also be included. Hence, we need to delete all other books from newly added owners or disallow adding books from new owners.

While it is, in principle, possible to implement every conceivable update strategy in putback style, it can be tricky to get right. Providing support for programmers to ensure well-behavedness of bidirectional transformations while maintaining the expressiveness which allows to define every well-behaved bidirectional transformation is subject of future research.

6 Related Work

The literature most related to our theory of BXs can be classified according to two main categories.

The first focuses on taking an existing language for defining the *get* function, and then trying to derive a suitable *put* function through *bidirectionalization* techniques. This kind of bidirectional approach has been mostly followed in the database community, where it is known as the classical *view-update problem.* A database administrator may define views that provide simplified or restructured information to users. Since the view typically contains less information than the source, a view update can be translated in general into multiple source updates, and the problem lies in how to choose a suitable update translation strategy.

The second approach is to design a *bidirectional programming language* in which a programmer writes both the *get* function and the *put* function in a single expression. Bidirectional programming languages for the kinds of BXs considered in this paper, where the get function defines a view and the backward function a view update strategy, are usually called *lenses.*

Deeper and broader revisions of related work on bidirectional transformations can be found in [Foster, 2009, Pacheco, 2012, Czarnecki et al., 2009, Hu et al., 2011].

6.1 View-update Problem

Work on database view-update translation has a long tradition in the database community, going back to the late 70s and 80s when it was extensively studied.

In seminal work, Bancilhon and Spyratos [1981] remark that, given a complement function such that the tupled *get*-complement function is injective, the translation of view updates under a constant complement is unique. Their formulation is essentially isomorphic to very well-behaved BXs [Foster et al., 2007].

To formalize the notion of a reasonable complement, Hegner [2004] extends the *closed views* theory of Bancilhon and Spyratos [1981] with a notion of order on source and view updates, such that *put* is monotonic: the reflection of an insertion in the view is an insertion in the source, and similarly for deletions. He then establishes that for a *get* written in a monotonic SPJ (select-project-join) relational algebra, there is a unique translation of insertion and deletion view updates independently of the choice of the complement.

Although *closed views* guarantee that view updates do not affect parts of the database that are not visible through the view, they are often too conservative and disallow many reasonable view update translations that do not keep any complement constant. A more liberal theory of *open views* that rejects fewer updates but permits several reasonable translations is proposed by Dayal and Bernstein [1982]. They formalize that a correct view update translation shall simply not introduce view side effects, and propose algorithms that perform one possible update translation for views written in a SPJ relational algebra. Their formulation is essentially isomorphic to well-behaved BXs [Foster et al., 2007].

Nevertheless, an open strategy is inherently ambiguous, as it may introduce side effects in the source database that users are not aware of by only looking at the view. Since different translations may be more appropriate for different scenarios, Keller [1986] proposes an interactive algorithm that, based on a SPJ view definition, runs a dialog with the view programmer to choose a particular view update policy that obeys a set of intuitive criteria.

Other authors like Larson and Sheth [1991] point out that, for some examples, the available information at view definition time is not sufficient for a view programmer to unambiguously select a suitable view update strategy, and that information about the issued view update is necessary. Using similar criteria as Keller [1986], they propose an interactive algorithm that collects semantic information both at view definition time (provided by the view programmer) and at view update time (provided by the view user), and permits choosing from a larger number update policies for a broader class of view definitions written using also relational difference, union and intersection.

A common feature of these relational BX approaches is that they are *operation-based*, i.e., they consider actual edit operations in contrast to our *state-based* BXs that consider whole source and view states, and the proposed update strategies process insertion, deletion and modification updates independently. For example, an update strategy that reacts to block operations like deleting a group in the source if all members of the group are deleted in the view is not definable in an approach that considers independent view updates. Moreover, the approaches with dialog only consider update translators that satisfy a given set of criteria deemed intuitive (normally by trying to reduce the number of source side effects), but not all well-behaved update translators. For example, the lens from Example 2.3 would be typically left out. Note also that *put* programming is not a commitment to an update translator at putback definition time: the view programmer can leave parameters (that do not affect the derived *get* function) to be controlled by users at view update time.

6.2 Bidirectional Programming Languages

In the last ten years, various bidirectional programing languages have become increasingly popular across a wide range of communities, including data synchronization, model transformations, graph transformations, relational databases and functional programming. We review only a few that are more related to our work.

