Characterizing Compatible View Updates in Syntactic Bidirectionalization*

Naoki Nishida[1] and Germán Vidal[2]

[1] Graduate School of Informatics, Nagoya University
Furo-cho, Chikusa-ku, 464-8603 Nagoya, Japan,
nishida@i.nagoya-u.ac.jp
[2] MiST, DSIC, Universitat Politècnica de València
Camino de Vera, s/n, 46022 Valencia, Spain
gvidal@dsic.upv.es

Abstract. Given a function that takes a *source* data and returns a *view*, bidirectionalization aims at producing automatically a new function that takes a modified view and returns the corresponding, modified source. In this paper, we consider simple first-order functional programs specified by (conditional) term rewrite systems. Then, we present a bidirectionalization technique based on the injectivization and inversion transformations from [24]. We also prove a number of relevant properties which ensure that changes in both the source and the view are correctly propagated and that no undesirable side-effects are introduced. Furthermore, we introduce the use of narrowing—an extension of rewriting that replaces matching with unification—to precisely characterize *compatible* (also called *in-place*) view updates so that the resulting bidirectional transformations are well defined. Finally, we discuss some directions for dealing with view updates that are not compatible.

Keywords: Bidirectional transformations · functional programming languages · term rewriting · narrowing

1 Introduction

The framework of bidirectional transformations (bx) considers two representations of some data and the functions that convert one representation into the other and vice versa (see, e.g., [18] for an overview). Typically, we have a function called "get" that takes a *source* and returns a *view*. In turn, the function "put" takes a possibly updated view and returns the corresponding source. In

* This work has been partially supported by the EU (FEDER) and the *Spanish Ministerio de Ciencia, Innovación y Universidades*/AEI under grant TIN2016-76843-C4-1-R, by the *Generalitat Valenciana* under grants PROMETEO-II/2015/013 (SmartLogic) and Prometeo/2019/098 (DeepTrust), by the COST Action IC1405 on Reversible Computation - extending horizons of computing, and by JSPS KAKENHI Grant Number JP17H01722.

this context, *bidirectionalization* [22] aims at automatically producing one of the functions, typically producing a function put from the corresponding function get (but the opposite approach is also possible, see, e.g., [10]). For this purpose, a so called *complement* function is often introduced so that the get function becomes injective (see, e.g., [12]).

Somewhat independently, reversible computation considers execution principles that can proceed both forward (i.e., normal computation) and backward. Moreover, *reversibilization* aims at transforming an irreversible computation principle into a reversible one. In particular, Landauer's seminal work [20] states how any computation principle can be made reversible by adding the history of the computation to each state. Although it may seem impractical at first, there are several useful reversibilization techniques that are roughly based on this idea (e.g., [8, 22, 24, 26]).

In this work, we consider a simple first-order functional programming language for our developments.[3] Consider, e.g., the following simple function:[4]

$$\begin{array}{l}\mathsf{fn}\ [\,] = [\,]\\ \mathsf{fn}\ ((\mathsf{Name}\ n\ l){:}xs) = n{:}ys\ \ \mathsf{where}\ ys = \mathsf{fn}\ xs\\ \mathsf{fn}\ ((\mathsf{City}\ c){:}xs) = ys\ \ \mathsf{where}\ ys = \mathsf{fn}\ xs\end{array}$$

The function fn takes a list of names of the form (Name *first_name last_name*) and cities of the form (City *name*), and returns a list of *first_name*s. E.g.,

$$\mathsf{fn}\ [\mathsf{Name\ John\ Smith}, \mathsf{City\ London}, \mathsf{Name\ Ada\ Lovelace}]$$

evaluates to [John, Ada].

Trivially, function fn is not injective and, thus, its inverse is not a function. The framework of [24] introduces a Landauer embedding to make term rewriting reversible, which is then mapped to an *injectivization* transformation on (conditional) term rewrite systems. For the above function fn, it would return the following injective version:

$$\begin{array}{l}\mathsf{fn}^{\mathsf{i}}\ [\,] = \langle [\,], \beta_1\rangle\\ \mathsf{fn}^{\mathsf{i}}\ ((\mathsf{Name}\ n\ l){:}xs) = \langle n{:}ys,\ \beta_2\ l\ ws\rangle\ \ \mathsf{where}\ \langle ys, ws\rangle = \mathsf{fn}^{\mathsf{i}}\ xs\\ \mathsf{fn}^{\mathsf{i}}\ ((\mathsf{City}\ c){:}xs) = \langle ys,\ \beta_3\ c\ ws\rangle\ \ \mathsf{where}\ \langle ys, ws\rangle = \mathsf{fn}^{\mathsf{i}}\ xs\end{array}$$

In contrast to the original function, the inversion of function fn^{i} can easily be obtained by switching the left- and right-hand sides of every equation (see Def. 2). Here, the call fn^{i} [Name John Smith, City London, Name Ada Lovelace] now returns $\langle[\mathsf{John}, \mathsf{Ada}], \beta_2\ \mathsf{Smith}\ (\beta_3\ \mathsf{London}\ (\beta_2\ \mathsf{Lovelace}\ (\beta_1)))\rangle$.

The net effect is essentially equivalent to the introduction of a complement in the syntactic bidirectionalization approach of Matsuda et al. [22]. There, a complement function is first derived, which is then merged with the original function using tupling. While [24] considers a slightly more general class of programs (as

[3] In this section, we denote programs using a Haskell-like notation, but they will be specified using conditional term rewrite systems in the remainder of the paper.

[4] As it is common practice, we use ":" and [ ] as list constructors.

we do), [22] defines several optimizations to avoid introducing unnecessary symbols in the computed complements (which might improve the number of updates their "put" function can deal with).

In this paper, we present a syntactic bidirectionalization technique based on the injectivization and inversion transformations of [24]. We prove that the so called GetPut law (using the terminology from the literature on *lenses* [11]) always holds for our bidirectional transformations, while the PutGet and PutPut laws hold for *compatible* view updates only. Then, we introduce the use of narrowing to formally characterize the class of view updates that are compatible. Finally, we consider other possible situations—namely, view updates that are not compatible—and discuss some possible approaches to deal with them.

2 Term Rewriting

In this paper, we will use (conditional) term rewrite systems to specify first-order functional programs. Therefore, in this section, we recall some basic concepts of term rewriting. We refer the reader to, e.g., [2, 29] for further details.

Terms and Substitutions. A *signature* $\mathcal{F}$ is a set of ranked function symbols (i.e., function symbols with an associated arity). Given a set of variables $\mathcal{V}$ with $\mathcal{F} \cap \mathcal{V} = \varnothing$, we denote the domain of *terms* by $\mathcal{T}(\mathcal{F}, \mathcal{V})$. We use $\mathsf{f}, \mathsf{g}, \ldots$ to denote function symbols and $x, y, \ldots$ to denote variables. Positions are used to address the nodes of a term viewed as a tree. A *position* p in a term t, in symbols $p \in \mathcal{P}\mathsf{os}(t)$, is represented by a finite sequence of natural numbers, where the empty sequence ϵ denotes the root position. We let $t|_p$ denote the *subterm* of t at position p and $t[s]_p$ the result of *replacing the subterm* $t|_p$ by the term s. $\mathcal{V}ar(t)$ denotes the set of variables appearing in t. A term t is *ground* if $\mathcal{V}ar(t) = \varnothing$.

A *substitution* $\sigma : \mathcal{V} \mapsto \mathcal{T}(\mathcal{F}, \mathcal{V})$ is a mapping from variables to terms such that $\mathcal{D}\mathsf{om}(\sigma) = \{x \in \mathcal{V} \mid x \neq \sigma(x)\}$ is its domain. A substitution σ is *ground* if $\sigma(x)$ is ground for all $x \in \mathcal{D}\mathsf{om}(\sigma)$. Substitutions are extended to morphisms from $\mathcal{T}(\mathcal{F}, \mathcal{V})$ to $\mathcal{T}(\mathcal{F}, \mathcal{V})$ in the natural way. We denote the application of a substitution σ to a term t by $t\sigma$ (postfix notation). The identity substitution is denoted by id. We let "$\circ$" denote the composition of substitutions, i.e., $\sigma \circ \theta(x) = (x\theta)\sigma = x\theta\sigma$. The *restriction* $\theta\restriction_V$ of a substitution θ to a set of variables V is defined as follows: $x\theta\restriction_V = x\theta$ if $x \in V$ and $x\theta\restriction_V = x$ otherwise.

