Bidirectionalization Transformation Based on Automatic Derivation of View Complement Functions

Kazutaka Matsuda[†] Zhenjiang Hu[†] Keisuke Nakano[†] Makoto Hamana[‡] Masato Takeichi[†]

[†]The University of Tokyo
kztk@ipl.t.u-tokyo.ac.jp
{hu,ksk,takeichi}@mist.i.u-tokyo.ac.jp

[‡]Gunma University
hamana@cs.gunma-u.ac.jp

Abstract

Bidirectional transformation is a pair of transformations: a *view function* and a *backward transformation*. A view function maps one data structure called source onto another called view. The corresponding backward transformation reflects changes in the view to the source. Its practically useful applications include replicated data synchronization, presentation-oriented editor development, tracing software development, and view updating in the database community. However, developing a bidirectional transformation is hard, because one has to give two mappings that satisfy the bidirectional properties for system consistency.

In this paper, we propose a new framework for bidirectionalization that can automatically generate a useful backward transformation from a view function while guaranteeing that the two transformations satisfy the bidirectional properties. Our framework is based on two known approaches to bidirectionalization, namely the constant complement approach from the database community and the combinator approach from the programming language community, but it has three new features: (1) unlike the constant complement approach, it can deal with transformations between algebraic data structures rather than just tables; (2) unlike the combinator approach, in which primitive bidirectional transformations have to be explicitly given, it can derive them automatically; (3) it generates a view update checker to validate updates on views, which has not been well addressed so far. The new framework has been implemented and the experimental results show that our framework has promise.

Categories and Subject Descriptors I.2.2 [*Artificial Intelligence*]: Automatic Programming—Program transformation, Program synthesis; D.1.1 [*Programming Techniques*]: Applicative (Functional) Programming; H.2.3 [*Database Management*]: Language—Data manipulation languages, Query languages

General Terms Languages, Design

Keywords Bidirectional Transformation, View Updating, Program Transformation, Automatic Program Generation, Program Inversion.

ICFP '07, 1–3, October 2007, Freiburg, Germany.

1. Introduction

There are many situations in which one data structure, called *source*, is transformed to another, called *view*, in such a way that changes on the view can be transformed back to those on the original data structure. This is called *bidirectional transformation*, and practical examples include synchronization of replicated data in different formats (Foster et al. 2005), presentation-oriented structured document development (Hu et al. 2004), interactive user interface design (Meertens 1998), coupled software transformation (Lämmel 2004), and the well-known *view updating* mechanism which has been intensively studied in the database community (Bancilhon and Spyratos 1981; Dayal and Bernstein 1982; Gottlob et al. 1988; Hegner 1990; Lechtenbörger and Vossen 2003).

To be concrete, suppose that we have a list of students and professors (the source), and we want to create a view that consists of all the students. This view can be defined by the following *view function*.

$$
\begin{aligned}
&\mathit{students} : \mathsf{Source} \to \mathsf{View}\\
&\mathit{students}\ [\,] = [\,]\\
&\mathit{students}\ (\mathsf{Student}\ \mathit{name}\ \mathit{grade}\ \mathit{major} : \mathit{ms})\\
&\quad = \mathsf{Student}\ \mathit{name}\ \mathit{grade}\ \mathit{major} : \mathit{students}\ \mathit{ms}\\
&\mathit{students}\ (\mathsf{Prof}\ \mathit{name}\ \mathit{position}\ \mathit{major} : \mathit{ms})\\
&\quad = \mathit{students}\ \mathit{ms}
\end{aligned}
$$

To develop a bidirectional transformation, in addition to the view function one needs to define another function, *a backward transformation function*, which is used for *view updating*, i.e., reflects changes on the view (such as modification of students' names) to the source. The following function $\mathit{students}_{\mathrm{B}}$[1] is a backward transformation that accepts a changed view and the original source and produces a new source.

$$
\begin{aligned}
&\mathit{students}_{\mathrm{B}} : (\mathsf{Source} \times \mathsf{View}) \to \mathsf{Source}\\
&\mathit{students}_{\mathrm{B}}\ ([\,], [\,]) = [\,]\\
&\mathit{students}_{\mathrm{B}}\ (\mathsf{Student}\ n\ g\ m : \mathit{ms}, \mathsf{Student}\ n'\ g'\ m' : \mathit{ss})\\
&\quad = \mathsf{Student}\ n'\ g'\ m' : \mathit{students}_{\mathrm{B}}\ (\mathit{ms}, \mathit{ss})\\
&\mathit{students}_{\mathrm{B}}\ (\mathsf{Prof}\ n\ p\ m : \mathit{ms}, \mathit{ss})\\
&\quad = \mathsf{Prof}\ n\ p\ m : \mathit{students}_{\mathrm{B}}\ (\mathit{ms}, \mathit{ss})
\end{aligned}
$$

However, there are several limitations in manually writing a pair of view function and backward transformation (such as $\mathit{students}$ and $\mathit{students}_{\mathrm{B}}$) to develop a bidirectional transformation. First, it is difficult to prove that the two functions satisfy the bidirectional property and form a bidirectional transformation (Section 2). Second, the consistency of the two functions is difficult to maintain. A small change in the view function may require a big change in the backward transformation function. For instance, suppose we want a view that contains only the names of the students who major in

[1] The subscript B stands for "backward".

Computer Science. While it is easy to give the following view function by composing another function with above the view function, it requires more thinking to write a backward transformation.

$$cs\_students = cs \circ students$$
$$cs\ [\,] = [\,]$$
$$cs\ (\mathsf{Student}\ name\ grade\ \mathsf{CS} : ss) = name : cs\ ss$$
$$cs\ (\_ : ss) = cs\ ss$$

Third, it is hard to automatically infer permitted changes in the view. Backward transformation functions, such as $students_{\mathrm{B}}$, are usually partial functions, disallowing some changes to the view. For example, $students_{\mathrm{B}}$ does not allow insertion of a new student to the view. When the source and the view are huge, it is better to have an inference system for validating view changes rather than to directly compute backward transformation until an error message appears.

Several methods for automatically deriving backward transformation functions from view definition functions have been proposed to overcome these three problems. In the database community, this issue, known as *view updating problem*, has been studied for a long time (Bancilhon and Spyratos 1981; Dayal and Bernstein 1982; Gottlob et al. 1988; Hegner 1990; Lechtenbörger and Vossen 2003).

One known approach (Bancilhon and Spyratos 1981) is to construct an injective function from the view function so that changes on the image of the injective function can be reflected back to its domain by inversion. To do this, lost information in the view generation is gathered as a *complement* for later backward transformation. Bancilhon and Spyratos proposed the concept of *view complement function* and the method of *view updating under constant complement*. Generally speaking, for a view function from a source to a view

$$f : S \to V$$

a view complement function of f is a function from the source to another view (called a complement view)

$$g : S \to V'$$

such that the tupled function

$$(f \vartriangle g) : S \to (V \times V')$$

is injective. This view complement function, g, provides the information that does not appear in the view generated by f. Then, for an original source s, the updated source s' corresponding to an updated view v can be calculated by

$$s' = (f \vartriangle g)^{-1}(v, g(s)).$$

Therefore, if a view complement function g can be derived and the inversion of $(f \vartriangle g)$ can be calculated, that means we know how to reflect changes on the view to the source. Since the derived backward transformation depends on g, we should choose an appropriate g. For example $g = id_S$ is the worst one because the derived backward transformation is defined only on the input (s, v) by $v = f(s)$, i.e., no updates are allowed. This approach has been applied to solving the view updating problem in the *relational database system* (Cosmadakis and Papadimitriou 1984; Laurent et al. 2001; Lechtenbörger and Vossen 2003). In fact, derivation of view complement functions and inversion calculation are simplified in the context of relational databases because views and sources are sets of flat tuples, and view functions are queries in a simple form being closed under composition. However, though tree-structured data, e.g. XML, is now widely used, how to apply the approach to view functions that can deal with *general algebraic data structures* such as trees is still an open problem. The challenge is to find a suitable form for these view functions such that they not only have enough descriptive power and view complement functions but are suitable for later inversion calculation.

Another approach, which has received increasing attention in recent years, is to design a set of general *combinators* (Foster et al. 2005; Hu et al. 2004; Meertens 1998) for constructing bigger bidirectional transformations by *composing* smaller ones. A set of primitive bidirectional transformations, each being defined by a pair of view function and backward transformation, is prepared, and a new bidirectional transformation is defined by assembling the primitive transformations with a fixed set of general combinators (Section 2.4.2). This approach has proved to be practically useful for domain-specific applications, because primitive bidirectional transformations for a specific application are easily determined, designed, and implemented. Moreover, this approach is general and can deal with trees other than relational tables provided that suitable primitive bidirectional transformations on trees are given. However, for an involved application or in a more general setting, a lot of primitive bidirectional transformations may need to be prepared, and it is still hard to verify whether a pair consisting of a view function and a backward transformation forms a (primitive) bidirectional transformation.

In this paper, we propose a new bidirectional transformation framework that combines the advantages of the above two approaches: automatic bidirectionalization and ability to deal with tree structures. We follow the combinator approach of keeping separate the design of primitive bidirectional transformations and the design of composition methods for gluing smaller transformations, but we borrow the idea of the view complement approach to obtain primitive bidirectional transformations that can manipulate arbitrary algebraic data structures.

The key to our framework is a suitable language for describing primitive view functions. It should be sufficiently powerful to specify various view functions over algebraic data structures, simple enough for derivation of view complement functions and inversion, and suitable for use as components to be composed with others. We choose a general first-order functional language and require view functions that are defined in the language to be *affine* (each variable is used at most once) and in the *treeless* form (no intermediate data structure is used in the definition). In fact, this class of functions has been considered elsewhere in the context of deforestation (fusion transformation) (Wadler 1990), where treeless functions are used to describe basic computation components.

In our framework, view complement functions can be automatically derived from view functions, and the derived view complement functions are suitable for tupling and inversion. Moreover, updatability of views can be represented by a regular tree language with which one can statically check whether changes in views are valid or not without really performing backward transformation.

We have implemented all the ideas in this paper using a bidirectionalizing system for automatically constructing backward transformation functions from view functions. The derived backward transformation function is *correct* in the sense that it forms a bidirectional transformation with the view function, *useful* in the sense that all the tracing information between the source and the view is recorded in such a way that any change in the view element that has a corresponding element in the source can be reflected to the source, *powerful* in the sense that it can deal with bidirectional transformation between arbitrary algebraic data structures such as lists and trees, and equipped with an *inference system* for validating changes in the view.

This paper is organized as follows. We start by explaining the basic concepts of bidirectional transformation in Section 2. Then, after presenting an overview of our system in Section 3, we define a language for view definition in Section 4, show how to derive view complement functions in Section 5, explain how to generate

backward transformation functions based on tupling and inversion in Section 6, and give an algorithm for updatability check in Section 7. We illustrate the whole bidirectionalization procedure with a concrete example in Section 8. Finally, we discuss related work in Section 9, and conclude the paper and highlight some future directions in Section 10.

For proofs (of all theorems), which are omitted in this paper, please see Matsuda et al. (2007).

2. Bidirectional Transformation

In this section, we briefly review notations, the basic concept of bidirectional transformation (i.e., view updating) (Bancilhon and Spyratos 1981; Dayal and Bernstein 1982; Gottlob et al. 1988; Hegner 1990; Lechtenbörger and Vossen 2003; Foster et al. 2005), and the technique of bidirectionalization based on derivation of *view complement functions* (Bancilhon and Spyratos 1981). These will serve as the basis of our approach.

