Validity Checking of Putback Transformations in Bidirectional Programming

Zhenjiang Hu[1], Hugo Pacheco[2], Sebastian Fischer[3]

[1] National Institute of Informatics, Japan
[2] Cornell University, USA
[3] Christian-Albrechts University of Kiel, Germany

Abstract. A bidirectional transformation consists of pairs of transformations —a forward transformation *get* produces a target view from a source, while a putback transformation *put* puts back modifications on the view to the source— satisfying sensible roundtrip properties. Existing bidirectional approaches are *get*-based in that one writes (an artifact resembling) a forward transformation and a corresponding backward transformation can be automatically derived. However, the unavoidable ambiguity that stems from the underspecification of *put* often leads to unpredictable bidirectional behavior, making it hard to solve nontrivial practical synchronization problems with existing bidirectional transformation approaches. Theoretically, this ambiguity problem could be solved by writing *put* directly and deriving *get*, but differently from programming with *get* it is easy to write invalid *put* functions. An open challenge is how to check whether the definition of a putback transformation is valid, while guaranteeing that the corresponding unique *get* exists. In this paper, we propose, as far as we are aware, the first *safe* language for supporting putback-based bidirectional programming. The key to our approach is a simple but powerful language for describing primitive putback transformations. We show that validity of putback transformations in this language is decidable and can be automatically checked. A particularly elegant and strong aspect of our design is that we can simply reuse and apply standard results for treeless functions and tree transducers in the specification of our checking algorithms.

1 Introduction

Bidirectional transformations (BXs for short) [6,10,16], originated from the *view updating* mechanism in the database community [1,7,12], have been recently attracting a lot of attention from researchers in the communities of programming languages and software engineering since the pioneering work of Foster et al. on a combinatorial language for bidirectional tree transformations [10]. Bidirectional transformations provides a novel mechanism for synchronizing and maintaining the consistency of information between input and output, and have seen many interesting applications, including the synchronization of replicated data in different formats [10], presentation-oriented structured document development [17], interactive user interface design [21] or coupled software transformation [19].

A *bidirectional transformation* basically consists of a pair of transformations: the *forward* transformation $get\ s$ is used to produce a target view v from a source s, while the *putback* transformation $put\ s\ v$ is used to reflect modifications on the view v to the source s. These two transformations should be *well-behaved* in the sense that they satisfy the following round-tripping laws.

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

The GetPut property requires that 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.

Example 1. As a simple example, consider a forward function $getAs$ that selects from a source list all the elements that are tagged with A:

$$\begin{aligned} getAs\ [\,] &= [\,] \\ getAs\ (A\ a : ss) &= a : getAs\ ss \\ getAs\ (B\ b : ss) &= getAs\ ss \end{aligned}$$

and a corresponding putback function $putAs$ that uses a view list to update A elements in the original source list:

$$\begin{aligned} putAs\ [\,]\ [\,] &= [\,] \\ putAs\ [\,]\ (v : vs) &= A\ v : putAs\ [\,]\ vs \\ putAs\ (A\ a : ss)\ [\,] &= putAs\ ss\ [\,] \\ putAs\ (A\ a : ss)\ (v : vs) &= A\ v : putAs\ ss\ vs \\ putAs\ (B\ b : ss)\ vs &= B\ b : putAs\ ss\ vs \end{aligned}$$

where we use the view to replace A elements, impose no effect on B elements, and stop when both the source and view lists are empty. We also deal with the cases when the view and the source lists do not have sufficient elements. □

Bidirectional programming is to develop well-behaved BXs 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 get and putback transformations and can be realized using standard programming languages, the programmer needs to show that the two transformations satisfy the well-behavedness laws, and a modification to one of the transformations requires a redefinition of the other transformation as well as a new well-behavedness proof.

To ease and enable maintainable bidirectional programming, it is preferable to write just a single program that can denote both transformations, which has motivated two different methods. One is to allow users to write the forward transformation in a familiar (unidirectional) programming language, and derive a suitable putback transformation through *bidirectionalization* techniques

[13, 20, 26, 28]. The other is to instruct users to write a program in a particular *bidirectional programming language* [3, 4, 10, 14, 15, 22, 23], from which both transformations can be derived. The latter languages tend to invite users to write BXs as they would write *get* functions, but may provide eventually different *put* strategies via a fixed set of combinators.