A substitution σ is *more general* than a substitution θ, denoted by $\sigma \leqslant \theta$, if there is a substitution δ such that $\delta \circ \sigma = \theta$. A *unifier* of two terms s and t is a substitution σ with $t\sigma = s\sigma$; furthermore, σ is the *most general unifier* of t and s, denoted by $\mathsf{mgu}(t, s)$ if, for every other unifier θ of t and s, we have $\sigma \leqslant \theta$.

Term Rewrite Systems. A set of rewrite rules $l \to r$ such that l is a nonvariable term and r is a term whose variables appear in l is called a *term rewrite system* (TRS for short); terms l and r are called the left-hand side and the right-hand side of the rule, respectively. We restrict ourselves to finite signatures and TRSs. Given a TRS $\mathcal{R}$ over a signature $\mathcal{F}$, the *defined* symbols $\mathcal{D}_\mathcal{R}$ are the root symbols

of the left-hand sides of the rules and the *constructors* are $\mathcal{C}_\mathcal{R} = \mathcal{F} \setminus \mathcal{D}_\mathcal{R}$. *Constructor terms* of $\mathcal{R}$ are terms over $\mathcal{C}_\mathcal{R}$ and $\mathcal{V}$, denoted by $\mathcal{T}(\mathcal{C}_\mathcal{R}, \mathcal{V})$. We sometimes omit $\mathcal{R}$ from $\mathcal{D}_\mathcal{R}$ and $\mathcal{C}_\mathcal{R}$ if it is clear from the context. A substitution σ is a *constructor substitution* (of $\mathcal{R}$) if $x\sigma \in \mathcal{T}(\mathcal{C}_\mathcal{R}, \mathcal{V})$ for all variables x.

In the following, we denote by $\overline{o_n}$ a sequence of elements $o_1, \ldots, o_n$ for some n. We write $\overline{o}$ when the number of elements is not relevant.

Given a TRS $\mathcal{R}$, we say that a term t is *basic* [17] if it has the form $\mathsf{f}(\overline{t_n})$ with $\mathsf{f} \in \mathcal{D}_\mathcal{R}$ a defined function symbol and $\overline{t_n} \in \mathcal{T}(\mathcal{C}_\mathcal{R}, \mathcal{V})$ constructor terms.

For a TRS $\mathcal{R}$, we define the associated rewrite relation $\to_\mathcal{R}$ as the smallest binary relation on terms satisfying the following: given terms $s, t \in \mathcal{T}(\mathcal{F}, \mathcal{V})$, we have $s \to_\mathcal{R} t$ iff there exist a position p in s, a rewrite rule $l \to r \in \mathcal{R}$, and a substitution σ such that $s|_p = l\sigma$ and $t = s[r\sigma]_p$; the rewrite step is sometimes denoted by $s \to_{p,l\to r} t$ to make explicit the position and rule used in this step. The instantiated left-hand side $l\sigma$ is called a *redex*. A term s is called *irreducible* or in *normal form* with respect to a TRS $\mathcal{R}$ if there is no term t with $s \to_\mathcal{R} t$. A *derivation* is a (possibly empty) sequence of rewrite steps. Given a binary relation $\to$, we denote by $\to^*$ its reflexive and transitive closure, i.e., $s \to^*_\mathcal{R} t$ means that s can be reduced to t in $\mathcal{R}$ in zero or more steps.

Programs. In this work, *programs* are denoted by so called *conditional* term rewrite systems (CTRSs) where rules have now the form $l \to r \Leftarrow C$ with C a *condition* (i.e., a sequence of equations). In particular, we consider *oriented* 3-CTRSs where $\mathcal{V}ar(r) \subseteq \mathcal{V}ar(l) \cup \mathcal{V}ar(C)$ and the equations are *oriented*, i.e., C has the form $s_1 \twoheadrightarrow t_1, \ldots, s_n \twoheadrightarrow t_n$ with $\twoheadrightarrow$ interpreted as *reachability* $\to^*_\mathcal{R}$. Also, we focus on a subclass of oriented 3-CTRSs called pcDCTRS [5, 23] ("pc" stands for *pure constructor*) where, for each conditional rule $l \to r \Leftarrow s_1 \twoheadrightarrow t_1, \ldots, s_n \twoheadrightarrow t_n$, the following conditions hold:

- l and $\overline{s_n}$ are basic terms and r and $\overline{t_n}$ are constructor terms, and
- $\mathcal{V}\mathsf{ar}(s_i) \subseteq \mathcal{V}\mathsf{ar}(l, \overline{t_{i-1}})$ for all $i = 1, \ldots, n$ (i.e., it is a *deterministic* CTRS; see below).

Finally, we also require the left-hand sides of the rules in a pcDCTRS to be non-overlapping, i.e., there is no pair of (different) rules $l_1 \to r_1 \Leftarrow C_1$ and $l_2 \to r_2 \Leftarrow C_2$ such that $l_1\sigma = l_2\sigma$ for some substitution σ. This is still quite a general class of CTRSs and it is particularly appropriate to represent typical (first-order) functional programs (e.g., so called *treeless* functional programs [31] or *extended top-down tree transducers* [27] can be seen as subclasses of pcDCTRSs). Intuitively speaking, a rule like $l \to r \Leftarrow s_1 \twoheadrightarrow t_1, \ldots, s_n \twoheadrightarrow t_n$ resembles a typical functional definition of the form

$$l_1 = r_1 \;\; \mathsf{where}\; t_1 = s_1, \ldots,\; t_n = s_n$$

Example 1. The definition of function fn in Section 1 can be specified using a pcDCTRS as follows:

$$\begin{aligned}
&\mathsf{fn}([\,]) \to [\,]\\
&\mathsf{fn}(\mathsf{name}(n, l) : xs) \to n : ys \Leftarrow \mathsf{fn}(xs) \twoheadrightarrow ys\\
&\mathsf{fn}(\mathsf{city}(c) : xs) \to ys \Leftarrow \mathsf{fn}(xs) \twoheadrightarrow ys
\end{aligned}$$

where the constructor symbols name and city are now denoted using small letters (in contrast to the Haskell-like notation used in Section 1).

Given a pcDCTRS $\mathcal{R}$, the associated (constructor-based) rewrite relation $\to_{\mathcal{R}}$ is defined as the smallest binary relation satisfying the following: given ground terms $s, t \in \mathcal{T}(\mathcal{F})$, we have $s \to_{\mathcal{R}} t$ iff there exist a position p in s with $s|_p$ a basic subterm, a rewrite rule $l \to r \Leftarrow \overline{s_n \twoheadrightarrow t_n} \in \mathcal{R}$, and a ground constructor substitution σ such that $s|_p = l\sigma$, $s_i\sigma \to_{\mathcal{R}}^* t_i\sigma$ for all $i = 1, \ldots, n$, and $t = s[r\sigma]_p$. Since the left-hand sides of a pcDCTRS are basic terms and there are no overlappings, every computation can be made deterministic by fixing a strategy (e.g., by always selecting the leftmost innermost redex).

Moreover, the fact that pcDCTRSs are *deterministic* CTRSs [13] (which does not necessarily imply that computations are deterministic) allows us to compute the bindings for the variables in the condition in a deterministic way. E.g., given a ground term s and a rule $l \to r \Leftarrow \overline{s_n \twoheadrightarrow t_n}$ with $s|_p = l\theta$, we have that $s_1\theta$ is ground. Therefore, one can reduce $s_1\theta$ to some term s_1' such that s_1' is an instance of $t_1\theta$ with some ground substitution θ_1. Now, we have that $s_2\theta\theta_1$ is ground and we can reduce $s_2\theta\theta_1$ to some term s_2' such that s_2' is an instance of $t_2\theta\theta_1$ with some ground substitution θ_2, and so forth. If all equations in the condition hold using $\theta_1, \ldots, \theta_n$, we have that $s \to s[r\sigma]_p$ with $\sigma = \theta\theta_1 \ldots \theta_n$.

Remark 1. In the remainder of this paper, we assume for simplicity that all defined symbols of the original pcDCTRS are *unary*. Note that any n-ary defined symbol can be trivially transformed into a unary function by putting all arguments into a fresh tuple symbol. Hence, in the following, program rules will have the following form: $\mathsf{f_0}(s) \to r \Leftarrow \mathsf{f_1}(s_1) \twoheadrightarrow t_1, \ldots, \mathsf{f_n}(s_n) \twoheadrightarrow t_n$.