The pioneering work of Foster et al. [2007] proposes one of the first bidirectional programming languages for defining views of tree-structured data. They recast many of the ideas for database view-updating into the design of a language of BXs named *lenses*, consisting of a *get* and a *put* function that satisfy well-behavedness laws analogous to the ones proposed by Dayal and Bernstein [1982] and Bancilhon and Spyratos [1981]. The novelty of their work is by putting emphasis on types and totality of lens transformations, and by proposing a series of combinators that allow reasoning about totality and well-behavedness of lenses in a compositional way. The kinds of BXs studied in our paper are precisely total (very) well-behaved lenses.

Bohannon et al. [2006] propose a language of lenses for relational data built using standard SPJ relational algebra combinators and composition. Their relational lenses provide one possible update policy based on a careful treatment of functional dependencies. They also develop a type system using record predicates and functional dependencies to express the exact conditions on the source and view schemas under which lenses are total and well-behaved.

Bohannon et al. [2008] design a language for the bidirectional transformation of string data, built using a set of regular operations and a type system of regular expressions. To overcome issues with order, their lens combinators adopt an update translation strategy based on keys that are introduced by the programmer in the form of annotations to lens expressions. Matching lenses [Barbosa et al., 2010] generalize the string lens language by lifting the update translation strategy from a key-based matching to support a set of different alignment heuristics that can be chosen by users.

Pacheco and Cunha [2010] propose a point-free functional language of total well-behaved lenses, using a simple positional update strategy, and later Pacheco et al. [2012] extend the matching lenses approach to infer and propagate insertion and deletion updates over arbitrary views defined in such point-free language.

Hidaka et al. [2010] propose the first linguistic approach for bidirectional graph transformations, by giving a bidirectional semantics to the UnCal graph algebra. Although their base semantics is compositional, they process deletions in the view by locating the correlated subgraph in the source, and for insertions in the view they have a dialog with the user at view update time to calculate the correlated inserted subgraph in the source.

All existing bidirectional programing approaches based on lenses focus on writing bidirectional programs that resemble writing the *get* function, and possibly take some additional parameters that provide limited control over the update strategy of the *put* function. Since these languages are state-based, the *put* function of a lens must align the updated view and the original source structures to identify the modifications and translate them to the source accordingly. Although for unordered data (relations, graphs) such alignment can be done rather straightforwardly, for ordered data (strings, trees) it is more problematic to find a reasonable alignment strategy, and thus to provide a reasonable view update translation strategy. Our results open the way to *put* programming languages, that in theory could give the programmer the possibility to express all well-behaved update translation strategies.

In his PhD thesis, Foster [2009] independently discusses a characterization of lenses in terms of *put* functions (considering only total *get* and *put* functions), in point-wise terms, similar to our main theorem. Interestingly, he arrives at a notion of *put semi-injectivity* that is slightly stronger than our injectivity of $put\ s$. He also uses the PutTwice law which we identify as idempotence of 'put'v. Nevertheless, he does so only to plead for a forward programming style and does not pursue a putback programming style. Moreover, he advocates that both styles are equivalent because writing a bidirectional program in a *get*-based bidirectional language is the same as writing a backward transformation. We disagree on the grounds that they are pragmatically distinct, as programmers of a BX in an existing *get*-based bidirectional language are limited by the language designer in their knowledge and control of the backward update strategy, and thus are supplied with "a" backward transformation and not "the" backward transformation. Our emphasis on putback programming highlights that programmers must be both aware of and responsible for the backward update strategy to specify a BX completely. We explore the putback style to demonstrate this difference and illustrate a possible way to derive *get* functions from *put* functions.

6.2.1 Symmetric Bidirectional Programming Languages

Bidirectional programming languages in the style of lenses are *asymmetric*, in the sense that *get* is surjective and the view type usually contains less information than the source type. In a *symmetric* bidirectional programming language, both the source and target types may contain information not present on the other side, giving rise to different laws and mathematical properties.

Meertens [1998] studies the construction of a language of *constraint maintainers* for preserving the consistency of artifacts in user interfaces. A maintainer has a forward function $get :: (S, T) \to T$ that propagates source updates, a backward function $put :: (T, S) \to S$ that propagates target updates and a consistency relation $R \subseteq (S, T)$ that the functions must preserve. Meertens [1998] also notes that composition of maintainers is not well-behaved in general, but interestingly proves that the composition of two well-behaved lenses with a common view type yields a well-behaved maintainer between their source types.