In general, a *get* function may not be injective, so there may exist many possible *put* functions that can be combined with it to form a valid BX. Recall the definition of *putAs* from Example 1; we could define another reasonable putback function for *getAs* by changing the second and third equations to:

$$\begin{array}{ll} putAs\ [\,] & (v : vs) = A\ v : B\ c : putAs\ [\,]\ vs \\ putAs\ (A\ a : ss)\ [\,] & = putAs\ (B\ a : ss)\ [\,] \end{array}$$

such that an additional B-tagged constant value c is added after each view value v and excessive A values are converted to B values.

This unavoidable ambiguity of put is what makes bidirectional programming challenging and unpredictable in practice. In fact, there is neither a clear consensus on the best requirements even for well-studied domains [5], nor a general way to specify which *put* should be selected. The effectiveness of existing bidirectional programming methods comes from limiting the programmers' knowledge and control of the putback transformation, to keep bidirectional programming manageable. Unfortunately, this makes it hard (or impossible) for programmers to mold the bidirectional behavior, and severely hinders the applicability of existing BX tools in solving practical nontrivial synchronization problems.

One interesting fact is that while *get* usually loses information when mapping from a source to a view, *put* must preserve information when putting back from the view to the source, according to the PutGet property. So, a natural question is: what if we replace the traditional get-based bidirectional programming style by a putback-based bidirectional programming style? This is, writing *put* and deriving *get* (or, in other words, specifying the intended putback transformation that best suits particular purposes, and deriving the forward transformation.)

Theoretically, it has been shown in [8, 9] that, *for a putback transformation put, if there exists a forward transformation get then such forward transformation is unique.* Practically, however, there is little work on put-based bidirectional programming. This is not without reason: as argued in [9], it is far from being straightforward to construct a framework that can directly support putback-based bidirectional programming. One of the challenges is how to check whether the definition of a *put* is in such a valid form that guarantees that the corresponding unique *get* exists. In contrast to programming *get*, it is easy to write invalid *put* functions. For instance, if we change the first equation for *putAs* to:

$$putAs\ (A\ a : ss)\ (v : vs) = A\ a : A\ v : putAs\ ss\ vs$$

then we will end up with an invalid *put* for which there is no *get* that forms a well-behaved BX. This raises the question of how to statically check the validity of *put*.

In this paper, we propose (as far as we are aware) the first *safe* language for supporting putback-based bidirectional programming. We propose to adopt a hybrid compositional approach, keeping the design of well-behaved primitive putback transformations separated from the design of compositional methods for gluing smaller BXs. In this approach, a set of primitive BXs is prepared, and a new BX is defined by assembling the primitive transformations with a fixed set of general combinators. This approach has two main advantages. First, a comprehensive set of useful generic combinators [3, 4, 10, 14, 15, 22, 23] already exists and can be used without further development. Second, since these combinators are rather limited in specifying sophisticated bidirectional behavior, it is practically useful to be able to write primitive BXs, that are often easily determined, designed and implemented for particular domain-specific applications.

The key to our approach is a suitable language for describing primitive putback transformations. We choose a general first-order functional language and require putback functions definable in the language to be affine (each view variable is used at most once) and in the treeless form (no intermediate data structures are used in a definition). In fact, this class of functions has been considered elsewhere in the context of deforestation [27], where treeless functions are used to describe basic computation components, and has a close relationship with theories of tree transducers [18]. As will be demonstrate later, this language is sufficiently powerful to specify various putback functions over algebraic data structures and, more importantly, validity of putback transformations in the language can be automatically checked.

The rest of this paper is organized as follows. Section 2 begins by briefly reviewing the basic *put*-based bidirectional programming concepts and properties that play an important role in our language design. Section 3 then introduces our Pdl language for specifying primitive putback functions, and Section 4 propose our checking algorithms for validating putback functions (and deriving forward transformations as a side effect). Section 5 discusses related work and Section 6 provides our conclusions together with possible directions for future work.

2 Putback-based Bidirectional Programming

Let us briefly review the basic concepts and results from [8, 9] that clarify the essence of putback-based programming and play an important role in our validity checking. Calculational proofs of all the results can be found in [8].

First of all, we define validity of a putback transformation *put* as follows.

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

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

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

To facilitate the validity checking of *put* without mentioning *get*, we introduce two new properties on *put* whose combination is equivalent to GetPut and PutGet.

