A Language for Bidirectional Updating Based on Injective Mapping

Shin-Cheng Mu   Zhenjiang Hu   Masato Takeichi

Department of Information Engineering
University of Tokyo
7-3-1 Hongo, Bunkyo-ku, Tokyo 113, Japan
{scm,hu,takeichi}@ipl.t.u-tokyo.ac.jp

Abstract

With the popularity of XML, bidirectional updating – maintaining the consistency between two pieces of structured data, again becomes a problem of interest. In this paper we propose a new framework, inspired by program inversion, to look at this problem. We design a functional language, Inv, in which all programs are injective and their inverses can be trivially constructed. The language is equivalent to reversible Turing machines and expressive enough to describe most transforms one would need. It is assumed that the data to be synchronised are related by a transform written in Inv. The semantics of Inv is then extended to deal with bidirectional updating. Given a transform, its synchronisation behaviour can be derived by algebraic reasoning. The main feature of our approach is being able to deal with duplication and structural changes.

Keywords: bi-directional transformation, view-updating, program inversion, algebraic program transformation.

1 Introduction

In many occasions would one encounter the task of maintaining the consistency between two pieces of structured data that are related by some transform. In some XML editors, for example [3, 23, 22], a source XML document is transformed to a user-friendly, editable *view* through a transform defined by the document designer. The editing performed by the user on the view needs to be reflected back to the source document. Similar techniques can also be used to synchronise several bookmarks stored in formats of different browsers, to maintain invariance among widgets in an user interface, or to maintain the consistency of data and view in databases.

As a canonical example, consider the XML document in Figure 1 representing an address book. When being displayed to the user, it might be converted to an HTML document as in Figure 2, with an additional index of names. The conversion is defined by the document designer in some domain-specific programming language. We would then wish that when the user, for example, adds or deletes a person in Figure 2, the original document in Figure 1 be updated correspondingly. Further more, the changes should also trigger an update of the index of names in the view. We may even wish that when an additional name is added to the index, a fresh, empty contact be added to the address book in both the source and the view. All these are better done without too much effort, other than specifying the transform itself, from the document designer.

View-updating [6, 9, 13, 20, 1] has been intensively studied in the database community. Recently, the problem of maintaining the consistency of two pieces of structured data was brought to our attention again by [16] and [14]. Though developed separately, their results turn out to be surprisingly similar, with two important features missing. Firstly, certain healthiness conditions ruled out those transforms that duplicate data. Secondly, structural changes (aligning structures after inserting to or deleting from a list or a tree) were not sufficiently dealt with.

In this paper, we propose a different view to look at the problem of bi-directional updating, inspired by previous studies of program inversion [2, 11]. The transform between the source and the view is specified as an injective function. We believe that it is a clear and suitable framework for such tasks. The main contributions of this paper are as follows.

- We designed a functional language, Inv (Section 3), in which only injective functions are definable and therefore every program is trivially invert-

```
<addrbook>
  <person>
    <name> Masato Takeichi </name>
    <email> takeichi@acm.org </email>
    <tel> +81-3-5841-7430 </tel>
  </person>
  <person>
    <name> Zhenjiang Hu </name>
    <email> hu@mist.i.u-tokyo.ac.jp </email>
    <email> hu@ipl.t.u-tokyo.ac.jp </email>
    <tel> +81-3-5841-7411 </tel>
  </person>
  <person>
    <name> Shin-Cheng Mu </name>
    <email> scm@mist.i.u-tokyo.ac.jp </email>
    <tel> +81-3-5841-7411 </tel>
  </person>
</addrbook>
```

Figure 1: An address book represented in XML.

```
<html>
  <ol>
    <li> Zhenjiang Hu </li>
    <li> Shin-Cheng Mu </li>
    <li> Masato Takeichi </li>
  </ol>
  <dl>
    <dt> Zhenjiang Hu </dt>
    <dd> hu@mist.i.u-tokyo.ac.jp <br/>
         hu@ipl.t.u-tokyo.ac.jp <br/>
         +81-3-5841-7411 </dd>
    <dt> Shin-Cheng Mu </dt>
    <dd> scm@mist.i.u-tokyo.ac.jp <br/>
         +81-3-5841-7411 </dd>
    <dt>Masato Takeichi </dt>
    <dd> takeichi@acm.org <br/>
         +81-3-5841-7430 </dd>
  </dl>
</html>
```

Figure 2: A view of the address book.

ible. A special operator for duplication specifies all element-wise dependency. In this language we can specify most of the transform we want, including the lenses given by [14] (Section 3.2.3).