Hofmann et al. [2011] build a language of *symmetric lenses* over algebraic data structures. A maintainer consists of two transformations $get :: (S, C) \to (T, C)$ and $put :: (T, C) \to (S, C)$ and a complement $c :: C$ that preserves a history of the source and target information lost through previous update translations. Unlike maintainers, symmetric lenses support composition. They also show that any symmetric lens can be viewed as the composition of two lenses with a common source consisting of the domain of consistent triples of type (S, C, T).

7 Conclusions and Future Work

In this article, we characterize the class of (very) well-behaved bidirectional transformations solely based on their putback functions. In doing so, we rephrase existing laws for BXs based on simple mathematical concepts such as injectivity, surjectivity, and idempotence. We use our characterization to show that (very) well behaved BXs are uniquely determined by their backward functions and corresponding forward functions can be obtained automatically. In sharp contrast to bidirectional programming approaches based on *get*, writing *put* is sufficient to express all (very) well-behaved BXs.

Following this putback-based characterization of BXs, we show how to implement existing update strategies as putback functions. We use the built-in search facilities of the functional-logic programming language Curry, to obtain the *get* function corresponding to a user-defined *put* function that satisfies necessary conditions for well-behavedness. We informally argue that our definitions satisfy these well-behavedness properties and discuss, based on a more complicated example, that ensuring them can be difficult without further assistance.

The first immediate direction for future work is to investigate the design of *put* programming languages for particular data domains, that guide users by only allowing them to define well-behaved BXs but retain the full power of writing *put*. Notwithstanding, users are not necessarily obliged to fully control the update strategy: as for *get* programming, default parameters for *put* may be used when adequate. In fact, we believe that a style of injective *put s* functions with clear inverses can prove to be equally manageable and even more intuitive than the traditional style of surjective *get* functions with ambiguous inverses. Moreover, a fully expressive *put* programming language for a particular data domain (e.g., databases and relational algebra) could serve as a unified framework to express and compare BX approaches in that domain; and interfaces for existing BX approaches could be implemented on top of the core framework. Such a unified framework would constitute a foundational response to the recently growing unification effort of the different BX communities, stated in publications such as [Czarnecki et al., 2009, Hu et al., 2011, Terwilliger et al., 2012].

Some bidirectional programming approaches [Keller, 1986, Larson and Sheth, 1991, Hidaka et al., 2010] relax the requirement imposed by PutGet to admit view side effects in some particular cases, instead of disallowing view updates. An interesting direction for future work would be to investigate how to extend our theory to consider view side effects. Another challenging direction is to discover how to uniquely specify classes of symmetric BXs, either directly or by decomposing them into pairs of asymmetric BXs.

Acknowledgments

We thank Keisuke Nakano who inspired an example we use to show the ambiguitiy in specifying bidirectional transformations using only the forward function.

References

F. Bancilhon and N. Spyratos. Update semantics of relational views. *ACM Transactions on Database Systems*, 6(4):557–575, 1981.

D. M. J. Barbosa, J. Cretin, J. N. Foster, M. Greenberg, and B. C. Pierce. Matching lenses: alignment and view update. In *Proceedings of the 15th ACM SIGPLAN International Conference on Functional Programming (ICFP 2010)*, pages 193–204. ACM, 2010.

A. Bohannon, B. C. Pierce, and J. A. Vaughan. Relational lenses: a language for updatable views. In *Proceedings of the 25th ACM SIGMOD Symposium on Principles of Database Systems (PODS 2006)*, pages 338–347. ACM, 2006.

A. Bohannon, J. N. Foster, B. C. Pierce, A. Pilkiewicz, and A. Schmitt. Boomerang: resourceful lenses for string data. In *Proceedings of the 35th ACM SIGPLAN Symposium on Principles of Programming Languages (POPL 2008)*, pages 407–419. ACM, 2008.

P. Buneman, J. Cheney, and S. Vansummeren. On the expressiveness of implicit provenance in query and update languages. *ACM Transactions on Database Systems*, 33(4), 2008.

K. Czarnecki, J. N. Foster, Z. Hu, R. Lämmel, A. Schürr, and J. Terwilliger. Bidirectional transformations: A cross-discipline perspective. In *Proceedings of the 2nd International Conference on Model Transformation (ICMT 2009)*, volume 5563 of *LNCS*, pages 260–283. Springer-Verlag, 2009.

U. Dayal and P. Bernstein. On the correct translation of update operations on relational views. *ACM Transactions on Database Systems*, 7:381–416, 1982.