2.1 Notations

Our notations, if not explained, follow Haskell[2], a functional programming language. For a partial function f, we write $f(x)\downarrow$ if $f(x)$ is defined, and write $f(x) = \bot$ otherwise. For a function $f : X \to Y$ and a function $g : X \to Z$, we define a tupled function $(f \vartriangle g) : X \to (Y \times Z)$ by

$$(f \vartriangle g)(x) = (f(x), g(x)).$$

For a partial function $f : X \to Y$ and a partial function $g : X \to Y$, we write $f \sqsubseteq g$ to denote $\forall x \in X, f(x)\downarrow \Rightarrow f(x) = g(x)$. Intuitively, $f \sqsubseteq g$ means that g is more widely defined than f.

2.2 View Function and Backward Transformation

Let V be the set of views and S be the set of sources. A total function $f : S \to V$ that constructs a view from a source is called *view function*. The following is an example of a view function

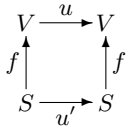
$$\begin{array}{lcl} \mathit{mapfst}(\mathsf{Nil}) & = & \mathsf{Nil} \\ \mathit{mapfst}(\mathsf{Cons}(\mathsf{Pair}(a,b),x)) & = & \mathsf{Cons}(a, \mathit{mapfst}(x)) \end{array}$$

that maps the source, a list of pairs, to the view, a list that contains all the first components of the pairs in the original list.

A function that translates an update on views to that on sources is called a backward transformation function. An update from x to x' is denoted as $x \rightarrowtail x'$; e.g., $\mathsf{Nil} \rightarrowtail \mathsf{Cons}(\mathsf{A}, \mathsf{Nil})$ represents the update on the view of mapfst from Nil to $\mathsf{Cons}(\mathsf{A}, \mathsf{Nil})$. Given a view function $f : S \to V$, a *backward transformation function* $\rho : (S \times V) \to S$ of f translates $f(s) \rightarrowtail v$, an update on views, to $s \rightarrowtail \rho(s, v)$, an update on sources, while satisfying the property:

$$\forall s \in S, \forall v \in V,\ \rho(s, v)\downarrow \Rightarrow f(\rho(s, v)) = v.$$

This property reads that the updated source produced by the backward transformation should not change the view with the view function. In other words, for a source element s and $v = f(s)$, let u be a view update $v \rightarrowtail v'$ and u' a translated source update $s \rightarrowtail \rho(s, v')$, then the following diagram commutes.



It might be easier to understand a backward transformation function $\rho : (S \times V) \to S$ as a mapping that accepts an original source and a changed view as input and produces a changed source as the result.

It is worth noting that backward transformation functions are partial: $\rho(s, v)\downarrow$ means that an update on views of $f(s) \rightarrowtail v$ is translated to an update on sources of $s \rightarrowtail \rho(s, v)$, and $\rho(s, v) = \bot$ means that it prohibits the view update $f(s) \rightarrowtail v$. Moreover, backward transformation functions are not unique, and different definitions give different translation policies. For instance, the following is a backward transformation function of the view function mapfst:

$$\rho(s, v) = \begin{cases} s & \text{if } v = \mathit{mapfst}(s) \\ \bot & \text{otherwise,} \end{cases}$$

which means that any change in the view is ignored and the source remains unchanged.

2.3 Bidirectional Properties

A backward transformation function ρ and a view function f should satisfy some bidirectional properties to guarantee consistency after bidirectional transformation is carried out. The following properties follow those in the *closed* view updating (Bancilhon and Spyratos 1981; Hegner 1990), where the source is completely hidden from the users when the view is updated.

Let $s \in S$ and $v, v' \in V$. A backward transformation function $\rho : (S \times V) \to S$ and a view function $f : S \to V$ should satisfy the following *bidirectional properties*.

ACCEPTABILITY
$\rho(s, f(s)) = s$

UNDOABILITY
$\rho(s, v)\downarrow \Rightarrow \rho(\rho(s, v), f(s)) = s$

COMPOSABILITY
$\rho(s, v)\downarrow \wedge \rho(\rho(s, v), v')\downarrow \Rightarrow \rho(\rho(s, v), v') = \rho(s, v')$

Acceptability means that if there is no change in views there should be no change in sources. Undoability means that all translated updates can be canceled by updates on views. Composability means that the update translation should preserve the compositional structure[3], and translated results do not depend on the update history.

2.4 Bidirectionalization

Bidirectionalization is a program transformation that derives a backward transformation function from a view function such that the two functions satisfy the bidirectional properties. It is very much related to the known *view update problem* in the database community, which discusses how to translate updates on views to updates on sources. We shall review two approaches on which our method is based.

2.4.1 Constant Complement Approach

Bancilhon and Spyratos (1981) proposed a general approach to bidirectionalization called *constant complement view updating*.

Definition 1 (View Complement Function). A function $g : S \to V'$ is said to be a *(view) complement function* to a view function $f : S \to V$, if the tupled function $f \vartriangle g : S \to (V \times V')$ is injective.

Intuitively, a view complement function of a view function provides information that is not visible in the view to a complement view such that information from both views can uniquely determine a source. For example, let add be a function defined by $\mathit{add}(x, y) = x + y$. Then, the function *fst* defined by $\mathit{fst}(x, y) = x$ is a view complement function of add. Note that the ranges of view function f and its complement function g can be different. In fact,

[2] Haskell 98 Report: `http://www.haskell.org/onlinereport/`

[3] Note that $u_1 = f(s) \rightarrowtail v$ is translated to $u'_1 = s \rightarrowtail \rho(s, v)$, $u_2 = v \rightarrowtail v'$ is translated to $u'_2 = \rho(s, v) \rightarrowtail \rho(\rho(s, v), v')$, and their composition $u = u_1 \circ u_2 = f(s) \rightarrowtail v'$ is translated to $u' = s \rightarrowtail \rho(s, v') = s \rightarrowtail \rho(\rho(s, v), v') = u'_1 \circ u'_2$.

the range of the view complement function is unimportant. This gives us freedom in derivation of view complement functions.

Finding a view complement function of a view function amounts to bidirectionalization, provided that inversion of can be calculated out (Bancilhon and Spyratos 1981). If a view complement function exists, a backward transformation function can be obtained by inversion. Let f be a view function and g be its view complement function. The function $\mathsf{upd}_{\langle f,g\rangle}$ defined by

$$\mathsf{upd}_{\langle f,g\rangle}(s,v) = (f \vartriangle g)^{-1}(v, g(s)) \qquad \text{(UPD)}$$

is a backward transformation function and satisfies bidirectional properties. The function $\mathsf{upd}_{\langle f,g\rangle}$ may be partial. For example, $\mathsf{upd}_{\langle mapfst,id\rangle}$ defines the same function as ρ in Section 2.2, which is defined only on the input (s,v) by $v = mapfst(s)$.

This general bidirectionalization framework has been used to bidirectionalize queries on *relational database system* (Cosmadakis and Papadimitriou 1984; Laurent et al. 2001; Lechtenbörger and Vossen 2003): derivation of view complement functions and inversion calculation is not difficult in this setting because views and sources are tuples and view definition functions are queries with a normal form being closed under composition. It is, however, unclear how to extend this approach to view functions that can deal with *general algebraic data structures* such as trees. In this paper, we intend to solve this problem.

2.4.2 Compositional Approach

The compositional approach (Foster et al. 2005; Hu et al. 2004; Meertens 1998) is to derive backward transformation functions based on the compositional structure of view functions. A view function is supposed to be defined by

- a primitive view function, or
- a composition of simpler view functions via several combinators.

The combinators for gluing view functions includes familiar constructs from functional programming languages:

- composition: $f \circ g$ defined by
$$(f \circ g)\ x = f(g\ x)$$
- mapping: $map\ f$ defined by
$$map\ f\ [x_1, \ldots, x_n] = [f\ x_1, \ldots, f\ x_n]$$
- product: $f \times g$ defined by
$$(f \times g)(x,y) = (f\ x, g\ y)$$
- conditional: if p then f else g defined by
$$(\text{if } p \text{ then } f \text{ else } g)\ x = \begin{cases} f\ x, & \text{if } p\ x \\ g\ x, & \text{otherwise} \end{cases}$$

It has been shown (Foster et al. 2005) that if one can prepare backward transformation functions for primitive view functions, one can get backward transformations for view functions that are constructed by primitive view functions and the above combinators for function gluing. The present paper shows how to automatically derive backward transformations for primitive view functions over arbitrary algebraic data structures.

2.5 Better Backward Transformation

Generally, there are many backward transformation functions for a given view function. Recall the constant complement approach to bidirectionalization and the view function add in Section 2.4.1. All functions below are view complement functions of add

$$fst(x,y) = x$$
$$sub(x,y) = x - y$$
$$id_{pair}(x,y) = (x,y)$$

and will lead to the following backward transformation functions based on the approach in Section 2.4.1.

$$\mathsf{upd}_{\langle add,fst\rangle}((x,y),v) = (x, v-y)$$
$$\mathsf{upd}_{\langle add,sub\rangle}((x,y),v) = ((v+(x-y))/2, (v-(x-y))/2)$$
$$\mathsf{upd}_{\langle add,id_{pair}\rangle}((x,y),v) = (x,y),\ \text{if } v = x+y$$

These backward transformation functions have different updatability: the first two allow any modification of the view, but the last one disallows arbitrary modification of the view because view v must be the same as $x+y$. Bancilhon and Spyratos (1981) introduce the following preorder, under which smaller view complement functions give more updatability.

Definition 2 (Collapsing Order). Let $f : S \to V,\ g : S \to V'$ be functions. The *collapsing order*, $\precsim$, is a preorder defined by

$$f \precsim g \iff \forall s, s' \in S,\ g(s) = g(s') \Rightarrow f(s) = f(s').$$

Order $f \precsim g$ means that, with respect to the results of mappings, f collapses input more than g. Hence, all elements in the input collapse into one in the result by the minimal functions, i.e., constant functions, and nothing collapses by the maximal functions, i.e., the injective functions. For the above examples, id_{pair} is greater than the others because it keeps the values of the input. The functions fst and sub are not comparable.

Note that a complement view keeps information that does not appear in the view, and that the backward transformation function derived from the view complement function should forbid any change in the information that the complement has kept. So, a smaller view complement function gives a better backward transformation function because it keeps less information. Formally, we have the following theorem (Bancilhon and Spyratos 1981).

Theorem 1. Let $f : S \to V$ be a view function, and $g_1 : S \to V'$ and $g_2 : S \to V''$ be two complement functions of f. If $g_1 \precsim g_2$, then

$$\mathsf{upd}_{\langle f,g_2\rangle} \sqsubseteq \mathsf{upd}_{\langle f,g_1\rangle}$$

holds.