3 Injectivization and Inversion

In this section, we mostly recall the injectivization and inversion transformations for pcDCTRSs introduced in [24] (with some slight modifications).

Definition 1 (injectivization). *Let $\mathcal{R}$ be a pcDCTRS. We produce a new CTRS $\mathbf{I}(\mathcal{R})$ by replacing each rule*

$$\mathsf{f_0}(s) \to r \Leftarrow \mathsf{f_1}(s_1) \twoheadrightarrow t_1, \ldots, \mathsf{f_n}(s_n) \twoheadrightarrow t_n$$

of $\mathcal{R}$ by a new rule of the form

$$\mathsf{f_0^i}(s) \to \langle r, \beta(\overline{y}, \overline{w_n})\rangle \Leftarrow \mathsf{f_1^i}(s_1) \twoheadrightarrow \langle t_1, w_1\rangle, \ldots, \mathsf{f_n^i}(s_n) \twoheadrightarrow \langle t_n, w_n\rangle$$

in $\mathbf{I}(\mathcal{R})$, where

- *$\mathsf{f_0^i}, \ldots, \mathsf{f_n^i} \in \mathcal{D}_{\mathbf{I}(\mathcal{R})}$ are fresh (not necessarily different) defined function symbols with $\mathsf{f_j^i} = \mathsf{f_k^i}$ iff $\mathsf{f_j} = \mathsf{f_k}$, for all j, k,*
- *$\beta \in \mathcal{C}_{\mathbf{I}(\mathcal{R})}$ is a fresh constructor symbol,*
- *$\{\overline{y}\} = (\mathcal{V}\mathsf{ar}(s) \backslash \mathcal{V}\mathsf{ar}(r, \overline{s_n}, \overline{t_n})) \cup \bigcup_{i=1}^{n} \mathcal{V}\mathsf{ar}(t_i) \backslash \mathcal{V}\mathsf{ar}(r, s_{i+1}, \ldots, s_n)$,*

– *and* $\overline{w_n}$ *are fresh variables.*

We assume that the variables of $\overline{y}$ *are in lexicographic order. Clearly, we have* $\mathcal{D}_{\mathbf{I}(\mathcal{R})} = \{\mathsf{f}^{\mathsf{i}} \mid \mathsf{f} \in \mathcal{D}_{\mathcal{R}}\}$ *and* $\mathcal{C}_{\mathbf{I}(\mathcal{R})} = \mathcal{C}_{\mathcal{R}} \cup \{\langle\,\rangle\} \cup \{\beta \mid l \to \langle \_, \beta(\ldots)\rangle \Leftarrow C \in \mathbf{I}(\mathcal{R})\}$.

Intuitively speaking, the β symbols are needed to know the applied rule, so that the backward steps are computationally deterministic. The variables in $\{\overline{y}\}$ are the variables that are *erased* in the rule (i.e., they are in the left-hand side but not in the corresponding right-hand side or the condition) as well as the variables that are needed for the inverse rule to be deterministic (a more detailed explanation and some examples can be found in [24]).

Now, given a term $\mathsf{f}(s)$ that reduces to a normal form v in a pcDCTRS $\mathcal{R}$, we have that $\mathsf{f}^{\mathsf{i}}(s)$ reduces to a normal form $\langle v, \pi\rangle$ in $\mathbf{I}(\mathcal{R})$, where π is called the *complement* of the reduction. Although our development originates from the introduction of a *Landauer embedding* [20] to make reductions reversible, complements are similar to the ones obtained by defining a *view complement function* as in [22] (and originally introduced in [3]). Indeed, [22] applies a separated stage of *tupling* [7] to combine the original function and its complement, while this is naturally embedded into the definition above.

Example 2. Let $\mathcal{R}$ be a pcDCTRS defining the function fn of Example 1. Then, we have that $\mathbf{I}(\mathcal{R})$ is defined by the following rules:

$$\begin{array}{l}
\mathsf{fn}^{\mathsf{i}}([\,]) \to \langle [\,], \beta_1\rangle \\
\mathsf{fn}^{\mathsf{i}}(\mathsf{name}(n,l) : xs) \to \langle n : ys, \beta_2(l, ws)\rangle \Leftarrow \mathsf{fn}^{\mathsf{i}}(xs) \twoheadrightarrow \langle ys, ws\rangle \\
\mathsf{fn}^{\mathsf{i}}(\mathsf{city}(c) : xs) \to \langle ys, \beta_3(c, ws)\rangle \Leftarrow \mathsf{fn}^{\mathsf{i}}(xs) \twoheadrightarrow \langle ys, ws\rangle
\end{array}$$

E.g., the normal form of $\mathsf{fn}^{\mathsf{i}}([\mathsf{name}(\mathsf{john}, \mathsf{smith}), \mathsf{city}(\mathsf{london}), \mathsf{name}(\mathsf{ada}, \mathsf{lovelace})])$ is $\langle[\mathsf{john}, \mathsf{ada}], \beta_2(\mathsf{smith}, \beta_3(\mathsf{london}, \beta_2(\mathsf{lovelace}, \beta_1))))\rangle$, as expected.

The inversion of an injectivized system now amounts to switching the left- and right-hand sides of the rule and of every equation in the condition, as follows:

Definition 2 (inversion). *Let* $\mathcal{R}$ *be a pcDCTRS and* $\mathcal{R}_f = \mathbf{I}(\mathcal{R})$ *be its injectivization. The inverse system* $\mathcal{R}_b = \mathbf{I}^{-1}(\mathcal{R}_f)$ *is obtained from* $\mathcal{R}_f$ *by replacing each rule*

$$\mathsf{f}^{\mathsf{i}}_0(s) \to \langle r, \beta(\overline{y}, \overline{w_n})\rangle \Leftarrow \mathsf{f}^{\mathsf{i}}_1(s_1) \twoheadrightarrow \langle t_1, w_1\rangle, \ldots, \mathsf{f}^{\mathsf{i}}_n(s_n) \twoheadrightarrow \langle t_n, w_n\rangle$$

of $\mathcal{R}_f$ *by a new rule of the form*

$$\mathsf{f}^{-1}_0(r, \beta(\overline{y}, \overline{w_n})) \to s \Leftarrow \mathsf{f}^{-1}_n(t_n, w_n) \twoheadrightarrow s_n, \ldots, \mathsf{f}^{-1}_1(t_1, w_1) \twoheadrightarrow s_1$$

in $\mathcal{R}_b$*, where* $\mathsf{f}^{-1}_0, \ldots, \mathsf{f}^{-1}_n \in \mathcal{D}_{\mathcal{R}_b}$ *are fresh (not necessarily different) defined function symbols with* $\mathsf{f}^{-1}_j = \mathsf{f}^{-1}_k$ *iff* $\mathsf{f}^{\mathsf{i}}_j = \mathsf{f}^{\mathsf{i}}_k$*, for all* j, k*. Here, we have* $\mathcal{D}_{\mathcal{R}_b} = \{\mathsf{f}^{-1} \mid \mathsf{f} \in \mathcal{R}\}$ *and* $\mathcal{C}_{\mathcal{R}_b} = \mathcal{C}_{\mathcal{R}_f}$.

The correctness of both injectivization and inversion, as well as the fact that $\mathbf{I}(\mathcal{R})$ and $\mathbf{I}^{-1}(\mathbf{I}(\mathcal{R}))$ are also pcDCTRSs, can be found in [24].

Example 3. Inversion of our running example (function fn^{i} in Ex. 2) is as follows:

$$\begin{aligned}
&\mathsf{fn}^{-1}([\,], \beta_1) \to [\,]\\
&\mathsf{fn}^{-1}(n\!:\!ys, \beta_2(l, ws)) \to \mathsf{name}(n, l)\!:\!xs \Leftarrow \mathsf{fn}^{-1}(ys, ws) \twoheadrightarrow xs\\
&\mathsf{fn}^{-1}(ys, \beta_3(c, ws)) \to \mathsf{city}(c)\!:\!xs \Leftarrow \mathsf{fn}^{-1}(ys, ws) \twoheadrightarrow xs
\end{aligned}$$

Remark 2. In the following, we let $\mathsf{bx}(\mathcal{R}) = \mathcal{R} \cup \mathbf{I}(\mathcal{R}) \cup \mathbf{I}^{-1}(\mathbf{I}(\mathcal{R}))$. Moreover, given a function $\mathsf{f} \in \mathcal{D}_{\mathcal{R}}$, we let $\mathsf{f}^{\mathsf{i}} \in \mathcal{D}_{\mathbf{I}(\mathcal{R})}$ denote its injectivization and $\mathsf{f}^{-1} \in \mathcal{D}_{\mathbf{I}^{-1}(\mathbf{I}(\mathcal{R}))}$ the inversion of f^{i}.