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

$$\forall\ s, s', v, v'.\ put\ s\ v\ =\ put\ s'\ v'\ \Rightarrow\ v\ =\ v' \qquad \textsc{PutDetermination}$$

  Note that the view determination implies that *put* s is injective (with $s = s'$).
- The second, that we call *source stability*, denotes a slightly stronger notion of surjectivity for every source:

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

Actually, these two laws together provide an equivalent characterization of the validity of *put*. The following theorem will be the basis for our later presented algorithms for checking of validity of *put* and deriving *get*.

Theorem 1 (Validity). *A put function is valid if and only if it satisfies the* PutDetermination *and* PutStability *properties.*

For the context of this paper, we are assuming that all functions are *total* —in the pure mathematical sense— between an input type and an output type.

3 Defining Putback Functions

In this section, we design a language for describing putback functions, such that the validity of putback functions written in our language can be automatically checked and the corresponding *get* functions can be automatically derived.

As explained in the introduction, we adopt a hybrid compositional approach, keeping separate the design of well-behaved primitive putback transformations and the design of compositional methods for gluing primitive bidirectional transformations. We will focus on the former —designing the language for specifying various primitive putback functions (with rich update strategies) over algebraic data structures— while existing generic combinators [3, 4, 10, 14, 15, 22, 23] can be reused to glue them together into larger transformations.

3.1 Pdl: A Putback Function Definition Language

We introduce Pdl, a treeless language for defining primitive *put* functions. By treeless, we mean that no composition can be used in the definition of a *put* function. It is a first-order functional programming language similar to both Wadler's language for defining basic functions for fusion transformation [27] and the language for defining basic *get* functions of Matsuda et al. [20], with a particularity that it also supports pattern expressions in source function calls.

The syntax of Pdl is given in Figure 1. A program in our language consists of a set of putback function definitions, and each definition consists of a sequence

Rule Definition

r	$::= f\ p_s\ p_v = e$	putback

Pattern

p	$::= \mathsf{C}\ p_1\ \ldots\ p_n$	constructor pattern
	$\mid\ x\ @\ p$	look-ahead variable
	$\mid\ x$	variable

Expression

e	$::= \mathsf{C}\ e_1\ \ldots\ e_n$	constructor application
	$\mid\ x$	variable
	$\mid\ \underline{f\ x_s\ x_v}$	function call (no nested calls)

where $\mathsf{C} \in \mathcal{C}$ is of arity n, $f \in \mathcal{P}$ and $x \in \mathcal{X}$.

Operational Semantics (Call-by-Value):

$$(\text{Con})\ \frac{e_1 \Downarrow r_1 \quad \cdots \quad e_n \Downarrow r_n}{\mathsf{C}\ e_1\ \ldots\ e_n \Downarrow \mathsf{C}\ r_1\ \ldots\ r_n} \qquad (\text{Fun})\ \frac{f\ p_s\ p_v \mathrel{\hat{=}} e \in \mathcal{R} \quad \exists\theta, f\ p_s\theta\ p_v\theta = f\ r_s\ r_v \quad e\theta \Downarrow u}{f\ r_s\ r_v \Downarrow u}$$

where "$e\theta$" denotes the expression that is obtained by replacing any variable x in e with the value $\theta(x)$, and $v_1, \ldots, v_n$ denote values; values are expressions that consist only of constructor symbols in $\mathcal{C}$.

Fig. 1. Putback Definition Language ($\mathcal{P}$ denotes putback function symbols, $\mathcal{C}$ denotes constructor symbols, $\mathcal{X}$ denotes variables)

of putback rules. A *putback rule*, as the name suggests, is used to put view information back into the source, and has the form:

$$f\ p_s\ p_v \mathrel{\hat{=}} e$$

It describes how f adapts the source p_s to e, when the view is of the form p_v. We make the following additional considerations:

- For the patterns p_s and p_v, in addition to traditional variable and constructor patterns, we introduce look-ahead variable patterns mainly for the purpose of abstracting constant patterns using variables. For example, we can write the constant pattern $[\,]$ as $xs@[\,]$, which allows us to syntactically distinguish whether an empty string appearing in the right-hand side is newly created or passed from the input.
- We require the body expression e to be in an extended *structured treeless* form [27], i.e., a function call should have shape $f\ x_s\ x_v$, where x_s is a variable in the source pattern p_s and x_v is a variable in the view pattern p_v. This means that a recursive call of a putback function updates components of the source with the components of the view, and it may appear inside a constructor application, but never inside another function call.