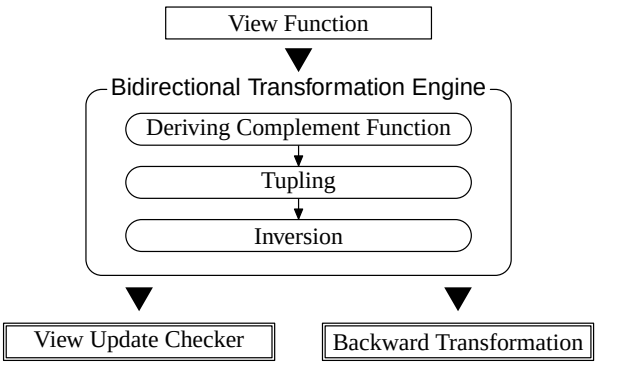


Figure 1. System Architecture

3. An Overview

Before we discuss the details of our new approach to bidirectionalization, we present an overview of our system, explaining its architecture and relation with the later sections and illustrating with

an example how it derives backward transformation functions from view functions that manipulate arbitrary algebraic data structures, including trees.

Figure 1 shows the architecture of our bidirectionalization system. The input to our system is a view function. The output consists of a backward transformation function and a checker that validates changes in the view. A change in the view is said to be *valid* if it can be reflected back to the source by the backward transformation function. The core part is the bidirectional transformation engine mapping from the input to the output.

3.1 View Function

Views are generated by application of view functions to sources. View functions are defined in a compositional way like

$$f = (f_1 \circ f_2) \times map\ f_3.$$

More precisely, a view function is a combination of primitive view functions and the gluing combinators, which were explained in Section 2.4.2.

Each primitive function is in the *affine, treeless* form defined by a constructor-based first-order functional language with pattern matching (Section 4). The patterns and constructors in the language make it easy to code primitive view functions from one algebraic data structure to another. As a simple example, consider generation of a view of a list from two lists by appending them together. This view function can be defined in our language as follows.

$$\begin{array}{ll} append(\mathsf{Nil}, y) & \hat{=}\ y \\ append(\mathsf{Cons}(a, x), y) & \hat{=}\ \mathsf{Cons}(a, append(x, y)) \end{array}$$

It decomposes the data by pattern matching and constructs new data by data constructors.

3.2 Bidirectional Transformation Engine

Since our view functions are compositional, our bidirectionalization basically consists of two parts:

- bidirectionalizing primitive view functions, and
- bidirectionalizing the gluing view functions combinators.

Given that bidirectionalization of combinators has been well studied (Foster et al. 2005; Hu et al. 2004), we will focus on bidirectionalization of primitive view functions, though the whole system should combine the two.

3.2.1 Deriving View Complement Functions

Our system starts by automatically deriving a *small* (with respect to the collapsing order in Definition 2) view complement function for a given view function so that tupling the two functions gives an injective function (Section 5). For example, a view complement function automatically derived by our system for $append$ is as follows.

$$\begin{array}{ll} append^c(\mathsf{Nil}, y) & \hat{=}\ \mathsf{B}_1 \\ append^c(\mathsf{Cons}(a, x), y) & \hat{=}\ \mathsf{B}_2(append^c(x, y)) \end{array}$$

In this definition, B_1 and B_2 are data constructors automatically generated by the system. A close look at the definition reveals that the derived view complement function actually computes the length of the first argument. One can easily verify that although $append$ is non-injective, tupling $append$ and $append^c$, $append \vartriangle append^c$, is injective.

3.2.2 Deriving Backward Transformation Functions by Tupling and Inversion

After obtaining the view complement function, our system generates a backward transformation function based on two program transformations, tupling and inversion, based on the constant complement approach to bidirectionalization (Section 6).

For the example $append$, our system first automatically derives the following definition for the tupled function $append^{\vartriangle}$, $append \vartriangle append^c$.

$$\begin{array}{ll} append^{\vartriangle}(\mathsf{Nil}, y) & \hat{=}\ (y, \mathsf{B}_1) \\ append^{\vartriangle}(\mathsf{Cons}(a, x), y) & \hat{=}\ (\mathsf{Cons}(a, s), \mathsf{B}_2(t)) \\ \quad \textbf{where}\ (s, t) \hat{=} append^{\vartriangle}(x, y) & \end{array}$$

Then, it derives the following inverse of the tupled function.

$$\begin{array}{ll} (append^{\vartriangle})^{-1}(y, \mathsf{B}_1) & \hat{=}\ (\mathsf{Nil}, y) \\ (append^{\vartriangle})^{-1}(\mathsf{Cons}(a, s), \mathsf{B}_2(t)) & \hat{=}\ (\mathsf{Cons}(a, x), y) \\ \quad \textbf{where}\ (x, y) \hat{=} (append^{\vartriangle})^{-1}(s, t) & \end{array}$$

Finally, it applies Equation (UPD) in Section 2.4.1 and produces the following backward transformation function.

$$\mathsf{upd}_{\langle append, append^c \rangle}(s, v) \hat{=} (append^{\vartriangle})^{-1}(v, append^c(s))$$

To see what the derived backward transformation function actually is, let us rename $\mathsf{upd}_{\langle append, append^c \rangle}$ to $append_{\mathrm{B}}$. Applying the fusion transformation (Wadler 1990) can yield the following definition.

$$\begin{array}{ll} append_{\mathrm{B}}((\mathsf{Nil}, y), v) & \hat{=}\ (\mathsf{Nil}, v) \\ append_{\mathrm{B}}((\mathsf{Cons}(a, x), y), \mathsf{Cons}(b, v)) & \hat{=}\ (\mathsf{Cons}(b, s), t) \\ \quad \textbf{where}\ (s, t) \hat{=} append_{\mathrm{B}}((x, y), v) & \end{array}$$

That is, $append_{\mathrm{B}}$ is such a function, accepting the original source (x, y) and a new view v and returning a new source (x', y'), where x' is the first n elements of v and y' is the rest. Here, n is the length of x.

3.2.3 Generating View Update Checker

As seen in the above example, a derived backward transformation function may be partial: function $append_{\mathrm{B}}$ is defined only if the length of the updated view is not less than the length of the first list in the original source. Therefore, the backward transformation will fail if an updated view does not fall in its domain.

Our system automatically generates from a given view function, together with the original source, a view update checker, represented by a tree automaton, which can check whether an update on the view is valid or not. Section 7 explains in detail how to generate view update checkers, including a generated automaton for the example $append$ in this section.

4. View Definition Language

In this section, we introduce our language, VDL, for defining view functions. It is a first-order functional programming language that is similar to Wadler's language for defining basic functions for fusion (Wadler 1990).

4.1 The Language VDL

The syntax and semantics[4] of the language is given in Figure 2. A program of our language consists of a set of function definitions, and each function is defined by several *rules* of the form

$$f(p_1, \ldots, p_n) \hat{=} e.$$

To simplify our presentation, we assume that there is no overlap among rules of the same function, i.e., no two patterns in the left-hand side overlap.

There are two important syntactic restrictions on each rule declaration.

[4] Note that VDL has the call-by-value semantics, where values are expressions that consists of only constructor symbols in $\mathcal{C}$.

Syntax:

$$
\begin{array}{llll}
\mathit{rule} & ::= & f(p_1, \ldots, p_n) \mathrel{\hat{=}} e & \\
p & ::= & \mathsf{C}(p_1, \ldots, p_n) & \text{constructor pattern} \\
 & \mid & x & \text{variable pattern} \\
e & ::= & \mathsf{C}(e_1, \ldots, e_n) & \text{constructor application} \\
 & \mid & f(x_1, \ldots, x_n) & \text{function call} \\
 & \mid & x & \text{variable}
\end{array}
$$

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

Operational Semantics:

$$\text{(Con)}\frac{e_1 \Downarrow v_1 \quad \cdots \quad e_n \Downarrow v_n}{\mathsf{C}(e_1, \ldots, e_n) \Downarrow \mathsf{C}(v_1, \ldots, v_n)}$$

$$\text{(Fun)}\frac{f(p_1, \ldots, p_n) \mathrel{\hat{=}} e \in \mathcal{R} \quad \exists \theta, f(p_1\theta, \ldots, p_n\theta) = f(v_1, \ldots, v_n) \quad e\theta \Downarrow u}{f(v_1, \ldots, v_n) \Downarrow u}$$

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

Figure 2. View Definition Language

- The expression, e, of a rule is in a *treeless* form (Wadler 1990), i.e., a function call, which may appear inside a constructor application but never appears inside another function call. It can be seen in Figure 2 that each argument to a function call is a variable instead of an expression. This restriction ensures no intermediate data structure in e.
- Variable occurrences in a rule are *affine*, i.e., every variable in the left-hand side of a rule occurs at most once in the corresponding right-hand side. This restriction ensures that there is no duplication of data.

These two syntactic restrictions play an important role in our automatic bidirectionalization framework, simplifying the generation of a view complement function from a view function written in VDL.

Though restricted, this language is sufficiently powerful to describe many interesting view functions. It is not difficult to see that the view functions we have seen so far, such as *students*, *mapfst*, and *append*, can be coded in VDL with slight syntactic modification. In the following, we give more examples of view functions in VDL.

Example 1 (Identity View Function). The simplest view function is the identity function, which creates a view that is the same as its source. It can be defined in VDL as follows.

$$\mathit{id}(x) \mathrel{\hat{=}} x$$

Example 2 (Projection View Functions). The projection view functions are useful for selecting a component from the source. They can be defined in VDL as follows.

$$\begin{array}{ll} \mathit{fst}(x, y) & \mathrel{\hat{=}} x \\ \mathit{snd}(x, y) & \mathrel{\hat{=}} y \end{array}$$

Example 3 (Constant View Functions). A constant view function is useful for creating a view that is independent of its source. An example of the constant view function is defined in VDL as follows.

$$\mathit{nil}(x) \mathrel{\hat{=}} \mathsf{Nil}$$

Example 4 (Recursive View Functions on Natural Numbers). Many view functions are defined recursively by traversing over data structures. For example, the view function for addition of two natural numbers is defined by

$$\begin{array}{ll} \mathit{add}(\mathsf{Z}, y) & \mathrel{\hat{=}} y \\ \mathit{add}(\mathsf{S}(x), y) & \mathrel{\hat{=}} \mathsf{S}(\mathit{add}(x, y)). \end{array}$$

As in Haskell, we use a symbol starting with an uppercase letter to denote a constructor and a symbol starting with a lowercase letter to denote a function or a variable. Function add is defined by traversing over one data structure in the source, while the following function, max, for computing the maximum of two natural numbers is defined by simultaneously traversing over two data structures in the source.

$$\begin{array}{ll} \mathit{max}(\mathsf{Z}, y) & \mathrel{\hat{=}} y \\ \mathit{max}(\mathsf{S}(x), \mathsf{Z}) & \mathrel{\hat{=}} \mathsf{S}(x) \\ \mathit{max}(\mathsf{S}(x), \mathsf{S}(y)) & \mathrel{\hat{=}} \mathsf{S}(\mathit{max}(x, y)) \end{array}$$

Example 5 (Recursive View Functions on Lists and Trees). Our language can be used to define view functions on various data structures such as lists and trees. As a view function on lists, the function *zip* for zipping two lists is defined below.

$$\begin{array}{ll} \mathit{zip}(\mathsf{Nil}, y) & \mathrel{\hat{=}} \mathsf{Nil} \\ \mathit{zip}(\mathsf{Cons}(a, x), \mathsf{Nil}) & \mathrel{\hat{=}} \mathsf{Nil} \\ \mathit{zip}(\mathsf{Cons}(a, x), \mathsf{Cons}(b, y)) & \mathrel{\hat{=}} \mathsf{Cons}(\mathsf{Pair}(a, b), \mathit{zip}(x, y)) \end{array}$$

As a view function on trees, the function for flipping a binary tree is defined below.

$$\begin{array}{ll} \mathit{flip}(\mathsf{Leaf}) & \mathrel{\hat{=}} \mathsf{Leaf} \\ \mathit{flip}(\mathsf{Node}(n, l, r)) & \mathrel{\hat{=}} \mathsf{Node}(n, \mathit{flip}(r), \mathit{flip}(l)) \end{array}$$

4.2 Notations for Manipulating Programs in VDL

In the rest of this paper, we will discuss several program transformations and prove important properties for them. To do this, we give a more formal definition of our programs in VDL, and prepare some notations and functions for later program manipulation.