4 A Framework for Syntactic Bidirectionalization

In this section, we present a framework for bidirectionalization where narrowing is used to characterize compatible view updates.

4.1 Bidirectionalization

Our bidirectionalization is based on the injectivization and inversion transformations from Section 3. Let $\mathcal{R}$ be a pcDCTRS and let $\mathsf{f} \in \mathcal{D}_{\mathcal{R}}$ be a function such that

$$\mathsf{f}(s) \to^*_{\mathsf{bx}(\mathcal{R})} v$$

for some constructor terms $s, v \in \mathcal{T}(\mathcal{C}_{\mathcal{R}})$. By construction, there exists a function f^{i} in $\mathsf{bx}(\mathcal{R})$ such that

$$\mathsf{f}^{\mathsf{i}}(s) \to^*_{\mathsf{bx}(\mathcal{R})} \langle v, \pi \rangle$$

where π (a constructor term) is called the *complement* of the derivation. Conversely, there is also a function f^{-1} in $\mathsf{bx}(\mathcal{R})$ such that

$$\mathsf{f}^{-1}(v, \pi) \to^*_{\mathsf{bx}(\mathcal{R})} s$$

Following the terminology in the bx literature, if f is a "get" function that takes a source and returns a view, we can automatically derive a corresponding "put" function following the so called *constant complement approach* [3] (as in [24]):

Definition 3 (put generation). *Let $\mathcal{R}$ be a pcDCTRS. Given a function $\mathsf{f} \in \mathcal{D}_{\mathcal{R}}$, the corresponding "$\mathsf{put}$" function, in symbols, $\mathsf{put}_{\mathsf{f}}$, is defined as follows:*

$$\mathsf{put}_{\mathsf{f}}(v, s) \to s' \Leftarrow \mathsf{f}^{\mathsf{i}}(s) \twoheadrightarrow \langle v', \pi \rangle, \mathsf{f}^{-1}(v, \pi) \twoheadrightarrow s'$$

where s, s', v and d' are variables that range over the constructor terms of the original pcDCTRS $\mathcal{R}$, i.e., $\mathcal{T}(\mathcal{C}_{\mathcal{R}})$, while π is a variable that ranges over $\mathcal{T}(\mathcal{C}_{\mathbf{I}(\mathcal{R})})$ to also account for the β symbols introduced in the injectivization stage.

For instance, the corresponding put function for function fn in Example 1 will be defined as follows:

$$\mathsf{put}_{\mathsf{fn}}(v, s) \to s' \Leftarrow \mathsf{fn}^{\mathsf{i}}(s) \twoheadrightarrow \langle v', \pi \rangle, \mathsf{fn}^{-1}(v, \pi) \twoheadrightarrow s'$$

where functions fn^{i} and fn^{-1} are defined in Examples 2 and 3, respectively.

Remark 3. In the following, we assume that $\mathsf{bx}(\mathcal{R})$ also includes $\mathsf{put_f}$ for all $\mathsf{f} \in \mathcal{D}_\mathcal{R}$, as defined above.

In the following, we prove the usual properties for for the bidirectional transformations obtained by our bidirectionalization technique. Basically, we prove that each function f and the corresponding $\mathsf{put_f}$ form a so called (very) *well-behaved* lens [11]. Let us start with the following essential property:

Theorem 1 (GetPut). *Let $\mathcal{R}$ be a pcDCTRS. Then, for all defined function $\mathsf{f} \in \mathcal{D}_\mathcal{R}$ in $\mathcal{R}$ and for all ground constructor term $s \in \mathcal{T}(\mathcal{C}_\mathcal{R})$, if the normal form of $\mathsf{f}(s)$ is a constructor term, then we have $\mathsf{put_f}(\mathsf{f}(s), s) \to^*_{\mathsf{bx}(\mathcal{R})} s$.*

Proof. Let us consider a derivation $\mathsf{f}(s) \to^*_\mathcal{R} v$ such that v is a constructor term. By definition of $\mathsf{put_f}$, we have to perform first a subcomputation for $\mathsf{f^i}(s)$. Since $\mathsf{f^i}$ represents a conservative extension of f, we have $\mathsf{f^i}(s) \to^*_{\mathsf{bx}(\mathcal{R})} \langle v, \pi\rangle$ for some π. Finally, by the correctness of the inversion transformation [24, Theorem 36], we have $\mathsf{f}^{-1}(v, \pi) \to^*_{\mathsf{bx}(\mathcal{R})} s$. □

Example 4. Consider function fn from our running example and the term $s = [\mathsf{name}(\mathsf{john}, \mathsf{smith}), \mathsf{city}(\mathsf{london}), \mathsf{name}(\mathsf{ada}, \mathsf{lovelace})]$. Here, the normal form of $\mathsf{fn}(s)$ is $[\mathsf{john}, \mathsf{ada}]$. Then, $\mathsf{put_{fn}}([\mathsf{john}, \mathsf{ada}], s)$ reduces to s since

$$\begin{aligned}&\mathsf{fn^i}([\mathsf{name}(\mathsf{john}, \mathsf{smith}), \mathsf{city}(\mathsf{london}), \mathsf{name}(\mathsf{ada}, \mathsf{lovelace})])\\ &\quad\to^* \langle[\mathsf{john}, \mathsf{ada}], \beta_2(\mathsf{smith}, \beta_3(\mathsf{london}, \beta_2(\mathsf{lovelace}, \beta_1)))\rangle\end{aligned}$$

and

$$\mathsf{fn}^{-1}([\mathsf{john}, \mathsf{ada}], \beta_2(\mathsf{smith}, \beta_3(\mathsf{london}, \beta_2(\mathsf{lovelace}, \beta_1)))) \to^* s$$

Other properties, though, do not always hold, since the generated put functions are not always defined. For instance, roughly speaking, the PutGet law states that $\mathsf{f}(\mathsf{put_f}(v, s)) \to^*_{\mathsf{bx}(\mathcal{R})} v$. This law does not hold in general:

Example 5. Consider again function fn from our running example, together with the derivation $\mathsf{fn^i}([\mathsf{name}(\mathsf{john}, \mathsf{smith})]) \to^* \langle[\mathsf{john}], \beta_2(\mathsf{smith}, \beta_1)\rangle$. Here, the term $\mathsf{fn}^{-1}([\,], \beta_2(\mathsf{smith}, \beta_1))$ cannot be reduced to a constructor term. Hence, the PutGet law does not hold, i.e., $\mathsf{f}(\mathsf{put_{fn}}([\,], [\mathsf{name}(\mathsf{john}, \mathsf{smith})])$ is not reduced to $[\,]$.

In the above example, the problem comes from the fact that the view $[\,]$ and the source $[\mathsf{name}(\mathsf{john}, \mathsf{smith})]$ are not *compatible*.[5] In this case, a view with exactly one name is required. Thus, in the following, we will consider *partial* versions of some laws [11] as in, e.g., [3, 22].

The notion of compatibility is formalized as follows:

Definition 4 (compatible view). *Let $\mathcal{R}$ be a pcDCTRS. We say that a term v (a view) is* compatible *with a term s (a source) w.r.t. a function $\mathsf{f} \in \mathcal{D}_\mathcal{R}$ if $\mathsf{put_f}(v, s)$ can be reduced to a constructor term in $\mathsf{bx}(\mathcal{R})$.*

[5] Sometimes, we also say that a view is compatible with a given complement since the complement is fully determined by the source.

For instance, given $s = [\mathsf{name}(\mathsf{john}, \mathsf{smith}), \mathsf{city}(\mathsf{london}), \mathsf{name}(\mathsf{ada}, \mathsf{lovelace})]$, the view $v = [\mathsf{rose}, \mathsf{ada}]$ is compatible with s, while $[\mathsf{ada}]$ is not; here, only lists with two elements are compatible with s.