– We assume that each rule is *affine*, i.e., every view variable in the left-hand side of a rule occurs at most once in the corresponding right-hand side.

Definition 2 (Putback Transformation in PDL). *A putback transformation is a total function defined by a set of putback rules.*

We can see that *putAs* in Example 1 is almost a putback transformation in PDL, except that some arguments of recursive calls are an empty list instead of a variable. This can be easily resolved by using a look-ahead variable.

Example 2. The following *putAs* is defined in PDL.

$$
\begin{array}{lll}
putAs\ [\,] & [\,] & = [\,] \\
putAs\ (ss@[\,]) & (v : vs) & = A\ v : putAs\ ss\ vs \\
putAs\ (A\ a : ss) & (vs@[\,]) & = putAs\ ss\ vs \\
putAs\ (A\ a : ss) & (v : vs) & = A\ v : putAs\ ss\ vs \\
putAs\ (B\ b : ss) & vs & = B\ b : putAs\ ss\ vs
\end{array}
$$

□

Let us demonstrate with more examples that PDL is powerful enough to describe various putback transformations (functions).

Example 3 (Fully Updating). The simplest putback function uses the view to fully update the original source, or in other words, to fully embed the view to the source. This can be defined in PDL as follows.

$$updAll\ s\ v = v$$

□

Example 4 (Updating Component). We may use the view to update the first or second component of a source pair, or say, to embed the view to first or second component of a source pair:

$$
\begin{array}{l}
updFst\ (Pair\ x\ y)\ v = Pair\ v\ y \\
updSnd\ (Pair\ x\ y)\ v = Pair\ x\ v
\end{array}
$$

□

Example 5 (Updating Data Structure). We may use the view to update the last element of a non-empty source list[4]:

$$
\begin{array}{ll}
updLast\ [s]\ v & = [v] \\
updLast\ (s : ss)\ v & = s : updLast\ ss\ v
\end{array}
$$

For this particular example, we consider the type of non-empty lists because otherwise *updLast* would not be total, since there is no rule for putting a view element back into an empty source list. □

[4] A non-empty list type can be defined as $A^+ = Wrap\ A \mid NeCons\ A\ A^+$, but for simplicity we abuse the notation and write our example using regular lists.

Two remarks are worth making. First, all putback rules in PDL should meet the syntactic constraints as discussed before; those that do not satisfy these constraints are not considered to be a putback rule. For instance, the following rule is not a putback rule, because s appears twice in thr right hand side.

$$putSyntacBad\ s\ v = putSyntacBad\ s\ s$$

Second, a putback transformation defined in PDL may not be valid. For instance, the putback transformation defined by

$$putInvalid\ s\ v = s$$

which completely ignores the view v. The function *putInvalid* is invalid in the sense there is no actual *get* function that can be paired with it to form a valid BX. In this paper we will show that the validity of any putback transformation in PDL can be automatically checked.

3.2 Properties of Putback Transformations in PDL

Putback transformation in PDL enjoy two features, which will play an important role in our later validity checking.

First, some equational properties on PDL putback transformations can be automatically proved inductively. This is because putback transformations are structured in a way such that any recursive call is applied to sub-components of the input. In fact, such structural and total recursive functions fall in the category where inductive validity is decidable [11]. More specifically, the following lemma holds.

Lemma 2 (Decidability of Inductivity of Putback Transformation). *Let put be a putback transformation. An equational property in the following form*

$$put\ e_1\ e_2 = p$$

can be automatically proved by induction, where e_1 and e_2 are two expressions and p is a pattern.

Note that the equational property that can be dealt with by the above lemma requires its right hand side to be a simple pattern, this is, a constructor term without (recursive) function calls.

Second, PDL putback transformations are closed under composition. This follows from the known fact that compositions of functions in treeless form are again functions in treeless form [27] and these function can be automatically derived. Usually, treeless functions are defined in a more general form:

$$f\ p_1\ \ldots\ p_n = e$$

where a function can have an arbitrary number of inputs. So, a putback transformation in PDL is a special case which has two predefined (source and view) inputs. The following lemma can be easily obtained, and will be used later.