Formally, a program P in our language VDL is a 4-tuple $(\mathcal{R}, \mathcal{F}, \mathcal{C}, \mathcal{X})$ where

- $\mathcal{R}$ is a set of rules (see Figure 2),
- $\mathcal{F}$ is a set of *function symbols* with associated arities,
- $\mathcal{C}$ is a set of *constructor symbols* with associated arities, and
- $\mathcal{X}$ is a set of *variables*

such that all sets are pairwise disjoint. We call an expression generated only by constructor symbols in $\mathcal{C}$ a *value* or a *tree value* and use $\mathcal{T}_\mathcal{C}$ to denote the set of all values.

A *substitution* is a mapping $\theta : \mathcal{X} \to \mathcal{T}_\mathcal{C}$ that assigns to a variable a value. We denote by $e\theta$ an expression obtained by replacing each variable x in e with a tree $\theta(x)$.

As discussed before, we do not allow rule overlapping in $\mathcal{R}$. Formally, $\mathcal{R}$ is *non-overlapping* if for any two distinct rules

$$\begin{array}{l} f(p_1, \ldots, p_n) \mathrel{\hat{=}} e \\ f(p'_1, \ldots, p'_n) \mathrel{\hat{=}} e' \end{array}$$

there is no substitution θ satisfying $(p_1, \ldots, p_n)\theta = (p'_1, \ldots, p'_n)\theta$.

We sometimes use vector notations $\overrightarrow{e}$ to denote sequence $e_1, \ldots, e_n$ when the length of sequence n is not concerned. For example, a rule $f(p_1, \ldots, p_n) \mathrel{\hat{=}} e$ is denoted as $f(\overrightarrow{p}) \mathrel{\hat{=}} e$.

For a rule r, we write $\mathsf{Vars}(r)$ to denote the set of all variables occurring in r, $\mathsf{UsedVars}(r)$ the set of all variables occurring in the right-hand side of r, and $\mathsf{LostVars}(r) = \mathsf{Vars}(r) \setminus \mathsf{UsedVars}(r)$.

To prove the properties of programs, we sometimes need to distinguish function symbols from their meanings. We denote the semantics of f by $[\![f]\!]_P$. Under the operational semantics of VDL, shown in Figure 2, a program P yields a partial function $[\![f]\!]_P$:

$(\mathcal{T}_\mathcal{C} \times \cdots \times \mathcal{T}_\mathcal{C}) \to \mathcal{T}_\mathcal{C}$ for each function symbol $f \in \mathcal{F}$:

$$[\![f]\!]_P(v_1, \ldots, v_n) = \begin{cases} v & \text{if } f(v_1, \ldots, v_n) \Downarrow v, \\ \bot & \text{otherwise.} \end{cases}$$

Note that when it is clear from the context, $[\![f]\!]_P$ is sometimes simply written as $[\![f]\!]$ or even f.

We add two semantic restrictions to VDL to avoid pathological situations in the proofs of the properties of programs written in VDL. First, a set of constructors $\mathcal{C}$ contains at least two constructors and one is zero-arity. Second, VDL does not contain functions that are undefined everywhere. For example, the following function f is undefined everywhere.

$$f(x) \mathrel{\hat{=}} f(x)$$

5. Deriving View Complement Functions

We shall develop algorithms for derivation of view complement functions from view functions so that tupling them gives an injective function. Compared to the algorithms in Cosmadakis and Papadimitriou (1984), Laurent et al. (2001) and Lechtenbörger and Vossen (2003), our algorithms are capable of dealing with functions on tree data structures. We start with a direct algorithm, and then improve it with a minimizing procedure with injectivity and range analysis.

5.1 A Direct Solution

For a given view function $f : S \to V$, to derive its complement function $g : S \to V'$, we should be clear about where the non-injectivity of f comes from. Recall that a complement function g of f is a function that makes the tupled function $(f \vartriangle g) : S \to (V \times V')$ *injective*. So, if f is injective, its complement function g can be an arbitrary function from S to V' (of course, updatability of the backward transformation depends on which g to choose). If f is non-injective, g should return different values for any distinct arguments $x, y \in S$ such that $f(x) = f(y)$.

Syntactically, there are basically two possible cases for a view function to be non-injective:

1. Some variables on the left-hand side of a rule disappear in the corresponding right-hand side. For example, function $fst(x, y) \mathrel{\hat{=}} x$ is non-injective.
2. The ranges of two right-hand sides of a view function overlap. For example, the following f is obviously non-injective:

$$f(\mathsf{A}) \mathrel{\hat{=}} \mathsf{A} \qquad f(\mathsf{B}) \mathrel{\hat{=}} \mathsf{A}.$$

Using this observation, we give an algorithm to derive a complement function of a view function.

Below, we use the context notation. A *context* K is a tree value containing special holes $\Box_1, \ldots, \Box_n$ and we denote by $K[e_1, \ldots, e_n]$ an expression obtained by replacing $\Box_i$ with e_i for each i in $\{1, \ldots, n\}$. Any expression in treeless form can always be separated as a context, function calls and variables as $K[f_1(\overrightarrow{x_1}), \ldots, f_n(\overrightarrow{x_n}), \overrightarrow{x}]$.

Algorithm 1 (Derivation of Complement Function: $\mathcal{ALCG}_\mathrm{c}$).

Input: A program $P = (\mathcal{R}, \mathcal{F}, \mathcal{C}, \mathcal{X})$ for view functions.

Output: A program P^c for view complement functions.

Procedure:

1. For each rule $r \in \mathcal{R}$

$$r = f(\overrightarrow{p}) \mathrel{\hat{=}} K[f_1(\overrightarrow{x_1}), \ldots, f_n(\overrightarrow{x_n}), \overrightarrow{x}]$$

construct a rule

$$r^c = f^c(\overrightarrow{p}) \mathrel{\hat{=}} \mathsf{B}_r(f_1^c(\overrightarrow{x_1}), \ldots, f_n^c(\overrightarrow{x_n}), \overrightarrow{y})$$

where $\{\overrightarrow{y}\} = \mathsf{LostVars}(r)$ and B_r is a fresh constructor, and ${f^c}, {f_1}^c, \ldots, {f_n}^c \notin \mathcal{F}$ are function symbols corresponding to $f, f_1, \ldots, f_n$ respectively.

2. Create a program as follows.

$$P^c = (\{r^c \mid r \in \mathcal{R}\}, \{f^c \mid f \in \mathcal{F}\}, \{\mathsf{B}_r \mid r \in \mathcal{R}\} \cup \mathcal{C}, \mathcal{X}).\Box$$

Theorem 2 (Soundness of $\mathcal{ALCG}_\mathrm{c}$). Let $P = (\mathcal{R}, \mathcal{F}, \mathcal{C}, \mathcal{X})$ be a program and $P^c = (\mathcal{R}^c, \mathcal{F}^c, \mathcal{C}^c, \mathcal{X}^c)$ the derived program by $\mathcal{ALCG}_\mathrm{c}$. Then, for every function symbol $f \in \mathcal{F}$, $[\![f^c]\!]$ is a complement function of $[\![f]\!]$.

Note that by $\mathcal{ALCG}_\mathrm{c}$, a function is defined if and only if its complement function is defined, i.e., $f(\overrightarrow{v}) \downarrow$ if and only if $f^c(\overrightarrow{v}) \downarrow$.

Example 6. Consider function fst defined by

$$fst(x, y) \mathrel{\hat{=}} x.$$

In this definition, the second argument, y, is discarded. Algorithm $\mathcal{ALCG}_\mathrm{c}$ derives the rule

$$fst^c(x, y) \mathrel{\hat{=}} \mathsf{B}_1(y).$$

Here, B_1 is the newly-introduced constructor for the first rule. Later, we use the constructor B_i for the ith rule. That is, function fst^c "complements" lost value "y" in the definition of fst.

Example 7. Consider the function add defined in Example 4. In this definition, all variables are preserved from the left-hand side to the right-hand side of each rule. Algorithm $\mathcal{ALCG}_\mathrm{c}$ derives the following two rules.

$$\begin{array}{ll} add^c(\mathsf{Z}, y) & \mathrel{\hat{=}} \mathsf{B}_1 \\ add^c(\mathsf{S}(x), y) & \mathrel{\hat{=}} \mathsf{B}_2(add^c(x, y)) \end{array}$$

This function add^c actually returns the information of the first argument.

Example 8. Consider the function max defined in Example 4. Algorithm $\mathcal{ALCG}_\mathrm{c}$ derives the following three rules.

$$\begin{array}{ll} max^c(\mathsf{Z}, y) & \mathrel{\hat{=}} \mathsf{B}_1 \\ max^c(\mathsf{S}(x), \mathsf{Z}) & \mathrel{\hat{=}} \mathsf{B}_2 \\ max^c(\mathsf{S}(x), \mathsf{S}(y)) & \mathrel{\hat{=}} \mathsf{B}_3(max^c(x, y)) \end{array}$$

This complement function is basically equivalent, with respect to the collapsing order, to the following function, $minle$, which returns the minimum of arguments and a boolean value indicating whether the first argument is "less than or equal" to the second.

$$minle(x, y) = \begin{cases} (x, 1) & \text{if } x \le y \\ (y, 0) & \text{if } x > y \end{cases}$$

In general, there can be infinitely many complement functions for a given view function. Ideally, we want to obtain a complement function that is minimal with respect to the collapsing order. The function, fst^c, derived by $\mathcal{ALCG}_\mathrm{c}$ is a minimal complement of fst, but a complement function derived by $\mathcal{ALCG}_\mathrm{c}$ is not always a minimal one. Next we will consider how to obtain smaller complement functions.

5.2 Making it Smaller

Algorithm $\mathcal{ALCG}_\mathrm{c}$ does not always return a minimal complement function. For example, consider the following function not.

$$not(\mathsf{True}) \mathrel{\hat{=}} \mathsf{False} \quad not(\mathsf{False}) \mathrel{\hat{=}} \mathsf{True}$$

Since the function not is injective, a minimal complement of this function can be any constant function. But Algorithm $\mathcal{ALCG}_\mathrm{c}$ derives the rules

$$not^c(\mathsf{True}) \mathrel{\hat{=}} \mathsf{B}_1 \quad not^c(\mathsf{False}) \mathrel{\hat{=}} \mathsf{B}_2$$

which is obviously not a constant function.