Now, we can prove the PutGet law for compatible view updates:

Theorem 2 (PutGet). *Let $\mathcal{R}$ be a pcDCTRS, $\mathsf{f} \in \mathcal{D}_\mathcal{R}$ a defined function and $s \in \mathcal{T}(\mathcal{C}_\mathcal{R})$ a ground constructor term. Then, $\mathsf{f}(\mathsf{put}_\mathsf{f}(v, s)) \to^*_{\mathsf{bx}(\mathcal{R})} v$ for all constructor term v that is compatible with s w.r.t. f.*

Proof. Since v is compatible with s w.r.t. f, we have that $\mathsf{put}_\mathsf{f}(v, s)$ reduces to a constructor normal form, say s'. Hence, there exist reductions $\mathsf{f}^\mathsf{i}(s) \to^* \langle v'', \pi\rangle$ and $\mathsf{f}^{-1}(v, \pi) \to^* s'$. Now, since $\mathsf{f}^{-1}(v, \pi) \to^* s'$, by the correctness of the inversion transformation, we have that $\mathsf{f}^\mathsf{i}(s') \to^* \langle v, \pi\rangle$. Moreover, by the correctness of the injectivization transformation, we have $\mathsf{f}(s') \to^* v$, and the claim follows. □

Example 6. Consider again function fn and the terms

$$s = [\mathsf{name}(\mathsf{john}, \mathsf{smith}), \mathsf{city}(\mathsf{london}), \mathsf{name}(\mathsf{ada}, \mathsf{lovelace})]$$

and $v = [\mathsf{john}, \mathsf{ada}]$ from Example 4. Given an updated (compatible) view $v' = [\mathsf{rose}, \mathsf{ada}]$, we have $\mathsf{fn}(\mathsf{put}_\mathsf{fn}(v', s)) \to^* v'$ since $\mathsf{put}_\mathsf{fn}(v', s) \to^* s'$ with $s' = [\mathsf{name}(\mathsf{rose}, \mathsf{smith}), \mathsf{city}(\mathsf{london}), \mathsf{name}(\mathsf{ada}, \mathsf{lovelace})]$ and $\mathsf{fn}(s') \to^* v'$, as expected.

The following result also holds for compatible views. In the following, we say that two terms are *joinable* if they can be reduced to the same constructor term.

Theorem 3 (PutPut). *Let $\mathcal{R}$ be a pcDCTRS, $\mathsf{f} \in \mathcal{D}_\mathcal{R}$ a defined function and $s \in \mathcal{T}(\mathcal{C}_\mathcal{R})$ a ground constructor term. Then, $\mathsf{put}_\mathsf{f}(v_1, \mathsf{put}_\mathsf{f}(v_2, s))$ and $\mathsf{put}_\mathsf{f}(v_1, s)$ are joinable for all constructor terms v_1, v_2 that are compatible with s w.r.t. f.*

Proof. Since v_2 is compatible with s w.r.t. f, we have that $\mathsf{put}_\mathsf{f}(v_2, s)$ reduces to a constructor normal form, say s'. Hence, there exist reductions $\mathsf{f}^\mathsf{i}(s) \to^* \langle v, \pi\rangle$ and $\mathsf{f}^{-1}(v_2, \pi) \to^* s'$. Now, since $\mathsf{f}^{-1}(v_2, \pi) \to^* s'$, by the correctness of the inversion transformation, we have that $\mathsf{f}^\mathsf{i}(s') \to^* \langle v_2, \pi\rangle$. Therefore, and this is the key point for proving this property, the computed complement, π, is the same for both s and s'. Then, trivially, we have that v_1 is also compatible with s' w.r.t. f. Hence, $\mathsf{put}_\mathsf{f}(v_1, s')$ reduces to a constructor normal form, say s'', and, thus, we have reductions $\mathsf{f}^\mathsf{i}(s') \to^* \langle v_2, \pi\rangle$ and $\mathsf{f}^{-1}(v_1, \pi) \to^* s''$.

On the other hand, since v_1 is compatible with s w.r.t. f, we have that $\mathsf{put}_\mathsf{f}(v_1, s)$ reduces to the same constructor normal form, s'', since $\mathsf{f}^\mathsf{i}(s) \to^* \langle v, \pi\rangle$ and $\mathsf{f}^{-1}(v_1, \pi) \to^* s''$, as seen before. □

Example 7. Consider again function fn and the terms

$$s = [\mathsf{name}(\mathsf{john}, \mathsf{smith}), \mathsf{city}(\mathsf{london}), \mathsf{name}(\mathsf{ada}, \mathsf{lovelace})]$$

$v_1 = [\mathsf{rose}, \mathsf{ada}]$ and $v_2 = [\mathsf{john}, \mathsf{paul}]$. Here, we have $\mathsf{put}_\mathsf{fn}(v_1, s) \to^* s_1$, with $s_1 = [\mathsf{name}(\mathsf{rose}, \mathsf{smith}), \mathsf{city}(\mathsf{london}), \mathsf{name}(\mathsf{ada}, \mathsf{lovelace})]$. On the other hand, we have $\mathsf{put}_\mathsf{fn}(v_2, s) \to^* s_2$, with $s_2 = [\mathsf{name}(\mathsf{john}, \mathsf{smith}), \mathsf{city}(\mathsf{london}), \mathsf{name}(\mathsf{paul}, \mathsf{lovelace})]$ and $\mathsf{put}_\mathsf{fn}(v_1, s_2) \to^* s_1$.

The above result ensures that our put functions do not have undesirable side-effects on the source. In other words, the complement associated to the updated source obtained by a put function will still be the same, no matter the (compatible) view used. A bidirectional transformation fulfilling the above laws is called—in the lenses approach [11]—a *partial very well-behaved lens*. Note that it is "partial" since it is only very well behaved for compatible view updates.

Moreover, when both PutPut and GetPut laws hold, we have

$$\mathsf{put}_{\mathsf{f}}(\mathsf{f}(s), \mathsf{put}_{\mathsf{f}}(v, s)) \to^* s$$

i.e., the effects of a view update can always be undone (see, e.g., [12]). Intuitively speaking, the reason for this behaviour in our context is that the computed complement is the same for s and for $\mathsf{put}_{\mathsf{f}}(v, s)$ when v is a compatible view update, as mentioned before.

4.2 Using Narrowing to Characterize Compatible Updates

Let us first briefly introduce the *narrowing* principle [28, 19]. It mainly extends term rewriting by replacing pattern matching with unification, so that terms containing logic (i.e., *free*) variables can be (non-deterministically) reduced.

Example 8. Consider the term $\mathsf{fn}^{-1}(x, \beta_1)$ and the rule "$\mathsf{fn}^{-1}([\,], \beta_1) \to [\,]$". While the term cannot be reduced using rewriting, narrowing performs the following step: $\mathsf{fn}^{-1}(x, \beta_1) \leadsto_{\{x \mapsto [\,]\}} [\,]$, where $\{x \mapsto [\,]\}$ is a unifier between the term and the left-hand side of the rule.

Now, we present narrowing for pcDCTRSs. For this class of programs, one can naturally extend Bockmayr's *conditional rewriting without evaluation of the premise* [6] to narrowing as follows. In the following, a *goal* is a sequence of equations of the form $s_1 \twoheadrightarrow t_1, \ldots, s_n \twoheadrightarrow t_n$, where each s_i is either basic or a constructor term and $t_1, \ldots, t_n$ are constructor terms.

Definition 5 (constructor-based conditional narrowing).
Let $\mathcal{R}$ be a pcDCTRS. Constructor-based conditional narrowing is defined as the smallest relation satisfying the transition rules of Fig. 1, where $(l \to r \Leftarrow C) \ll \mathcal{R}$ denotes that $l \to r \Leftarrow C$ is a copy of a rule in $\mathcal{R}$ renamed with fresh variables, and $\mathit{inn}(s)$ selects the position of a basic subterm (i.e., a term of the form $\mathsf{f}(\overline{t_n})$ with f a defined function symbol and $\overline{t_n}$ constructor terms).