- Given any possibly non-injective function defined in a simple functional language, we show how to mechanically compile it into Inv by annotating the output with extra information (Section 3.3). It shows that Inv is equivalent to reversible Turing machines [7].
- To deal with inconsistencies resulting from user editing, we give Inv an alternative semantics, under which the behaviour of programs can be reasoned by algebraic reasoning (Section 4). It can deal with duplication as well as structural changes, with an additional cost of introducing extra tags, which is not suitable for *loosely coupled* synchronisation such as [14], but okay for our purpose.

In Section 5 we will see more useful example transforms useful in an editor, including list reversal, filtering, merging and splitting ordered lists, as well as how they react to user editing. The result will soon be integrated into our XML editor in the Programmable Structured Document project [23].

2 From Relations to Functions

In this section and the next we will give a brief introduction of relations, a point-free functional language Fun, and finally the injective language Inv, each of them a refinement of the former.

2.1 Views

The *View* datatype defines the basic types of data we deal with.

$$\begin{array}{lcl}
View & ::= & () \\
 & & \mid Int \mid String \\
 & & \mid (View, View) \\
 & & \mid L\ View \mid R\ View \\
 & & \mid List\ View \mid Tree\ View \\
 & & \\
List\ a & ::= & [\,] \mid a : List\ a \\
Tree\ a & ::= & Node\ a\ (List\ (Tree\ a))
\end{array}$$

The atomic types include integer, string, and unit, the type having only one value (). Composite types include pairs, sum (*L View* and *R View*), lists (*List View*) and rose trees (*Tree View*). The (:) operator, forming lists, associates to the right. We also follow the convention writing the list $1{:}2{:}3{:}[\,]$ as $[1,2,3]$. More extensions dealing with editing will be given in Section 4.1.

For XML processing we can think of XML documents as rose trees. This very simplified view omits some non-essential features and some more tricky features of XML, which will be one of our future work.

2.2 Relations

The program derivation community started rebuilding the theory of functional programming basing on relations since around the 80's [4, 8]. More recent research also argued that the relation is a suitable model to talk about program inversion [17]. In this section we will give a minimal introduction to relations.

A relation of type $A \to B$ is a set of pairs (a, b) where $a \in A$ and $b \in B$. When a pair (a, b) is a member of a relation R, we say that a is mapped to b by R. A (partial) function[1] is a special case of a relation that is *simple* — if $(a, b) \in R$ and $(a, b') \in R$, then $b = b'$. In such cases we can safely denote b by $R\,a$. For example, the function $\mathit{fst} :: (A \times B) \to A$, usually writtten pointwisely as $\mathit{fst}\,(a, b) = a$, is defined by:

$$\mathit{fst} \;=\; \{((a, b), a) \mid a \in A \wedge b \in B\}$$

where (a, b) indeed uniquely determines a. The function $\mathit{snd} :: (A \times B) \to B$ is defined similarly.

The *domain* of $R :: A \to B$, denoted $\mathit{dom}\,R$, is the set $\{(a, a) \in A \mid \exists b \in B :: (a, b) \in R\}$. The *range* of R $(\mathit{ran}\,R)$ is defined symmetrically. The *converse* of a relation R, written R°, is obtained by swapping the pairs in R. That is,

$$(b, a) \in R^\circ \;\equiv\; (a, b) \in R$$

An injective function is one whose converse is also a function. In such cases a value in the domain uniquely defines its image in the range, and vice versa. The term *inverse of a function* is usually reserved to denote the converse of an injective function. Given relations $R :: A \to B$ and $S :: B \to C$, their composition $R; S$ is defined by:

$$R; S \;=\; \{(a, c) \mid (a, b) \in R \wedge (b, c) \in S\}$$

The converse operator $^\circ$ distributes into composition contravariantly:

$$(R; S)^\circ \;=\; S^\circ; R^\circ$$

Given two relations R and S of the same type, one can take their union $R \cup S$ and intersection $R \cap S$. The union, when R and S have disjoint domains, corresponds to conditional branches in a programming language. The intersection is a very powerful mechanism for defining relations. For example, the function $\delta\,a = (a, a)$, which duplicates its argument, can be defined by:

$$\mathit{dup} \;=\; \mathit{fst}^\circ \cap \mathit{snd}^\circ$$

The relation fst° maps a to (a, a') for some arbitrary a'. Similarly, snd° maps a to (a', a) for an arbitrary a'. The only point where they coincide is (a, a). That is, taking their intersection we get

$$\delta \;=\; \{(a, (a, a)) \mid a \in A\}$$

which is what we expect.