To derive smaller complement functions, we improve Algorithm $\mathcal{ALG}_{\mathrm{c}}$ by *analyzing injectivity* of function, and *calculating ranges* of the right-hand-side expressions. These two kinds of analysis are useful to minimize complement functions for the following reasons:

1. If the input function is recognized to be injective, we should return a constant function. This requires determination of the injectivity of a function. Fortunately, this is decidable in VDL. In the next subsection, we present an algorithm to determine injectivity.
2. Let e be an expression that may contain free variables. By *the range of an expression* e, we mean the set of evaluated values of all possible ground instances of e, i.e., the set defined by
$$\mathsf{Range}(e) = \{v \mid \exists\theta : \mathcal{X} \to \mathcal{T}_{\mathcal{C}},\ e\theta \Downarrow v\}.$$
Suppose that we have two rules
$$f(\mathsf{C}_1(x)) \ \hat{=}\ e_1 \quad f(\mathsf{C}_2(x)) \ \hat{=}\ e_2.$$
If the ranges of e_1 and e_2 do not overlap, $f^c(\mathsf{C}_1(x))$ and $f^c(\mathsf{C}_2(x))$ can be safely collapsed to the same value for some input, and hence $[\![f^c]\!]$ becomes smaller. In Section 5.2.2 we present an algorithm to calculate the range of an expression by using a tree automaton.
3. In addition to the above, the range analysis is helpful to remove unnecessary unary constructors. For example, let f be a view function and f^c be its complement function.
$$f(\mathsf{C}_1(x)) \mathrel{\hat{=}} \mathsf{D}_1(f(x)) \quad f(\mathsf{C}_2) \mathrel{\hat{=}} \mathsf{D}_2 \quad f(\mathsf{C}_3) \mathrel{\hat{=}} \mathsf{D}_2$$
$$f^c(\mathsf{C}_1(x)) \mathrel{\hat{=}} \mathsf{B}_1(f^c(x)) \quad f^c(\mathsf{C}_2) \mathrel{\hat{=}} \mathsf{B}_2 \quad f^c(\mathsf{C}_3) \mathrel{\hat{=}} \mathsf{B}_3$$
The unary constructor, B_1, can be removed if the ranges of $\mathsf{D}_1(f(x))$ and D_2 do not overlap. This is because the ranges of right-hand-side expressions of tupled function $(f \vartriangle f^c)$ do not overlap if for any two rules r_1, r_2 of f either the ranges of right-hand-side expressions of r_1, r_2 or ${r_1}^c, {r_2}^c$ do not overlap. The obtained complement function defined as
$$f^c(\mathsf{C}_1(x)) \mathrel{\hat{=}} f^c(x) \quad f^c(\mathsf{C}_2) \mathrel{\hat{=}} \mathsf{B}_2 \quad f^c(\mathsf{C}_3) \mathrel{\hat{=}} \mathsf{B}_3$$
is smaller than the original complement function.

5.2.1 Injectivity Analysis

We present an algorithm that determines the injectivity of a function. The algorithm consists of three major steps. In every step, the algorithm marks functions if they are non-injective and otherwise proceeds to other steps. All functions unmarked at the end are injective.

Algorithm 2 (Injectivity Checking: $\mathcal{ALG}_{\mathrm{i}}$).

Input: A program $P = (\mathcal{R}, \mathcal{F}, \mathcal{C}, \mathcal{X})$ for view functions.

Output: For each function $f \in \mathcal{F}$, "f is injective" or "f is non-injective".

Procedure:

1. Mark those functions that have a rule discarding variables.
2. Mark those functions whose ranges of right-hand sides of two distinct rules overlap.
3. Repeat
   Mark those functions that call marked functions.
   Until no marking can be done.
4. Return "f is non-injective" if f is marked, otherwise return "f is injective". □

Theorem 3 (Soundness and Completeness of $\mathcal{ALG}_{\mathrm{i}}$). For every function symbol $f \in \mathcal{F}$, $\mathcal{ALG}_{\mathrm{i}}$ returns "f is injective" if and only if $[\![f]\!]$ is an injective function.

5.2.2 Range Analysis

For every expression occurring in a program, we can construct an automaton that accepts exactly the trees in the range of the expression. Our idea was based on the existing result that the image of a linear tree transducer is a regular tree language (Engelfriet 1975).

Let $P = (\mathcal{R}, \mathcal{F}, \mathcal{C}, \mathcal{X})$ be a program and $\mathcal{E}$ be a set of expressions occurring in $\mathcal{R}$. For an expression $e \in \mathcal{E}$, we construct a nondeterministic (bottom-up) finite tree automaton A_e over $\mathcal{C}$. This automaton is a tuple $(Q, \mathcal{C}, \{q_e\}, \Delta)$ with a set of states Q, a set of constructors $\mathcal{C}$, the unique final state q_e, and a set of transition rules Δ where

- $Q = \{q_f \mid f \in \mathcal{F}\} \cup \{q_{e'} \mid e' \in \mathcal{E}\} \cup \{q_*\}$
- Δ consists of
  - $q_* \to q_{e'}$ with $e' = x \in \mathcal{E}$ and $x \in \mathcal{X}$,
  - $q_f \to q_{e'}$ with $e' = f(\dots) \in \mathcal{E}$ and $f \in \mathcal{F}$,
  - $\mathsf{C}(q_{e_1}, \dots, q_{e_n}) \to q_{e'}$ with $e' = \mathsf{C}(e_1, \dots, e_n) \in \mathcal{E}$,
  - $q_{e'} \to q_f$ for $f(\dots) \mathrel{\hat{=}} e' \in \mathcal{R}$ and
  - $\mathsf{C}(q_*, \dots, q_*) \to q_*$ with $\mathsf{C} \in \mathcal{C}$.

The following lemma states that automaton A_e exactly accepts the trees in the range of e.

Lemma 1 (The Range of Expressions). Let $P = (\mathcal{R}, \mathcal{F}, \mathcal{C}, \mathcal{X})$ be a program. For each expression e occurring in P, a tree t is in the range of e if and only if the tree automaton, A_e, accepts t.

The ranges of two expressions e and e' overlap if and only if the language accepted by the intersection of two tree automata A_e and $A_{e'}$ is not empty. Since finite tree automata are closed under intersection and the emptiness of a finite tree automaton is decidable (Comon et al. 1997), we have the following corollary.

Corollary 1. For a program $P = (\mathcal{R}, \mathcal{F}, \mathcal{C}, \mathcal{X})$, whether the ranges of two expressions in $\mathcal{R}$ overlap or not is decidable.

5.2.3 Deriving Smaller Complement Functions

With injectivity and range analysis, we can improve Algorithm $\mathcal{ALG}_{\mathrm{c}}$ and derive smaller complement functions. We change three parts in the original algorithm. First, we remove $f^c(\dots)$ for every injective function f from arguments of B in $\mathcal{ALG}_{\mathrm{c}}$ in the construction of the complement, because a complement function of any injective function is a constant function and can be ignored. Second, we use the same constructor for those rules of f^c when the ranges of the right-hand-side expressions of these rules do not overlap. Third, we remove a unary constructor from a rule for f^c, if the range of the right-hand-side expression of the corresponding rule of f does not overlap with the ranges of other rules of f.

As a preprocessing step, we calculate a partition of $\mathcal{R} = \mathcal{R}_1 \uplus \dots \uplus \mathcal{R}_k$ such that for each rule subset $\mathcal{R}_i$ the following hold.

- For all $r, r' \in \mathcal{R}_i$, r and r' define the same function.
- For all $r, r' \in \mathcal{R}_i$, the sum of non-injective functions and lost variables in both rules are the same.
- For all $r, r' \in \mathcal{R}_i$, the ranges of the right-hand-side expressions of r and r' do not overlap.

Algorithm 3 (Improvement of $\mathcal{ALG}_{\mathrm{c}}$: $\mathcal{ALG}_{\mathrm{sc}}$).

Input: A program $P = (\mathcal{R}, \mathcal{F}, \mathcal{C}, \mathcal{X})$ for view functions and a partition of $\mathcal{R} = \mathcal{R}_1 \uplus \dots \uplus \mathcal{R}_k$.

Output: A program P^c for view complement functions.

Procedure:
For each rule r for defining $f \in \mathcal{F}$:

$$r = f(\overrightarrow{p}) \mathrel{\hat{=}} K[f_1(\overrightarrow{x_1}), \ldots, f_n(\overrightarrow{x_n}), \overrightarrow{x}] \in \mathcal{R}_j$$

do the following:

1. Construct a rule

$$r^c_{\mathsf{pre}} = f^c(\overrightarrow{p}) \mathrel{\hat{=}} \mathsf{B}_k(f'^c_1(\overrightarrow{x'_1}), \ldots, f'^c_m(\overrightarrow{x'_m}), \overrightarrow{y})$$

   where the function calls $f'_1(\overrightarrow{x'_1}), \ldots, f'_m(\overrightarrow{x'_m})$ are obtained from $f_1(\overrightarrow{x_1}), \ldots, f_n(\overrightarrow{x_n})$ all injective function calls removed, and $\{\overrightarrow{y}\} = \mathsf{LostVars}(r)$.
2. If r^c_{pre} is in the form of

$$f^c(\overrightarrow{p}) \mathrel{\hat{=}} \mathsf{B}_j(f'^c(\overrightarrow{x'}))$$

   and the right-hand-side expression of r does not overlap with the right-hand-side expression of any other rule r' for $f \in \mathcal{F}$, construct a rule

$$r^c = f^c(\overrightarrow{p}) \mathrel{\hat{=}} f'^c(\overrightarrow{x'}),$$

   otherwise, construct a rule

$$r^c = r^c_{\mathsf{pre}}.$$

3. Create a program as follows.

$$P^c = (\{r^c \mid r \in \mathcal{R}\}, \{f^c \mid f \in \mathcal{F}\}, \{\mathsf{B}_j \mid \mathcal{R}_j\}, \mathcal{X}) \quad \square$$

Theorem 4 (Soundness of $\mathcal{ALCG}_{\mathrm{sc}}$). Let $P = (\mathcal{R}, \mathcal{F}, \mathcal{C}, \mathcal{X})$ be a program and $P^c = (\mathcal{R}^c, \mathcal{F}^c, \mathcal{C}^c, \mathcal{X}^c)$ the derived program by $\mathcal{ALCG}_{\mathrm{sc}}$. Then, for every function symbol $f \in \mathcal{F}$, $[\![f^c]\!]$ is a complement function of $[\![f]\!]$.

Example 9 (Role of Rule Partition). Consider the function, f, defined by

$$\begin{array}{l} r_1 = f(\mathsf{A}_1) \mathrel{\hat{=}} \mathsf{C}_1 \\ r_2 = f(\mathsf{A}_2) \mathrel{\hat{=}} \mathsf{C}_2 \\ r_3 = f(\mathsf{A}_3) \mathrel{\hat{=}} \mathsf{C}_1 \end{array}$$

and suppose that $\mathcal{R} = \{r_1, r_2\} \uplus \{r_3\}$. Then, Algorithm $\mathcal{ALCG}_{\mathrm{sc}}$ returns the following complement function.

$$\begin{array}{l} f^c(\mathsf{A}_1) \mathrel{\hat{=}} \mathsf{B}_1 \\ f^c(\mathsf{A}_2) \mathrel{\hat{=}} \mathsf{B}_1 \\ f^c(\mathsf{A}_3) \mathrel{\hat{=}} \mathsf{B}_2 \end{array}$$

However, if $\mathcal{R} = \{r_1\} \uplus \{r_2, r_3\}$, $\mathcal{ALCG}_{\mathrm{sc}}$ will return another complement function.

$$\begin{array}{l} f^c(\mathsf{A}_1) \mathrel{\hat{=}} \mathsf{B}_1 \\ f^c(\mathsf{A}_2) \mathrel{\hat{=}} \mathsf{B}_2 \\ f^c(\mathsf{A}_3) \mathrel{\hat{=}} \mathsf{B}_2 \end{array}$$

So different rule partitions can lead to different complement functions. This is why we separate rule partitions from Algorithm $\mathcal{ALCG}_{\mathrm{sc}}$.