Intuitively speaking, given a goal $s_1 \twoheadrightarrow t_1, \ldots, s_n \twoheadrightarrow t_n$, we proceed either by unifying the leftmost equation when s_1 is a constructor term and $\mathsf{mgu}(s_1, t_1)$ exists (rule *unification*) or we apply a narrowing step (rule *narrowing*). In the latter case, we select a basic subterm $s_1|_p$ of s_1 that unifies with the left-hand side of a (renamed) rule, say $l \to r \Leftarrow C$, using mgu σ, and return a new goal $(C, s_1[r]_p \twoheadrightarrow t_1, \ldots, s_n \twoheadrightarrow t_n)\sigma$.

Let us note that narrowing is often non-deterministic due to the free variables in goals, since the selected subterm might unify with the left-hand sides

(unification)

$$\frac{n > 1 \ \wedge \ s_1 \in \mathcal{T}(\mathcal{C}, \mathcal{V}) \ \wedge \ \sigma = \mathsf{mgu}(s_1, t_1)}{(s_1 \twoheadrightarrow t_1, \ldots, s_n \twoheadrightarrow t_n) \ \leadsto_\sigma \ (s_2 \twoheadrightarrow t_2, \ldots, s_n \twoheadrightarrow t_n)\sigma}$$

(narrowing)

$$\frac{p = inn(s_1) \ \wedge \ (l \to r \Leftarrow C) \ll \mathcal{R} \ \wedge \ \sigma = \mathsf{mgu}(s_1|_p, l)}{(s_1 \twoheadrightarrow t_1, \ldots, s_n \twoheadrightarrow t_n) \ \leadsto_\sigma \ (C, s_1[r]_p \twoheadrightarrow t_1, \ldots, s_n \twoheadrightarrow t_n)\sigma}$$

Fig. 1. Constructor-based conditional narrowing

of several rules (so rule *narrowing* gives rise to some branching). Moreover, narrowing derivations might be infinite even when the rules of a pcDCTRS are terminating (see, e.g., [25]). Indeed, our constructor-based conditional narrowing over pcDCTRSs is essentially equivalent to SLD resolution over equivalent logic programs [21].

In order to narrow a given term s, we start with a goal of the form $s \twoheadrightarrow x$, where x is a fresh variable. A *successful* narrowing derivation for s has the form $(s \twoheadrightarrow x) \leadsto_\sigma^* (t \twoheadrightarrow x)$ with t a constructor term; here, we say that $\sigma\restriction_{\mathcal{V}\mathsf{ar}(s)}$ is the *computed answer substitution* of the successful derivation.

Definition 6 (success set, $\mathsf{nwing}_\mathcal{R}$). *Let $\mathcal{R}$ be a pcDCTRS. We denote by $\mathsf{nwing}_\mathcal{R}(s)$ the success set of a term s in $\mathcal{R}$, where $\sigma \in \mathsf{nwing}_\mathcal{R}(s)$ if there is a successful narrowing derivation for the goal $s \twoheadrightarrow x$ in $\mathcal{R}$ with computed answer substitution σ, where x is a fresh variable.*

The following auxiliary result is useful to define the notion of *view skeleton*.

Lemma 1. *Let $\mathcal{R}$ be a pcDCTRS and $\mathsf{f} \in \mathcal{D}$ be a function with $\mathsf{f}^\mathsf{i}(s) \to_{\mathsf{bx}(\mathcal{R})}^* \langle v, \pi \rangle$ for some constructor terms s, v, and π. Then, $\mathsf{nwing}_{\mathsf{bx}(\mathcal{R})}(\mathsf{f}^{-1}(x, \pi))$ is a singleton up to variable renaming, where x is a fresh variable.*

Proof. By construction, the β symbol in the right-hand side of every rule of f^i is different. Therefore, the rules defining f^{-1} have also basic terms in the left-hand sides and, moreover, they are non-overlapping. Hence, since π has been obtained using the rules of an injectivized function f^i, any computation for $\mathsf{f}^{-1}(x, \pi)$ must be computationally deterministic even for narrowing since π is ground and, thus, can only unify (at most) with the left-hand side of one rule. Termination is straightforward since every rule of f^{-1} removes one β symbol with every narrowing step and the number of β symbols in π is finite. □

We observe that the above result does not hold when some β symbols are removed using an optimization like that in [22] which is based on an injectivity analysis. In our approach, the β symbols are essential to drive the narrowing steps and ensure that the derivation is finite and computationally deterministic, no matter if the original function is injective or not.

Definition 7 (view skeleton). *Let $\mathcal{R}$ be a pcDCTRS, $\mathsf{f} \in \mathcal{D}$ be a defined function and s be a constructor term. The view skeleton associated to f and s in*

$\mathsf{bx}(\mathcal{R})$ *is defined as follows:*

$$\mathsf{skel}_{\mathsf{bx}(\mathcal{R})}(\mathsf{f}, s) = x\sigma \text{ where } \mathsf{f}^{\mathsf{i}}(s) \to^* \langle v, \pi\rangle \text{ and } \mathsf{nwing}_{\mathsf{bx}(\mathcal{R})}(\mathsf{f}^{-1}(x, \pi)) = \{\sigma\}$$

with x a fresh variable and v, π constructor terms.

Now, we can precisely characterize the view updates that are compatible with a given source:

Lemma 2. *Let $\mathcal{R}$ be a pcDCTRS, $\mathsf{f} \in \mathcal{D}$ be a defined function, and s be a constructor term. Let $\mathsf{skel}_{\mathsf{bx}(\mathcal{R})}(\mathsf{f}, s) = v''$. Then, a constructor term v' is compatible with s w.r.t. f iff there exists a substitution θ such that $v' = v''\theta$.*

Proof. Let us proceed with the "if" part. Since $\mathsf{skel}_{\mathsf{bx}(\mathcal{R})}(\mathsf{f}, s) = v''$, by definition, we have $\mathsf{nwing}_{\mathsf{bx}(\mathcal{R})}(\mathsf{f}^{-1}(x, \pi)) = \{\sigma\}$, where x is a fresh variable and $v'' = x\sigma$. Therefore, there exists a narrowing derivation of the form $\mathsf{f}^{-1}(x, \pi) \twoheadrightarrow y \leadsto^*_\sigma u \twoheadrightarrow y$, where y is a fresh variable and u is a constructor term. By the soundness of narrowing, we have $\mathsf{f}^{-1}(x\sigma, \pi) \to^* u$. Finally, since rewriting is closed under substitution, we can conclude $\mathsf{f}^{-1}(x\sigma\theta, \pi) = \mathsf{f}^{-1}(v', \pi) \to^* u\theta$ with $u\theta$ a constructor term (note that $\pi\theta = \pi$ since π is ground), thus v' is compatible with s w.r.t. function f.

The "only if" part proceeds analogous, but uses the completeness of narrowing instead of its soundness. □

Informally speaking, a view skeleton represents the constructors that depend on the *history* of the considered computation, which is represented by a given complement. Variables in a skeleton represent information that is independent of the reduction steps and, thus, the same complement can be used by function f^{-1} in the definition of put_f. In some sense, our notion of skeleton is related to the use of polymorphic functions in the *semantic* approach to bidirectionalization [30], though our approach is in principle rather different.

Example 9. Consider again our running example and function fn as well as its associated functions fn^{i} and fn^{-1}. Given the source (constructor) term

$$s = [\mathsf{name}(\mathsf{john}, \mathsf{smith}), \mathsf{city}(\mathsf{london}), \mathsf{name}(\mathsf{ada}, \mathsf{lovelace})]$$

we have the following reduction:

$$\mathsf{fn}^{\mathsf{i}}(s) \to^* \langle [\mathsf{john}, \mathsf{ada}], \pi\rangle \text{ with } \pi = \beta_2(\mathsf{smith}, \beta_3(\mathsf{london}, \beta_2(\mathsf{lovelace}, \beta_1)))$$

Here, the view skeleton associated to s w.r.t. fn is $[n_1, n_2]$ since we have the following narrowing derivation:

$$\begin{aligned}&\mathsf{f}^{-1}(x, \pi) \twoheadrightarrow y\\ &\quad \leadsto^*_{\{x \mapsto [n_1, n_2]\}} [\mathsf{name}(n_1, \mathsf{smith}), \mathsf{city}(\mathsf{london}), \mathsf{name}(n_2, \mathsf{lovelace})] \twoheadrightarrow y\end{aligned}$$

Therefore, we have that a view update like $[\mathsf{richard}, \mathsf{ada}]$ will be compatible with s (since it is an instance of $[n_1, n_2]$), while a view update like $[\mathsf{john}]$ will not.