Lemma 3 (Putback Transformation Fusion). *Let put be a putback transformation and f be a one-input treeless function. Then a new putback transformation put′ can be automatically derived from the following definition.*

$$put'\ s\ v = put\ s\ (f\ v)$$

4 Validity Checking

Given a *put* function in PDL, we will now give an algorithm to check whether it is valid. According to Theorem 1, we need to check two conditions: view determination of *put* and source stability of *put*. Additionally, we need to check that *put* is a total function, what in PDL can be easily done by checking the exhaustiveness of the patterns for all the rules. To simplify our presentation, we will consider putback transformations that are single recursive functions.

4.1 View Determination Checking

First, let us see how to check injectivity of *put s*. Notice that $\mathcal{FV}(p_v) \subseteq \mathcal{FV}(e)$ is a necessary condition, where $\mathcal{FV}(e)$ denotes a set of free variables in expression e. This is because if there is a view pattern variable v that does not appear in e, then we can construct two different views, say v_1 and v_2, such that they match p_v but differ in the part of the code matching v and satisfy $put\ s\ v_1 = put\ s\ v_2$ for any s matching p_s. For instance, the following view embedding rule

$$putNoInj\ (A\ s)\ v = A\ s$$

will make *putNoInj* non-injective because, for any two views v_1 and v_2, we have $putNoInj\ (A\ s)\ v_1 = putNoInj\ (A\ s)\ v_2 = A\ s$

In fact, the above necessary condition is also a sufficient condition. Following [20], we can prove the following stronger lemma.

Lemma 4 (Injectivity Checking). *Let put be a putback transformation in* PDL*. Then put s is injective, for any s, if and only if* $\mathcal{FV}(p_v) \subseteq \mathcal{FV}(e)$ *holds for any putback rule put* $p_s\ p_v \mathrel{\hat{=}} e$*.*

However, proving that *put s* is injective, for any s, is not sufficient to guarantee that *put* satisfies view determination. For example, consider a putback function that sums two natural numbers:

$$bad\ Z\ v = v$$
$$bad\ (S\ s)\ v = S\ (bad\ s\ v)$$

Even though *bad s* is injective, we can easily find a counter-example showing that *bad* is not view deterministic:

$$bad\ Z\ (S\ Z) = S\ Z$$
$$bad\ (S\ Z)\ Z = S\ Z$$

where different views $S\ Z$ and Z lead to the same source $S\ Z$. In fact, there is no (functional) left inverse *get* such that $get\ (bad\ s\ v) = v$.

This requires finding a more general method to check the view determination property. Let us first take a closer look at the view determination property:

$$\forall\ s, s', v, v'.\ put\ s\ v\ =\ put\ s'\ v'\ \Rightarrow\ v\ =\ v'$$

Since *put* must map different views to different sources, this property is equivalent to stating that the inverse mapping from the result of putback to the input view is be functional (or single-valued), i.e., a relation that returns at most one view for each source. This hints us to divide the checking problem into two steps for a given putback transformation *put*: (1) deriving such an inverse mapping, say R_{put}, and (2) checking that R_{put} is single-valued.

Deriving Inverse Mapping from *put*

Consider a putback transformation *put* defined by a set of putback rules, ignoring rules in the form:

$$put\ p_s\ p_v = put\ p'_s\ p_v$$

for which view determination trivially holds. Now the inverse mapping R from the result of *put* to its input view can be defined by inverting the remaining putback rules $put\ p_s\ p_v = e$, i.e.,

$$R_{put}\ e = p_v\ \ \text{iff}\ \ put\ p_s\ p_v = e$$

Example 6. As a concrete example, recall the *putAs* function from Example 2. We can automatically derive the following "relation" R_{PutAs}.

$$\begin{array}{ll} R_{putAs}\ [\,] & = [\,] \\ R_{putAs}\ (A\ v : putAs\ ss\ vs) & = v : vs \\ R_{putAs}\ (putAs\ ss\ vs) & = v : vs \\ R_{putAs}\ (B\ b : putAs\ ss\ vs) & = vs \end{array}$$

It covers all the putback rules except for the rule $putAs\ (A\ a : ss)\ (vs\ as\ [\,]) = putAs\ ss\ vs$. □

The above derived R_{put} would be a bit unusual, in that *put* could appear on the left-hand side. In fact, each equation can be normalized into the form:

$$R_{put}\ p = e$$

where p is a pattern and e is an expression as in PDL. The idea is to eliminate recursive calls $put\ x_s\ x_v$ by introducing a new pattern variable $x'_s = put\ x_s\ x_v$ (and thus $R_{put}\ x'_s = x_v$), and replacing $put\ x_s\ x_v$ by x'_s in the left-hand side and x_v by $R_{put}\ x'_s$ in the right-hand side of the equation.