Example 10 (Complements of Injective Functions). Consider the function, $mapnot$, defined as follows.

$$\begin{array}{ll} mapnot(\mathsf{Cons}(a, x)) & \mathrel{\hat{=}} \mathsf{Cons}(not(a), mapnot(x)) \\ mapnot(\mathsf{Nil}) & \mathrel{\hat{=}} \mathsf{Nil} \\ not(\mathsf{True}) & \mathrel{\hat{=}} \mathsf{False} \\ not(\mathsf{False}) & \mathrel{\hat{=}} \mathsf{True} \end{array}$$

In contrast with $mapfst$ defined above, $mapnot$ is injective. With injective analysis we know that $mapnot$ and not are injective functions, so $\mathcal{ALCG}_{\mathrm{sc}}$ returns the following complement functions

$$\begin{array}{ll} mapnot^c(\mathsf{Cons}(a, x)) & \mathrel{\hat{=}} \mathsf{B}_1 \\ mapnot^c(\mathsf{Nil}) & \mathrel{\hat{=}} \mathsf{B}_1 \\ not^c(\mathsf{True}) & \mathrel{\hat{=}} \mathsf{B}_2 \\ not^c(\mathsf{False}) & \mathrel{\hat{=}} \mathsf{B}_2 \end{array}$$

which is a minimal complement function of $mapnot$ with respect to the collapsing order.

It is worth remarking that Algorithm $\mathcal{ALCG}_{\mathrm{sc}}$ will derive constant functions for injective functions if a partition of $\mathcal{R}$ is $\mathcal{R} = \mathcal{R}_{f_1} \uplus \cdots \uplus \mathrm{R}_{f_n}$ where $\mathcal{R}_{f_i}$ is the set of all rules for f_i. The existence of such a partition is easily checked by the range analysis.

Example 11 (Removing Constructors). Consider the function zip in Example 5. Algorithm $\mathcal{ALCG}_{\mathrm{sc}}$ returns

$$\begin{array}{ll} zip^c(\mathsf{Nil}, y) & \mathrel{\hat{=}} \mathsf{B}_1(y) \\ zip^c(\mathsf{Cons}(a, x), \mathsf{Nil}) & \mathrel{\hat{=}} \mathsf{B}_2(a, x) \\ zip^c(\mathsf{Cons}(a, x), \mathsf{Cons}(b, y)) & \mathrel{\hat{=}} zip^c(x, y) \end{array}$$

which is a minimal complement function of zip^c. Note that $\mathcal{ALCG}_{\mathrm{sc}}$ has removed the constructor from the third rule, compared to the old algorithm, $\mathcal{ALCG}_{\mathrm{c}}$.

The complement functions obtained by $\mathcal{ALCG}_{\mathrm{sc}}$ have two good characteristics. First, they have the same form as view functions, which makes the later tupling step and the inversion step easy. Second, as will be seen later, the updatability of backward transformation functions with these complement functions is easy to understand.

6. Generating Backward Transformation Functions

After obtaining a view complement function $f^c : S \to V'$ for a given view function $f : S \to V$, we get the following backward transformation according to Equation (UPD).

$$\rho(s, v) = (f \vartriangle f^c)^{-1}(v, f^c(s))$$

That is, a backward transformation function can be derived if the tupled function, $(f \vartriangle f^c)$, and its inverse $(f \vartriangle f^c)^{-1}$ can be effectively derived.

The point is how to calculate an inverse program. Although this is generally difficult, now we need merely to treat the tupled function of the form $(f \vartriangle f^c)$. Thanks to the correspondence between the rules of f and f^c, we can obtain a program of $(f \vartriangle f^c)$ which is in a good form for this inversion. In the following, we show how tupling and inversion can be done automatically.

6.1 Calculation of Program of $(f \vartriangle f^c)$

A program for tupled function $(f \vartriangle f^c)$ can be straightforwardly calculated because the rules of f and the corresponding f^c have the same patterns and the same form of recursive calls in the right-hand sides. However, we cannot directly describe the tupled function in the treeless form because of the tuple structure, which needs to be treated specially. We extend language VDL with "**where**-clauses" and tuples:

$$\begin{array}{l} \mathit{rule} ::= \cdots \\ \quad \mid\ f(p_1, \ldots, p_n) \mathrel{\hat{=}} (e_1, \ldots, e_m) \\ \qquad \textbf{where}\ (x_1, \ldots, x_k) \mathrel{\hat{=}} g(y_1, \ldots, y_m) \\ \qquad\qquad \cdots \\ \qquad\qquad (x'_1, \ldots, x'_{k'}) \mathrel{\hat{=}} g'(y'_1, \ldots, y'_{m'}) \end{array}$$

where $e_1, \ldots, e_n$ do not include any function calls, i.e., they have the same forms as patterns, and all variables appear on the left-hand sides of the **where**-clause are different from those on the right-hand sides. The two restrictions above are the treeless condition of tupled functions. Additionally, this new form of rules must satisfy the affine condition. That is, all variables used at most once (actually, all variables are used exactly once in tupled functions). The operational semantics is straightforwardly extended. Note that this extension is behind the scene of the view definition users; it is only used internally during bidirectionalization transformation.

Algorithm 4 (Tupling).

Input: A **where**-free program P.

Output: A program P^Δ for tupled functions.

Procedure:

1. Let P^c be the program derived from P by $\mathcal{ALG}_{\mathrm{sc}}$.
2. For each non-injective function f, and for each rule r of f in P do
   (a) Let r^c be the corresponding rule for r in P^c.
   (b) Structure r and r^c in the following forms
   $$r = f(\overrightarrow{p}) \mathrel{\hat{=}} K[\overrightarrow{t}, \overrightarrow{u}, \overrightarrow{x}]$$
   $$r^c = f^c(\overrightarrow{p}) \mathrel{\hat{=}} K'[\overrightarrow{t'}, \overrightarrow{x'}]$$
   where
   - $\overrightarrow{t}$: *non-injective* function calls $f_1(\overrightarrow{y_1}), \ldots, f_n(\overrightarrow{y_n})$,
   - $\overrightarrow{t}'$: function calls of the forms ${f_1}^c(\overrightarrow{y_1}), \ldots, {f_n}^c(\overrightarrow{y_n})$,
   - $\overrightarrow{u}$: *injective* function calls $g_1(\overrightarrow{z_1}), \ldots, g_m(\overrightarrow{z_m})$.

   (c) Prepare fresh variables $\mathtt{t}_i, \mathtt{t}'_i, \mathtt{u}_j$ for $i \in \{1, \ldots, n\}, j \in \{1, \ldots, m\}$.
   (d) Construct the rule, r^Δ, as follows:
   $$r^\Delta = f^\Delta(\overrightarrow{p}) \mathrel{\hat{=}} (K[\overrightarrow{\mathtt{t}}, \overrightarrow{\mathtt{u}}, \overrightarrow{x}], K'[\overrightarrow{\mathtt{t}'}, \overrightarrow{x'}])$$
   $$\textbf{where } \{(\mathtt{t}_i, \mathtt{t}'_i) \mathrel{\hat{=}} {f_i}^\Delta(\overrightarrow{y_i})\}_{i\in\{1,\ldots,n\}}$$
   $$\{\mathtt{u}_j \mathrel{\hat{=}} g_j(\overrightarrow{z_j})\}_{j\in\{1,\ldots,m\}}$$
3. For each injective function g, and for each rule r of g in P, construct the rule, r', as follows in the similar way:
   $$r' = g(\overrightarrow{p}) \mathrel{\hat{=}} K[\overrightarrow{\mathtt{u}}, \overrightarrow{x}]$$
   $$\textbf{where } \{\mathtt{u}_j \mathrel{\hat{=}} g_j(\overrightarrow{z_j})\}_{j\in\{1,\ldots,m\}}$$
4. Gather all r^Δ and r' to form P^Δ. □

This algorithm correctly gives a program of tupled functions, i.e., $[\![f^\Delta]\!](\overrightarrow{t}) = ([\![f]\!] \vartriangle [\![f^c]\!])(\overrightarrow{t})$. An example is given in Section 8.

6.2 Calculation of Program of $(f \vartriangle f^c)^{-1}$

Next, we calculate an inverse program for $(f \vartriangle f^c)^{-1}$ from the program of $(f \vartriangle f^c)$. The basic idea is to swap the left-hand side and the right-hand side of each rule and to apply inversion recursively.

Algorithm 5 (Inversion of Tupled Functions).

Input: A program $P^\Delta = (\mathcal{R}, \mathcal{F}, \mathcal{C}, \mathcal{X})$ for tupled functions.

Output: A program
$$(P^\Delta)^{-1} = \left(\mathcal{R}^{-1}, \left\{f^{-1} \mid f \in \mathcal{F}\right\}, \mathcal{C}, \mathcal{X}\right)$$
for tupled functions.

Procedure: For each rule r in $\mathcal{R}$
$$r = f(\overrightarrow{p}) \mathrel{\hat{=}} (\overrightarrow{e}) \textbf{ where } \{(\mathtt{t}_i, \mathtt{t}'_i) \mathrel{\hat{=}} f_i(\overrightarrow{y_i})\}_{i\in\{1,\ldots,n\}} \{\mathtt{u}_j \mathrel{\hat{=}} g_j(\overrightarrow{z_j})\}_{j\in\{1,\ldots,m\}}$$
construct the rule r^{-1} in $\mathcal{R}^{-1}$ as follows.
$$r^{-1} = f^{-1}(\overrightarrow{e}) \mathrel{\hat{=}} (\overrightarrow{p}) \textbf{ where } \{(\overrightarrow{y_i}) \mathrel{\hat{=}} f_i^{-1}(\mathtt{t}_i, \mathtt{t}'_i)\}_{i\in\{1,\ldots,n\}} \{(\overrightarrow{z_j}) \mathrel{\hat{=}} g_j^{-1}(\mathtt{u}_j)\}_{j\in\{1,\ldots,m\}}$$ □

Theorem 5 (Correctness). Let P be a program, and $(P^\Delta)^{-1}$ the generated program. Then, $(u_1, u_2) = [\![f^\Delta]\!]_{P^\Delta}(t_1, \ldots, t_n)$ implies $(t_1, \ldots, t_n) = [\![(f^\Delta)^{-1}]\!]_{(P^\Delta)^{-1}}(u_1, u_2)$.

Note that the obtained inverse program may be nondeterministic. However, since the original function to be inversed is injective in our framework, it is possible to uniquely determine a rule with a domain analysis similar to the range analysis discussed before, when the inverse program is executed. An example of inversion is given in Section 8.

7. Generating View Update Checker

A view update checker is designed to decide whether or not an update on views is valid without execution of the backward transformation. An update on views is said to be valid if it can be successfully reflected to the source by the derived backward transformation function. Recall that in our framework, a backward transformation is given by $\mathsf{upd}_{\langle f, f^c\rangle}(s, v) \mathrel{\hat{=}} (f \vartriangle f^c)^{-1}(v, f^c(s))$. This means that, for a view function f and the original source, s, we can check whether or not a view update is valid by confirming whether $(v, f^c(s))$ is in the range of $(f \vartriangle f^c)$, where v is an updated view.