4.3 Dealing with Non-Compatible View Updates

So far we have only considered compatible (often called "in-place") view updates. A challenging topic for future work involves dealing with view updates which are not compatible. This problem has been considered in the context of the lenses approach to bidirectional transformations (see, e.g., [4]), but we are not aware of any technique that deals with this issue in the context of bidirectionalization.

The problem with non-compatible view updates is that, in general, there are many non-deterministic possibilities to propagate the changes back to the source. Consider again our running example with function fn. Given the source $s = [\mathsf{name}(\mathsf{john}, \mathsf{smith}), \mathsf{city}(\mathsf{london})]$, we get the view $[\mathsf{john}]$ with complement $\beta_2(\mathsf{smith}, \beta_3(\mathsf{london}, \beta_1))$. Now, given an arbitrary modified view, say $[\mathsf{john}, \mathsf{rose}]$, a put function might return any of the following modified sources:

$$
\begin{aligned}
s_1 &= [\mathsf{name}(\mathsf{john}, \mathsf{smith}), \mathsf{name}(\mathsf{rose}, \bot), \mathsf{city}(\mathsf{london})] \\
s_2 &= [\mathsf{name}(\mathsf{john}, \mathsf{smith}), \mathsf{city}(\mathsf{london}), \mathsf{name}(\mathsf{rose}, \bot)] \\
s_3 &= [\mathsf{name}(\mathsf{john}, \mathsf{smith}), \mathsf{name}(\mathsf{rose}, \bot)] \\
s_4 &= [\mathsf{name}(\mathsf{john}, \mathsf{smith}), \mathsf{city}(\mathsf{london}), \mathsf{city}(\bot), \mathsf{name}(\mathsf{rose}, \bot)] \\
&\dots
\end{aligned}
$$

where $\bot$ denotes an undefined value. In all cases, $\mathsf{fn}(s_i)$ reduces to $[\mathsf{john}, \mathsf{rose}]$, $i = 1, \dots, 4$, so all these alternatives might—in principle—be considered correct.

Furthermore, *aligning* the views so that the changes can be identified is also a difficult problem. Consider, e.g., that we change $[\mathsf{john}]$ to $[\mathsf{rose}]$. Here, one can assume that this is an in-place change and, thus, produce the source $[\mathsf{name}(\mathsf{rose}, \mathsf{smith}), \mathsf{city}(\mathsf{london})]$, but it could also be the result of a deletion and an insertion, so that the right source would be $[\mathsf{name}(\mathsf{rose}, \bot), \mathsf{city}(\mathsf{london})]$ instead. The larger the views, the more complex the aligment is. Some heuristics have been developed (see, e.g., [4]), but the approach has also some drawbacks (see the discussion in [9]).

Another approach by Diskin, Xiong, and Czarnecki [9] proposes to decompose the view update propagation into two separate operations: computing *deltas* (the differences between two data structures), and propagating deltas. In this context, the authors consider two operations, dget and dput, which are similar to the usual get and put operations from the standard approach but deal with deltas instead.

In our setting, we could specify a delta by means of a function (or a sequence of functions). For instance, the change from $[\mathsf{john}]$ to $[\mathsf{john}, \mathsf{rose}]$ could be specified by using a function that appends a new element at the end of a list, which is defined as follows:

$$
\begin{aligned}
&\mathsf{app_v}([\,], s) \to [s] \\
&\mathsf{app_v}(x : xs, s) \to x : \mathsf{app_v}(xs, s)
\end{aligned}
$$

so that the change from $[\mathsf{john}]$ to $[\mathsf{john}, \mathsf{rose}]$ is given by the following application: $\mathsf{app_v}([\mathsf{john}], \mathsf{rose})$. Then, one can apply dput to produce an equivalent function, $\mathsf{app_s}$, on the source:

$$
\begin{aligned}
&\mathsf{app_s}([\,], s) \to [\mathsf{name}(s, \bot)] \\
&\mathsf{app_s}(x : xs, s) \to x : \mathsf{app_s}(xs, s)
\end{aligned}
$$

In general, though, this is not the only possibility; namely, any function that appends the given name after all existing names in the source would be correct, no matter the position and number of cities. E.g., if the source is

$$s = [\mathsf{name}(\mathsf{john}, \mathsf{smith}), \mathsf{city}(\mathsf{london})]$$

we have that $\mathsf{app}_\mathsf{s}(s, \mathsf{rose})$ returns

$$[\mathsf{name}(\mathsf{john}, \mathsf{smith}), \mathsf{name}(\mathsf{rose}, \bot), \mathsf{city}(\mathsf{london})]$$

but it could also return

$$[\mathsf{name}(\mathsf{john}, \mathsf{smith}), \mathsf{city}(\mathsf{london}), \mathsf{name}(\mathsf{rose}, \bot)] \ .$$

Analogously to the case of put, there is some degree of non-determinism that must be fixed using either some user intervention or some heuristic.

We consider the delta-based framework a promising approach for future work.

5 Related Work

First, as mentioned before, the injectivization and inversion transformations for pcDCTRSs are taken from [24]. This paper, though, is concerned with reversible rewriting rather than bidirectional programming. The definition of a put function is sketched with an example (following the constant complement approach as in [12]) to show the potential of these transformations, but no law is formalized, compatibility of view updates is not considered, etc. Other, related approaches to program inversion in functional programming can be found in [15, 14]. See also [32] for more details on reversible programming languages.

The closest related work is the *syntactic* approach to the bidirectionalization of functional programs in [22]. While the basic technique shares some similarities with our development (both are based on the constant complement approach), our framework deals with more general programs (e.g., a function producing an inorder traversal of a binary tree can be represented with a pcDCTRS [31] while it cannot be represented with a *treeless* function as required in [22]); moreover, our notion of compatible view updates based on narrowing seems more useful than the *checker* of [22], since we produce a view *skeleton* which represents in a compact way all possible view updates by means of variables that denote updatable subterms.

Indeed, the use of narrowing to identify the parts of the view that are independent of the computed complement (and, thus, of the applied rules) is somehow similar to the requirement of polymorphic functions in the *semantic* approach to bidirectionalization of [30]. In particular, if a function is not polymorphic, our narrowing based approach will easily determine that no compatible view updates are possible. Consider, e.g., the following simple function to check if all the elements of a binary list are zero:

$$\begin{aligned} \mathsf{f}([\,]) &\to \mathsf{true} \\ \mathsf{f}(\mathsf{0} : xs) &\to \mathsf{true} \Leftarrow \mathsf{f}(xs) \twoheadrightarrow \mathsf{true} \\ \mathsf{f}(\mathsf{1} : xs) &\to \mathsf{false} \end{aligned}$$

Clearly, the type is $\mathsf{f} :: [Bin] \to Bool$, where Bin is a type with constructors $\mathsf{0}$ and $\mathsf{1}$ and $Bool$ is the usual Boolean type. Since the function is not polymorphic, one cannot apply the approach in [30]. Let us now consider our approach. First, injectivization and inversion return the following functions:

$$\begin{aligned}
\mathsf{f^i}([\,]) &\to \langle \mathsf{true}, \beta_1 \rangle \\
\mathsf{f}(\mathsf{0}:xs) &\to \langle \mathsf{true}, \beta_2(w) \rangle \Leftarrow \mathsf{f^i}(xs) \twoheadrightarrow \langle \mathsf{true}, w \rangle \\
\mathsf{f^i}(\mathsf{1}:xs) &\to \langle \mathsf{false}, \beta_3(xs) \rangle \\
\\
\mathsf{f^{-1}}(\mathsf{true}, \beta_1) &\to [\,] \\
\mathsf{f^{-1}}(\mathsf{true}, \beta_2(w)) &\to \mathsf{0}:xs \Leftarrow \mathsf{f^{-1}}(\mathsf{true}, w) \twoheadrightarrow xs \\
\mathsf{f^{-1}}(\mathsf{false}, \beta_3(xs)) &\to \mathsf{1}:xs
\end{aligned}$$

Given an arbitrary source, e.g., $s = [\mathsf{0}, \mathsf{1}, \mathsf{0}]$, we have $\mathsf{f^i}(s) \to^* \langle \mathsf{false}, \beta_2(\beta_3([\mathsf{0}])) \rangle$ and, then, $\mathsf{skel}_{\mathsf{bx}(\mathcal{R})}(\mathsf{f}, s)$ returns just false (since $\mathsf{nwing}_{\mathsf{bx}(\mathcal{R})}(\mathsf{f^{-1}}(x, \beta_2(\beta_3([\mathsf{0}]))))$ produces the computed answer $\{x \mapsto \mathsf{false}\}$). Therefore, no view different from false would be compatible.