J. Foster. *Bidirectional Programming Languages.* PhD thesis, University of Pennsylvania, December 2009.

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

G. Gottlob, P. Paolini, and R. Zicari. Properties and update semantics of consistent views. *ACM Transactions on Database Systems*, 13(4):486–524, 1988.

M. Hanus (editor). Curry: An integrated functional logic language (vers. 0.8.3), 2012. Available at: `http://www.informatik.uni-kiel.de/~curry/papers/report.pdf`.

S. J. Hegner. An order-based theory of updates for closed database views. *Annals of Mathematics and Artificial Intelligence*, 40:63–125, 2004.

S. Hidaka, Z. Hu, K. Inaba, H. Kato, K. Matsuda, and K. Nakano. Bidirectionalizing graph transformations. In *Proceedings of the 15th ACM SIGPLAN International Conference on Functional Programming (ICFP 2010)*, pages 205–216. ACM, 2010.

M. Hofmann, B. C. Pierce, and D. Wagner. Symmetric lenses. In *Proceedings of the 38th annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL 2011)*, pages 371–384. ACM, 2011.

M. Hofmann, B. C. Pierce, and D. Wagner. Edit lenses. In *Proceedings of the 39th annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL 2012)*, pages 495–508. ACM, 2012.

Z. Hu, S.-C. Mu, and M. Takeichi. A programmable editor for developing structured documents based on bidirectional transformations. *Higher-Order and Symbolic Computation*, 21(1-2):89–118, 2008.

Z. Hu, A. Schürr, P. Stevens, and J. F. Terwilliger. Dagstuhl Seminar on Bidirectional Transformations (BX). *SIGMOD Record*, 40(1):35–39, 2011.

D. Jackson. *Software Abstractions: Logic, Language, and Analysis.* MIT Press, London, England, revised edition, 2012.

A. Keller. Choosing a view update translator by dialog at view definition time. In *Proceedings of the 12th International Conference on Very Large Databases (VLDB 86)*, pages 467–474. Morgan Kaufmann Publishers, 1986.

KiCS2 developers. The kiel curry system, 2012. Available at: `http://www-ps.informatik.uni-kiel.de/kics2/`.

R. Lämmel. Coupled Software Transformations (Extended Abstract). In *Proceedings of the 1st International Workshop on Software Evolution Transformations (SETS 2004)*, 2004.

J. A. Larson and A. P. Sheth. Updating relational views using knowledge at view definition and view update time. *Information Systems*, 16(2):145 – 168, 1991.

S. Marlow (editor). Haskell 2010 language report. available at: `http://www.haskell.org/onlinereport/haskell2010/`, 2010.

K. Matsuda, Z. Hu, K. Nakano, M. Hamana, and M. Takeichi. Bidirectionalization transformation based on automatic derivation of view complement functions. In *12th ACM SIGPLAN International Conference on Functional Programming (ICFP 2007)*, pages 47–58. ACM, Oct. 2007.

L. Meertens. Designing constraint maintainers for user interaction. Manuscript available at `http://www.kestrel.edu/home/people/meertens`, 1998.

H. Pacheco. *Bidirectional Data Transformation by Calculation.* PhD thesis, University of Minho, July 2012.

H. Pacheco and A. Cunha. Generic point-free lenses. In *Proceedings of the 10th International Conference on Mathematics of Program Construction (MPC 2010)*, volume 6120 of *LNCS*, pages 331–352. Springer-Verlag, 2010.

H. Pacheco, A. Cunha, and Z. Hu. Delta lenses over inductive types. In *Proceeding of the 1st International Workshop on Bidirectional Transformations (BX 2012)*, volume 49 of *Electronic Communications of the EASST*, 2012.

J. Terwilliger, A. Cleve, and C. Curino. How clean is your sandbox? In *Proceedings of the 5th International Conference on Model Transformation (ICMT 2012)*, volume 7307 of *LNCS*, pages 1–23. Springer-Verlag, 2012.

J. Voigtländer. Bidirectionalization for free! (pearl). In *Proceedings of the 36th annual ACM SIGPLAN-SIGACT Symposium on Principles of programming languages (POPL 2009)*, pages 165–176. ACM, 2009.

Y. Xiong, D. Liu, Z. Hu, H. Zhao, M. Takeichi, and H. Mei. Towards automatic model synchronization from model transformations. In *Proceedings of the 22nd IEEE/ACM International Conference on Automated Software Engineering (ASE 2007)*, pages 164–173. ACM, Nov. 2007.