We define below a nondeterministic (bottom-up) tree automaton for validating view updates. The tree automaton has three kinds of states: state q_* is reached by any view, state q_f is reached by a view in the range of f, and state $q^t_{f^\Delta}$ is reached by a view v such that (v, t) is in the range of f^Δ. Therefore, when the final state is a state $q^{t_0}_{f^\Delta}$, the tree automaton exactly accepts a view v such that (v, t_0) is in the range of f^Δ (i.e., (v, t_0) is in the domain of $(f^\Delta)^{-1}$).

Definition 3 (View Update Checker). Let P be a program, P^c be a complement program derived by our algorithm, t_0 be a complement view, and $P^\Delta = (\mathcal{R}^\Delta, \mathcal{F}^\Delta, \mathcal{C}^\Delta, \mathcal{X}^\Delta)$ be a tupled program of P and P^c. A *view updating checker* is defined as a tree automaton $A_{\mathsf{U}} = (Q, \mathcal{C}^\Delta, \{q_f^{t_0}\}, \Delta)$ where

- $Q = \{q_*\} \cup \{q_f \mid f \in \mathcal{F}\} \cup \{q^t_{f^\Delta} \mid f^\Delta \in \mathcal{F}^\Delta, t \text{ is a subtree of } t_0\}$
- Δ consists of the following transition rules:
  - $\mathsf{C}(q_*, \ldots, q_*) \to q_*$ with $\mathsf{C} \in \mathcal{C}$,
  - $K[\overrightarrow{q_f}, \overrightarrow{q_*}] \to q_f$ with $f(\overrightarrow{p}) \mathrel{\hat{=}} K[\overrightarrow{t}, \overrightarrow{x}] \in \mathcal{R}$ and $t_i = f'_i(\overrightarrow{e})$, and
  - $K[\overrightarrow{q^{t''\theta}_{f'^\Delta}}, \overrightarrow{q_{g'}}, \overrightarrow{q_*}] \to q^{t''}_{f^\Delta}$ with
  $$\left[\begin{array}{l} f^\Delta(\overrightarrow{p}) \mathrel{\hat{=}} (K[\overrightarrow{t}, \overrightarrow{u}, \overrightarrow{x}], K'[\overrightarrow{t'}, \overrightarrow{x'}]) \\ \quad \textbf{where } \{(t_i, t'_i) \mathrel{\hat{=}} f'^\Delta_i(\overrightarrow{y_i})\}_{i\in\{1,\ldots,n\}} \\ \qquad\qquad \{u_j \mathrel{\hat{=}} g'_j(\overrightarrow{z_j})\}_{j\in\{1,\ldots,m\}} \end{array}\right] \in \mathcal{R}^\Delta$$
  where t'' is a subtree of t_0 and $t'' = K'[\overrightarrow{t'}, \overrightarrow{x'}]\theta$.

Theorem 6 (Validity of View Update Checker). A view updating checking tree automaton $A_{\mathsf{U}} = (Q, \mathcal{C}^\Delta, \{q^{t_0}_{f^\Delta}\}, \Delta)$ in Definition 3 exactly accepts view v such that (v, t_0) is in the range of f^Δ.

We show examples of automatically generated view update checkers for some view functions. Note that the view update checking automata below have been reduced where unnecessary states have been removed.

Example 12. Consider function *append* and its complement function in Section 3. When the initial source is
$$(s_1, s_2) = (\mathsf{Cons}(\mathsf{True}, \mathsf{Cons}(\mathsf{False}, \mathsf{Nil})), \mathsf{Nil}),$$
the view is $append(s_1, s_2) = \mathsf{Cons}(\mathsf{True}, \mathsf{Cons}(\mathsf{False}, \mathsf{Nil}))$ and the complement view is $append^c(s_1, s_2) = \mathsf{B}_2(\mathsf{B}_2(\mathsf{B}_1))$.

Then, the view update checker generated by our system is the automaton $A = (Q, \mathcal{C}, \{q^{\mathsf{B}_2(\mathsf{B}_2(\mathsf{B}_1))}_{append^\Delta}\}, \Delta)$ where

- $Q = \{q_*, q^{\mathsf{B}_2(\mathsf{B}_2(\mathsf{B}_1))}_{append^\Delta}, q^{\mathsf{B}_2(\mathsf{B}_1)}_{append^\Delta}, q^{\mathsf{B}_1}_{append^\Delta}\}$ and
- Δ consists of the transition rules of
  - $t \to q_*$ where $t \in \{\mathsf{True}, \mathsf{False}, \mathsf{Nil}\}$,
  - $\mathsf{Cons}(q_*, q_*) \to q_*$,
  - $\mathsf{Cons}(q_*, q^{\mathsf{B}_2(\mathsf{B}_1)}_{append^\Delta}) \to q^{\mathsf{B}_2(\mathsf{B}_2(\mathsf{B}_1))}_{append^\Delta}$,

- $\mathsf{Cons}(q_*, q^{\mathsf{B}_1}_{append^\Delta}) \to q^{\mathsf{B}_2(\mathsf{B}_1)}_{append^\Delta}$, and
- $q_* \to q^{\mathsf{B}_1}_{append^\Delta}$.

In fact, this automaton only accepts lists that are 2 or longer, which means one can only update the view of lists in such a way that its length is not less than 2.

Example 13. Consider the function, *filter*, defined as

$$\begin{array}{ll} \mathit{filter}(\mathsf{Nil}) & \hat{=}\ \mathsf{Nil} \\ \mathit{filter}(\mathsf{Cons}(\mathsf{A}_1, x)) & \hat{=}\ \mathsf{Cons}(\mathsf{A}_1, \mathit{filter}(x)) \\ \mathit{filter}(\mathsf{Cons}(\mathsf{A}_2, x)) & \hat{=}\ \mathsf{Cons}(\mathsf{A}_2, \mathit{filter}(x)) \\ \mathit{filter}(\mathsf{Cons}(\mathsf{A}_3, x)) & \hat{=}\ \mathit{filter}(x) \end{array}$$

and the complement function derived by our algorithm as follows.

$$\begin{array}{ll} \mathit{filter}^c(\mathsf{Nil}) & \hat{=}\ \mathsf{B}_1 \\ \mathit{filter}^c(\mathsf{Cons}(\mathsf{A}_1, x)) & \hat{=}\ \mathsf{B}_2(\mathit{filter}^c(x)) \\ \mathit{filter}^c(\mathsf{Cons}(\mathsf{A}_2, x)) & \hat{=}\ \mathsf{B}_2(\mathit{filter}^c(x)) \\ \mathit{filter}^c(\mathsf{Cons}(\mathsf{A}_3, x)) & \hat{=}\ \mathsf{B}_3(\mathit{filter}^c(x)) \end{array}$$

When the initial source is

$$s = \mathsf{Cons}(\mathsf{A}_2, \mathsf{Cons}(\mathsf{A}_3, \mathsf{Cons}(\mathsf{A}_1, \mathsf{Nil}))),$$

the view is $\mathit{filter}(s) = \mathsf{Cons}(\mathsf{A}_2, \mathsf{Cons}(\mathsf{A}_1, \mathsf{Nil}))$ and the complement view is $\mathit{filter}^c(s) = \mathsf{B}_2(\mathsf{B}_3(\mathsf{B}_2(\mathsf{B}_1)))$.

Then, the view update checker derived by our system is automaton $A = (Q, \mathcal{C}, \{q^{\mathsf{B}_2(\mathsf{B}_3(\mathsf{B}_2(\mathsf{B}_1)))}_{filter^\Delta}\}, \Delta)$ where

- $Q = \{q^{\mathsf{B}_2(\mathsf{B}_3(\mathsf{B}_2(\mathsf{B}_1)))}_{filter^\Delta}, q^{\mathsf{B}_3(\mathsf{B}_2(\mathsf{B}_1))}_{filter^\Delta}, q^{\mathsf{B}_2(\mathsf{B}_1)}_{filter^\Delta}, q^{\mathsf{B}_1}_{filter^\Delta}\}$
- Δ consists of transition rules
  - $\mathsf{Cons}(t, q^{\mathsf{B}_3(\mathsf{B}_2(\mathsf{B}_1))}_{filter^\Delta}) \to q^{\mathsf{B}_2(\mathsf{B}_3(\mathsf{B}_2(\mathsf{B}_1)))}_{filter^\Delta}$ where $t \in \{\mathsf{A}_1, \mathsf{A}_2\}$,
  - $\mathsf{Cons}(t, q^{\mathsf{B}_1}_{filter^\Delta}) \to q^{\mathsf{B}_2(\mathsf{B}_1)}_{filter^\Delta}$ where $t \in \{\mathsf{A}_1, \mathsf{A}_2\}$,
  - $q^{\mathsf{B}_2(\mathsf{B}_1)}_{filter^\Delta} \to q^{\mathsf{B}_3(\mathsf{B}_2(\mathsf{B}_1))}_{filter^\Delta}$, and
  - $\mathsf{Nil} \to q^{\mathsf{B}_1}_{filter^\Delta}$.

This automaton accepts lists that are 2 long, and each list element is either A_1 or A_2.

8. An Example

To give a whole picture of how our system works concretely, recall the example in the Introduction. The following view function, *students*, is the same as that in the Introduction, except that we write Cons for (:) and Nil for [].

$$\begin{array}{l} \mathit{students}(\mathsf{Nil}) \hat{=} \mathsf{Nil} \\ \mathit{students}(\mathsf{Cons}(\mathsf{Student}(\mathit{name}, \mathit{grade}, \mathit{major})), \mathit{ms})) \\ \quad \hat{=} \mathsf{Cons}(\mathsf{Student}(\mathit{name}, \mathit{grade}, \mathit{major}), \mathit{students}(\mathit{ms})) \\ \mathit{students}(\mathsf{Cons}(\mathsf{Prof}(\mathit{name}, \mathit{position}, \mathit{major})), \mathit{ms})) \\ \quad \hat{=} \mathit{students}(\mathit{ms}) \end{array}$$

The function, *students*, extracts all student members from a member list. This behavior of *students* is similar to the function, *filter*. The derived complement function by our algorithm is as follows.

$$\begin{array}{l} \mathit{students}^c(\mathsf{Nil}) \hat{=} \mathsf{B}_1 \\ \mathit{students}^c(\mathsf{Cons}(\mathsf{Student}(\mathit{name}, \mathit{grade}, \mathit{major})), \mathit{ms})) \\ \quad \hat{=} \mathsf{B}_2(\mathit{students}^c(\mathit{ms})) \\ \mathit{students}^c(\mathsf{Cons}(\mathsf{Prof}(\mathit{name}, \mathit{position}, \mathit{major})), \mathit{ms})) \\ \quad \hat{=} \mathsf{B}_3(\mathit{name}, \mathit{position}, \mathit{major}, \mathit{students}^c(\mathit{ms})) \end{array}$$