There are other, related works that use narrowing in the context of bidirectional transformations [10, 16]. However, these works consider narrowing (or the *universal resolving algorithm* [1], which is essentially similar) as a mechanism for inverse computation. In their approach, no injectivization is performed and, thus, inverse computation might be non-deterministic. This is rather different to our approach, where inversion is only applied to injective functions and, moreover, narrowing is only applied to terms of the form $\mathsf{f^{-1}}(x, \pi)$, with π a ground constructor term, so that narrowing derivations are always finite and computationally deterministic.

6 Discussion

To summarize, we have presented a (syntactic) bidirectionalization technique based on some injectivization and inversion transformations, where programs are specified by means of pcDCTRSs, a general class of conditional term rewrite systems. We have proved a number of laws for our generated put functions, namely that we produce (partial) very well-behaved lenses in the terminology of [11]. Moreover, we have precisely characterized those view updates that are compatible with a given source (also called "in-place" updates) using narrowing [28, 19], an extension of rewriting to deal with logic variables. In some way, our approach combines ideas from three previous approaches: reversible rewriting [24], syntactic bidirectionalization [22], and semantic bidirectionalization [30], while it provides new insights by showing that narrowing can easily be used to identify the parts of a view that are updatable without modifying the complement.

As future work, we plan to develop a technique to deal with non-compatible view updates, along the lines presented in Section 4.3.

Acknowledgements

We thank the anonymous reviewers for their useful comments and suggestions to improve this paper.

References

1. Abramov, S.M., Glück, R.: The universal resolving algorithm and its correctness: inverse computation in a functional language. Sci. Comput. Program. **43**(2-3), 193–229 (2002)
2. Baader, F., Nipkow, T.: Term Rewriting and All That. Cambridge University Press (1998)
3. Bancilhon, F., Spyratos, N.: Update semantics of relational views. ACM Transactions on Database Systems **6**(4), 557–575 (1981)
4. Barbosa, D.M., Cretin, J., Foster, N., Greenberg, M., Pierce, B.C.: Matching lenses: alignment and view update. In: Hudak, P., Weirich, S. (eds.) Proc. of the 15th ACM SIGPLAN International Conference on Functional Programming (ICFP 2010). pp. 193–204. ACM (2010)
5. Bergstra, J.A., Klop, J.W.: Conditional Rewrite Rules: confluence and termination. Journal of Computer and System Sciences **32**, 323–362 (1986)
6. Bockmayr, A., Werner, A.: LSE Narrowing for Decreasing Conditional Term Rewrite Systems. In: Conditional Term Rewriting Systems, CTRS'94, Jerusalem. Springer LNCS 968 (1995)
7. Chin, W.N.: Towards an Automated Tupling Strategy. In: Proc. of Partial Evaluation and Semantics-Based Program Manipulation, Copenhagen, Denmark, June 1993. pp. 119–132. ACM, New York (1993)
8. Danos, V., Krivine, J.: Reversible communicating systems. In: Gardner, P., Yoshida, N. (eds.) Proc. of the 15th International Conference on Concurrency Theory (CONCUR 2004). Lecture Notes in Computer Science, vol. 3170, pp. 292–307. Springer (2004)
9. Diskin, Z., Xiong, Y., Czarnecki, K.: From State- to Delta-Based Bidirectional Model Transformations: the Asymmetric Case. Journal of Object Technology **10**, 6: 1–25 (2011)
10. Fischer, S., Hu, Z., Pacheco, H.: The essence of bidirectional programming. SCIENCE CHINA Information Sciences **58**(5), 1–21 (2015)
11. 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)
12. Foster, N., Matsuda, K., Voigtländer, J.: Three complementary approaches to bidirectional programming. In: Gibbons, J. (ed.) Generic and Indexed Programming - International Spring School, SSGIP 2010, Oxford, UK, March 22-26, 2010, Revised Lectures. Lecture Notes in Computer Science, vol. 7470, pp. 1–46. Springer (2012)
13. Ganzinger, H.: Order-sorted completion: The many-sorted way. Theor. Comput. Sci. **89**(1), 3–32 (1991)
14. Glück, R., Kawabe, M.: A method for automatic program inversion based on LR(0) parsing. Fundam. Inform. **66**(4), 367–395 (2005)
15. Glück, R., Kawabe, M.: A program inverter for a functional language with equality and constructors. In: Ohori, A. (ed.) Proceedings of the first Asian Symposium on Programming Languages and Systems (APLAS 2003). Lecture Notes in Computer Science, vol. 2895, pp. 246–264. Springer (2003)
16. Hidaka, S., Hu, Z., Inaba, K., Kato, H., Matsuda, K., Nakano, K.: Bidirectionalizing graph transformations. In: Hudak, P., Weirich, S. (eds.) Proceedings of the 15th ACM SIGPLAN International Conference on Functional programming (ICFP 2010). pp. 205–216. ACM (2010)

17. Hirokawa, N., Moser, G.: Automated Complexity Analysis Based on the Dependency Pair Method. In: Armando, A., Baumgartner, P., Dowek, G. (eds.) Proc. of IJCAR 2008. Lecture Notes in Computer Science, vol. 5195, pp. 364–379. Springer (2008)
18. Hu, Z., Schürr, A., Stevens, P., Terwilliger, J.F.: Bidirectional transformation "bx" (dagstuhl seminar 11031). Dagstuhl Reports **1**(1), 42–67 (2011), available from URL: `http://drops.dagstuhl.de/volltexte/2011/3144/`
19. Hullot, J.M.: Canonical forms and unification. In: Bibel, W., Kowalski, R.A. (eds.) Proceedings of the 5th International Conference on Automated Deduction. Lecture Notes in Computer Science, vol. 87, pp. 318–334. Springer (1980)
20. Landauer, R.: Irreversibility and heat generation in the computing process. IBM Journal of Research and Development **5**, 183–191 (1961)
21. Lloyd, J.W.: Foundations of Logic Programming. Springer-Verlag, Berlin (1987), second edition
22. Matsuda, K., Hu, Z., Nakano, K., Hamana, M., Takeichi, M.: Bidirectionalization transformation based on automatic derivation of view complement functions. In: Hinze, R., Ramsey, N. (eds.) Proc. of the 12th ACM SIGPLAN International Conference on Functional Programming, ICFP 2007. pp. 47–58. ACM (2007)
23. Nagashima, M., Sakai, M., Sakabe, T.: Determinization of conditional term rewriting systems. Theoretical Computer Science **464**, 72–89 (2012)
24. Nishida, N., Palacios, A., Vidal, G.: Reversible computation in term rewriting. J. Log. Algebr. Meth. Program. **94**, 128–149 (2018)
25. Nishida, N., Vidal, G.: Termination of narrowing via termination of rewriting. Appl. Algebra Eng. Commun. Comput. **21**(3), 177–225 (2010)
26. Phillips, I.C., Ulidowski, I.: Reversing algebraic process calculi. J. Log. Algebr. Program. **73**(1-2), 70–96 (2007)
27. Rounds, W.C.: Mappings and grammars on trees. Mathematical Systems Theory **4**(3), 257–287 (1970)
28. Slagle, J.R.: Automated theorem-proving for theories with simplifiers, commutativity and associativity. Journal of the ACM **21**(4), 622–642 (1974)
29. Terese: Term Rewriting Systems, Cambridge Tracts in Theoretical Computer Science, vol. 55. Cambridge University Press (2003)
30. Voigtländer, J.: Bidirectionalization for free! (pearl). In: Shao, Z., Pierce, B.C. (eds.) Proc. of the 36th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL 2009). pp. 165–176. ACM (2009)
31. Wadler, P.: Deforestation: Transforming programs to eliminate trees. Theoretical Computer Science **73**, 231–248 (1990)
32. Yokoyama, T., Axelsen, H., Glück, R.: Principles of a reversible programming language. In: Ramírez, A., Bilardi, G., Gschwind, M. (eds.) Proc. of the 5th Conference on Computing Frontiers. pp. 43–54. ACM (2008)