Example 7. After normalization, we can transform the R_{putAs} from Example 6 into the following.

$$\begin{array}{l} R_{putAs}\ [\,] \quad\quad\ = [\,] \\ R_{putAs}\ (A\ v : ss') = v : R_{putAs}\ ss' \\ R_{putAs}\ (A\ v : ss') = v : R_{putAs}\ ss' \\ R_{putAs}\ (B\ b : ss') = R_{putAs}\ ss' \end{array}$$

After removing duplicated rules, we get the following final R_{putAs}.

$$\begin{array}{l} R_{putAs}\ [\,] \quad\quad\ = [\,] \\ R_{putAs}\ (A\ v : ss') = v : R_{putAs}\ ss' \\ R_{putAs}\ (B\ b : ss') = R_{putAs}\ ss' \end{array}$$

□

Checking Single-valuedness of the Mapping

First, it is easy to show that the derived R can always be translated into a (finite state) top-down tree transducer [25] where each rule has the form $R_{put}\ p = e$ and all free variables in e are those in p and appear exactly once. This conclusion relies on the assumption that view variables are used exactly once in the right side of putback rules, as implied by the affinity syntactic constraint and the necessary injectivity of *put s*.

Note that, in general, R_{put} may not be a function, by containing overlapping patterns that may return different view values for the same source. For instance, our inversion algorithm will produce the following non-deterministic relation for the putback definition of *bad*:

$$\begin{array}{l} R_{bad}\ n \quad\ = n \\ R_{bad}\ (S\ n) = R_{bad}\ n \end{array}$$

where $R_{bad}\ (S\ 0) = S\ 0$ from the first equation, and $R_{bad}\ (S\ n) = 0$ from the second equation (followed by the first equation).

If the derived R_{put} returns at most one view value for every source value, then it corresponds directly (modulo removal of possibly overlapping but similar patterns) to a *get* in a treeless function similar to PDL. This is equivalent to stating that the corresponding tree transducers is single-valued, a problem that is fortunately known to be decidable in polynomial time [25].

Lemma 5 (Single-valuedness of *get*). *It is decidable if the relation R_{put} derived from a putback function put in* PDL *is a functional.*

4.2 Source Stability Checking

With the R_{put} relation derived in the previous section in hand, checking source stability of a putback function *put* amounts to proving that, for any source s, the GETPUT property holds:

$$put\ s\ (R_{put}\ s)\ =\ s$$

Algorithm: Validity Checking of Putback Transformation

Input: A program $P = (\mathcal{R}, \mathcal{F}, \mathcal{C}, \mathcal{X})$ for putback definitions in PDL.

Procedure:

check the syntactic constraints for each rule r in $\mathcal{R}$;
  {* check totality *}
check pattern exhaustiveness for each putback definition in $\mathcal{R}$;

for each $f\ p_s\ p_v \mathrel{\hat{=}} e \in \mathcal{R}$ **do**
  begin
    {* check view determination *}
    check injectivity: $\mathcal{FV}(p_v) \subseteq \mathcal{FV}(e)$;
    derive and **normalize** R_f;
    check view determination: R_f is single-valued;
    {* check source stability *}
    define $pr\ s\ v \mathrel{\hat{=}} f\ s\ (R_f\ v)$;
    fusion $pr\ s\ v \mathrel{\hat{=}} f\ s\ (R_f\ v)$ to be a new putback transformation;
    check property $pr\ s\ s = s$ inductively;
  end;
return *True* if all the checks are passed, and *False* otherwise.

Fig. 2. Validity Checking Algorithm

Note that GetPut implies in particular PutStability. Above that, at this point we only know that R_{put} is functional, but not that it constitutes a valid *get* function, i.e., that it is totally defined for all sources. This single proof also gives us that result.

The proof can be conducted as follows. First, we introduce a new (partial) function pr defined as:

$$pr\ x\ y = put\ x\ (R_{put}\ y)$$

Since R_{put} is in the treeless form, it follows from Lemma 3 that pr is a putback transformation in PDL. Now by Lemma 2, we know that $pr\ s\ s = s$ is inductively provable. That is, $put\ s\ (R_{put}\ s) = s$ is inductively provable, which is what we want.