Tupling the two functions *students* and $\mathit{students}^c$ gives

$$\begin{array}{l} \mathit{students}^\Delta(\mathsf{Nil}) \hat{=} (\mathsf{Nil}, \mathsf{B}_1) \\ \mathit{students}^\Delta(\mathsf{Cons}(\mathsf{Student}(\mathit{name}, \mathit{grade}, \mathit{major}), \mathit{ms})) \\ \quad \hat{=} (\mathsf{Cons}(\mathsf{Student}(\mathit{name}, \mathit{grade}, \mathit{major}), x), \mathsf{B}_2(y)) \\ \qquad \textbf{where}\ (x, y) \hat{=} \mathit{students}^\Delta(\mathit{ms}) \\ \mathit{students}^\Delta(\mathsf{Cons}(\mathsf{Prof}(\mathit{name}, \mathit{position}, \mathit{major}), \mathit{ms})) \\ \quad \hat{=} (x, \mathsf{B}_3(\mathit{name}, \mathit{position}, \mathit{major}, y)) \\ \qquad \textbf{where}\ (x, y) \hat{=} \mathit{students}^\Delta(\mathit{ms}), \end{array}$$

and inversion of this tupled function yields the following result.

$$\begin{array}{l} (\mathit{students}^\Delta)^{-1}(\mathsf{Nil}, \mathsf{B}_1) \hat{=} \mathsf{Nil} \\ (\mathit{students}^\Delta)^{-1}(\mathsf{Cons}(\mathsf{Students}(n, g, m), x), \mathsf{B}_2(y)) \\ \quad \hat{=} \mathsf{Cons}(\mathsf{Student}(n, g, m), \mathit{ms}) \\ \qquad \textbf{where}\ \mathit{ms} \hat{=} (\mathit{students}^\Delta)^{-1}(x, y) \\ (\mathit{students}^\Delta)^{-1}(x, \mathsf{B}_3(n, p, m, y)) \\ \quad \hat{=} \mathsf{Cons}(\mathsf{Prof}(n, p, m), \mathit{ms}) \\ \qquad \textbf{where}\ \mathit{ms} \hat{=} (\mathit{students}^\Delta)^{-1}(x, y) \end{array}$$

Then, a backward transformation $\rho = \mathsf{upd}_{\langle \mathit{students}, \mathit{students}^c \rangle}$ can be derived (after some fusion transformation) as follows.

$$\begin{array}{l} \rho(\mathsf{Nil}, \mathsf{Nil}) = \mathsf{Nil} \\ \rho(\mathsf{Cons}(\mathsf{Student}(n, g, m), \mathit{ms}), \mathsf{Cons}(\mathsf{Student}(n', g', m'), \mathit{ss})) \\ \quad = \mathsf{Cons}(\mathsf{Student}(n', g', m'), \rho(\mathit{ms}, \mathit{ss})) \\ \rho(\mathsf{Cons}(\mathsf{Prof}(n, g, m), \mathit{ms}), \mathit{ss}) \\ \quad = \mathsf{Cons}(\mathsf{Prof}(n, g, m), \rho(\mathit{ms}, \mathit{ss})) \end{array}$$

This is exactly the same function as $\mathit{students}_{\mathrm{B}}$ in the Introduction.

Now one can freely change the names in the view, and the backward transformation can reflect them to the source. Consider the case where the source s is as follows.

$$s = \mathsf{Cons}(\mathsf{Student}(\mathsf{X}, \mathsf{DC}, \mathsf{CS}), \mathsf{Cons}(\mathsf{Prof}(\mathsf{Y}, \mathsf{AP}, \mathsf{CS}), \mathsf{Nil}))$$

Let v be the view generated by the view function, *students*, on s, i.e., $v = \mathsf{Cons}((\mathsf{Student}(\mathsf{X}, \mathsf{DC}, \mathsf{CS}), \mathsf{Nil})$. Updating view v to $\mathsf{Cons}(\mathsf{Student}(\mathsf{X}, \mathsf{DC}, \mathsf{Math}), \mathsf{Nil})$ is acceptable and results in the following source.

$$\mathsf{Cons}(\mathsf{Student}(\mathsf{X}, \mathsf{DC}, \mathsf{Math}), \mathsf{Cons}(\mathsf{Prof}(\mathsf{Y}, \mathsf{AP}, \mathsf{CS}), \mathsf{Nil}))$$

However, both inserting and removing elements, e.g., updating the view to Nil, are prohibited.

This updatability can be precisely represented by an automaton. Let

$$\begin{array}{lcl} v_1 & = & \mathsf{B}_2(\mathsf{B}_3(\mathsf{Y}, \mathsf{AP}, \mathsf{CS}, \mathsf{B}_1)) \\ v_2 & = & \mathsf{B}_3(\mathsf{Y}, \mathsf{AP}, \mathsf{CS}, \mathsf{B}_1) \\ v_3 & = & \mathsf{B}_1, \end{array}$$

then updatability of v with respect to source s is captured by automaton $A = (Q, \mathcal{C}, \{q^{v_1}_{students}\}, \Delta)$, where

- $Q = \{q^{v_1}_{students}, q^{v_2}_{students}, q^{v_3}_{students}, q_*\}$
- Δ consists of transition rules
  - $\mathsf{C}(q_*, \ldots, q_*) \to q_*$ where $\mathsf{C} \in \mathcal{C}$,
  - $\mathsf{Cons}(\mathsf{Student}(q_*, q_*, q_*), q^{v_2}_{students}) \to q^{v_1}_{students}$,
  - $q^{v_3}_{students} \to q^{v_2}_{students}$, and
  - $\mathsf{Nil} \to q^{v_3}_{students}$.

Note that the view, $\mathsf{Cons}((\mathsf{Student}(\mathsf{X}, \mathsf{DC}, \mathsf{CS}), \mathsf{Nil})$, is accepted by the automaton, A, but the view, Nil, is not.

9. Related Work

Our work is based on the idea of deriving (relational) complement functions on relational databases, where view functions are expressed in terms of relational algebras. Cosmadakis and Papadimitriou (1984) showed that finding minimal complement functions when views are defined only by projections is NP-Complete. Laurent et al. (2001) proposed an algorithm to compute complement functions when views are defined by projections, selections, and joins. They also discussed the conditions for minimal complement functions. Their algorithm is expensive because it uses the results of NP-Complete sub-problems. Lechtenbörger and Vossen (2003) improved on Laurent et al.'s work. They proposed a polynomial-time algorithm for computing complement functions when views are defined by projections, selections, joins, and renaming. Their

algorithm computes smaller complement functions than Laurent et al.'s, and the obtained complement functions are minimal when view functions contain no projection. Our work can be considered an extension of these works such that view functions can be defined on tree-like data structures other than tuples. Moreover, our approach is based on syntactic program transformations, whereas the existing methods focus more on function semantics.

Our work was greatly motivated by works on bidirectional transformation on trees. In addition to the work by Foster et al. (2005) that proposed a combinatorial approach to the problem (as discussed in the Introduction), Hu et al. (2004) showed how bidirectional transformation can be used to maintain data dependency in the tree view, which can be seen as an application that develops constraint maintainers on trees (Meertens 1998). Mu et al. (2004) improved the consistency of Hu et al.'s framework by imposing restrictions on updating operators. These frameworks are domain-specific and designed for specific applications. In contrast, our work is more general.

Our work is also related to work on updating XML views constructed from relational data. Wang and Rundensteiner (2004) applied the work in Dayal and Bernstein (1982) to XML views over relational data. Braganholo et al. (2004) proposed an algorithm to map updates on XML views to relational data. However, they did not consider the case where the source is XML too, and our method may shed a new light on their problems.

10. Conclusion

This paper presents a new transformational approach to bidirectionalization that can automatically derive backward transformation programs from view definition programs written in a simple functional language. The new bidirectionalization method is built upon three program transformations: automatic derivation of complement functions, tupling transformation, and inverse transformation. These three transformations are composed together through programs in a treeless and affine form, which simplifies and enables implementation of the transformations and automatic generation of a view update checker. Our approach is different from many existing approaches where only a bidirectional interpreter is derived and execution of the interpreter requires passing the source through all the interpretation steps, our approach does produce a program for backward transformation. This makes it possible to utilize an optimizing compiler for more efficient execution of backward transformation.

There are several issues that are worth looking into in the future. First, it would be interesting to see if our view language could be extended with regular patterns and the restriction on variable uses could be relaxed so that more view functions could be defined and backward transformation could be derived. Second, we want to see if our bidirectionalization framework can be adapted to other bidirectional semantics. The bidirectional properties (semantics) used in this paper are known as "closed update semantics" where sources are invisible to the users who want to modify views. There is another useful semantics called "open update semantics" where both the source and the view are visible to users. Third, although the system has been implemented, we would like to test it with more practical applications.

References

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

V. P. Braganholo, S. B. Davidson, and C. A. Heuser. From XML view updates to relational view updates: old solutions to a new problem. In *VLDB '04: International Conference on Very Large Data Bases*, pages 276–287, 2004.

H. Comon, M. Dauchet, R. Gilleron, F. Jacquemard, D. Lugiez, S. Tison, and M. Tommasi. Tree automata techniques and applications. `http://www.grappa.univ-lille3.fr/tata`, 1997.

S. S. Cosmadakis and C. H. Papadimitriou. Updates of relational views. *J. ACM*, 31(4):742–760, 1984.

U. Dayal and P. A. Bernstein. On the correct translation of update operations on relational views. *ACM T. Database Syst.*, 7(3):381–416, 1982.

J. Engelfriet. Bottom-up and Top-down Tree Transformations — A Comparison. *Math. Syst. Theory*, 9(3):198–231, 1975.

J. N. Foster, M. B. Greenwald, J. T. Moore, B. C. Pierce, and A. Schmitt. Combinators for bi-directional tree transformations: a linguistic approach to the view update problem. In *POPL '05: Proceedings of the 32nd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 233–246, 2005.

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

S. J. Hegner. Foundations of canonical update support for closed database views. In *ICDT '90: Proceedings of the Third International Conference on Database Theory*, pages 422–436, 1990.

Z. Hu, S.-C. Mu, and M. Takeichi. A programmable editor for developing structured documents based on bidirectional transformations. In *PEPM '04: Proceedings of the 2004 ACM SIGPLAN symposium on Partial evaluation and semantics-based program manipulation*, pages 178–189, 2004.

R. Lämmel. Coupled software transformations (extended abstract). In *First International Workshop on Software Evolution Transformations*, pages 31–35, 2004.

D. Laurent, J. Lechtenbörger, N. Spyratos, and G. Vossen. Monotonic complements for independent data warehouses. *VLDB J.*, 10(4):295–315, 2001.

J. Lechtenbörger and G. Vossen. On the computation of relational view complements. *ACM T. Database Syst.*, 28(2):175–208, 2003.

K. Matsuda, Z. Hu, K. Nakano, M. Hamana, and M. Takeichi. Bidirectionalization transformation based on automatic derivation of view complement function. Technical Report 2007-44, Graduate School of Information Science and Technology, the University of Tokyo, 2007.

L. Meertens. Designing constraint maintainers for user interaction. `http://www.cwi.nl/~lambert`, 1998.

S.-C. Mu, Z. Hu, and M. Takeichi. An algebraic approach to bidirectional updating. In *APLAS '04: Second ASIAN Symposium on Programming Languages and Systems*, pages 2–18, 2004.

P. Wadler. Views: a way for pattern matching to cohabit with data abstraction. In *POPL '87: Proceedings of the 14th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, pages 307–313, 1987.

P. Wadler. Deforestation: Transforming programs to eliminate trees. *Theor. Comput. Sci.*, 73(2):231–248, 1990.

L. Wang and E. A. Rundensteiner. On the updatability of XML views published over relational data. In *ER 2004: International Conference on Conceptual Modeling*, pages 795–809, 2004.