Lemma 6 (Source Stability Checking). *Let put be a putback function in* PDL *and R_{put} be a treeless function. Then it is decidable if put is source stable.*

4.3 Checking Algorithm

Figure 2 summarizes our checking algorithm. The input is a program defining a set of putback definitions $\mathcal{F}$ using a set of rules $\mathcal{R}$ with a set of data constructors $\mathcal{C}$ and a set of variables $\mathcal{C}$. The checking algorithm will return *True* if all the putback definitions are valid, and return *False* otherwise.

Theorem 2 (Soundness and Completeness). *The putback checking algorithm is sound, in that if all putback functions pass the check then they are valid, and complete, in that there are no putback functions defined in* PDL *that are valid but do not pass the check.*

Proof. It directly follows from Lemmas 4 and 6. □

5 Related Work

The pioneering work of Foster et al. [10] 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 [1, 7] into the design of a language of *lenses*, consisting of a *get* and a *put* function that satisfy well-behavedness laws. 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 well-behaved lenses.

After that, many bidirectional languages have been proposed. Bohannon et al. [4] propose a language of lenses for relational data built using standard SPJ relational algebra combinators and composition. Bohannon et al. [3] design a language for the BX of string data, built using a set of regular operations and a type system of regular expressions. Matching lenses [2] generalize the string lens language by lifting the update strategy from a key-based matching to support a set of different alignment heuristics that can be chosen by users. Pacheco and Cunha [22] propose a point-free functional language of total well-behaved lenses, using a simple positional update strategy, and later [23] they 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. [13] propose the first linguistic approach for bidirectional graph transformations, by giving a bidirectional semantics to the UnCal graph algebra. All the above existing bidirectional programing approaches based on lenses focus on writing bidirectional programs that resemble the *get* function, and possibly take some additional parameters that provide limited control over the *put* function.

Since these *get*-based languages are often state-based, they must align the updated view and the original source structures to identify the modifications on the view 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 towards *put* programming languages, that in theory could give the programmer the possibility to express all well-behaved update translation strategies (for a given class of *get* functions).

In his PhD thesis, Foster [9] discusses a characterization of lenses in terms of *put* functions. However, he does so only to plead for a forward programming style and does not pursue a putback programming style. In [8], we independently

review classes of lenses solely in terms of their putback functions, rephrasing existing laws in terms of simple mathematical concepts. 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. Furthermore, in [24], a monadic combinator library for supporting putback style bidirectional programming is proposed. None of them considers mechanisms to ensure the validity of user-defined *put* functions and especially totality of the transformations. In the current paper, we explore the putback style to demonstrate that it can be advantageous and viable in practice, and illustrate a possible way to specify valid (total) *put* functions and correctly derive (total) *get* functions.

6 Conclusions and Future Work

In this paper, we have proposed a novel linguistic framework for supporting a putback-based approach to bidirectional programming: a new language has been designed for specifying primitive putback transformations, an automatic algorithm has been given to statically check whether a *put* is valid, and a derivation algorithm has been provided to construct an efficient *get* from a valid *put*. Our new framework retains the advantages of writing a single program to specify a BX but, in sharp contrast to get-based bidirectional programming, allows programmers to describe their intended *put* update strategies in a direct, predictable and, most importantly, unambiguous way.

The natural direction for future work is to consider extensions to PDL to support a larger class of BXs, while retaining the soundness and completeness of the validity checking algorithms. It remains open to prove results about the completeness of (pratical) putback-based programming, i.e., identifying classes of *get* functions for which concrete putback definition languages can specify all valid *put* functions.

References

1. Bancilhon, F., Spyratos, N.: Update semantics of relational views. ACM Transactions on Database Systems 6(4), 557–575 (1981)
2. Barbosa, D.M.J., Cretin, J., Foster, J.N., Greenberg, M., Pierce, B.C.: Matching lenses: alignment and view update. In: ICFP 2010. pp. 193–204. ACM (2010)
3. Bohannon, A., Foster, J.N., Pierce, B.C., Pilkiewicz, A., Schmitt, A.: Boomerang: resourceful lenses for string data. In: POPL 2008. pp. 407–419. ACM (2008)
4. Bohannon, A., Pierce, B.C., Vaughan, J.A.: Relational lenses: a language for updatable views. In: PODS 2006. pp. 338–347. ACM (2006)
5. Buneman, P., Cheney, J., Vansummeren, S.: On the expressiveness of implicit provenance in query and update languages. ACM Transactions on Database Systems 33(4) (2008)
6. Czarnecki, K., Foster, J.N., Hu, Z., Lämmel, R., Schürr, A., Terwilliger, J.: Bidirectional transformations: A cross-discipline perspective. In: ICMT 2009. LNCS, vol. 5563, pp. 260–283. Springer-Verlag (2009)

7. Dayal, U., Bernstein, P.: On the correct translation of update operations on relational views. ACM Transactions on Database Systems 7, 381–416 (1982)
8. Fischer, S., Hu, Z., Pacheco, H.: "Putback" is the Essence of Bidirectional Programming (2012), GRACE Technical Report 2012-08, National Institute of Informatics, 36pp
9. Foster, J.: Bidirectional Programming Languages. Ph.D. thesis, University of Pennsylvania (December 2009)
10. Foster, J.N., Greenwald, M.B., Moore, J.T., Pierce, B.C., Schmitt, A.: Combinators for bidirectional tree transformations: A linguistic approach to the view-update problem. ACM Transactions on Programming Languages and Systems 29(3), 17 (2007)
11. Giesl, J., Kapur, D.: Decidable classes of inductive theorems. In: Proc. IJCAR '01, LNAI 2083. pp. 469–484 (2001)
12. Gottlob, G., Paolini, P., Zicari, R.: Properties and update semantics of consistent views. ACM Transactions on Database Systems 13(4), 486–524 (1988)
13. Hidaka, S., Hu, Z., Inaba, K., Kato, H., Matsuda, K., Nakano, K.: Bidirectionalizing graph transformations. In: ICFP 2010. pp. 205–216. ACM (2010)
14. Hofmann, M., Pierce, B.C., Wagner, D.: Symmetric lenses. In: POPL 2011. pp. 371–384. ACM (2011)
15. Hofmann, M., Pierce, B.C., Wagner, D.: Edit lenses. In: POPL 2012. pp. 495–508. ACM (2012)
16. Hu, Z., Schürr, A., Stevens, P., Terwilliger, J.F.: Dagstuhl Seminar on Bidirectional Transformations (BX). SIGMOD Record 40(1), 35–39 (2011)
17. Hu, Z., Mu, S.C., Takeichi, M.: A programmable editor for developing structured documents based on bidirectional transformations. Higher-Order and Symbolic Computation 21(1-2), 89–118 (2008)
18. Kühnemann, A.: Comparison of deforestation techniques for functional programs and for tree transducers. In: FLOPS 99. pp. 114–130. Springer-Verlag (1999)
19. Lämmel, R.: Coupled Software Transformations (Extended Abstract). In: SETS 2004 (2004)
20. Matsuda, K., Hu, Z., Nakano, K., Hamana, M., Takeichi, M.: Bidirectionalization transformation based on automatic derivation of view complement functions. In: ICFP 2007. pp. 47–58. ACM (2007)
21. Meertens, L.: Designing constraint maintainers for user interaction (1998), manuscript available at `http://www. kestrel. edu/home/people/meertens`
22. Pacheco, H., Cunha, A.: Generic point-free lenses. In: MPC 2010. LNCS, vol. 6120, pp. 331–352. Springer-Verlag (2010)
23. Pacheco, H., Cunha, A., Hu, Z.: Delta lenses over inductive types. In: BX 2012. Electronic Communications of the EASST, vol. 49 (2012)
24. Pacheco, H., Hu, Z., Fischer, S.: Monadic combinators for "putback" style bidirectional programming. In: PEPM '14. pp. 39–50. ACM (2014)
25. Seidl, H.: Single-valuedness of tree transducers is decidable in polynomial time. Theor. Comput. Sci. 106(1), 135–181 (Jan 1992)
26. Voigtländer, J.: Bidirectionalization for free! (pearl). In: POPL 2009. pp. 165–176. ACM (2009)
27. Wadler, P.: Deforestation: Transforming programs to eliminate trees. In: ESOP 88. pp. 344–358. Springer-Verlag (1988)
28. Xiong, Y., Liu, D., Hu, Z., Zhao, H., Takeichi, M., Mei, H.: Towards automatic model synchronization from model transformations. In: ASE 2007. pp. 164–173. ACM (2007)