The converse operator $^\circ$ distributes into union and intersection. If we take the converse of δ, we get:

$$\delta^\circ \;=\; \mathit{fst} \cap \mathit{snd}$$

Given a pair, *fst* extracts its first component, while *snd* extracts the second. The intersection means that the results have to be equal. That is, δ° takes a pair and let it go through only if the two components are equal. That explains the observation in [11] that to "undo" a duplication, we have to perform an equality test.

2.3 The Point-free Functional Language Fun

The intersection is a very powerful construct for specification — with it one can define undecidable specifications. In this section we will refine the relational constructs to a point-free functional language which is computationally equivalent to conventional programming languages. The syntax of Fun is defined below[2]. We abuse the notation a bit by denoting the set of words $\{w^\circ \mid w \in L\}$ by L°, and by F_V we denote the union of F and the set of variable names V.

$$\begin{array}{lcl} F & ::= & C \mid C^\circ \\ & & \mid F; F \mid \mathit{id} \\ & & \mid \langle F, F\rangle \mid \mathit{fst} \mid \mathit{snd} \\ & & \mid F \cup F \\ & & \mid \mu(V \colon F_V) \\ \\ C & ::= & \mathit{nil} \mid \mathit{cons} \mid \mathit{zero} \mid \mathit{succ} \mid \mathit{inl} \mid \mathit{inr} \end{array}$$

The function $\mathit{nil} :: \mathit{Unit} \to \mathit{List}\,A$ is a constant function always returning the empty list, while $\mathit{cons} :: (A \times \mathit{List}\,A) \to \mathit{List}\,A$ extends a list by the given element. Converses are applied to base constructors (denoted by the non-terminal C) only. The converse of *cons*, for example, decomposes a non-empty list into the head and the tail. The converse of *nil* matches only the empty list and maps it to anything. The result is

[1]For convenience, we refer to possibly partial functions when we say "functions".

[2]In Fun there are no primitive operators for equality or inequality check. They can be defined recursively on natural numbers, so we omitted it from the language to make it simpler. In Inv, however, we do include those checks as primitives since they are important for inversion.

usually thrown away. To avoid non-determinism in the language, we let the range type of nil° be $Unit$. Functions $zero$ and $succ$ are defined similarly. Constructors inl and inr introduces the L and R tags, respectively, while their converses deconstruct the tags.

The function id is the identity function, the unit of composition. Functions fst and snd extract the first and second components of a pair respectively. Those who are familiar with the "squiggle" style of program derivation would feel at home with the "split" construct, defined by $\langle f, g\rangle\, a = (f\ a, g\ a)$. This definition, however, assumes that f and g be functional. Less well-known is its definition in terms of intersection:

$$\langle f, g\rangle \;=\; f; fst^\circ \;\cap\; g; snd^\circ$$

For example, the δ function in the previous section is defined by $\langle id, id\rangle$. The "product" functor $(f \times g)\,(a, b) = (f\ a, g\ b)$ on the other hand, is defined by

$$(f \times g) \;=\; \langle fst; f, snd; g\rangle$$

as all Squigglists know. Note that composition binds tighter than product, therefore $(f; g \times h)$ means $((f; g) \times h)$.

Union of functions is simply defined as set union. To avoid non-determinism, however, we require in $f \cup g$ that f and g have disjoint domains. One way to ensure the disjointness is to always write the union in the form $p_1; f_1 \cup p_2; f_2$ where p_1 and p_2 are combinations of different constructors. Dmain-disjointness in general, however, an interesting problem in itself. For this paper we think of it as a separate concern. We will discuss more about this in Section 3.1 when we talk about unions.

Use of intersection, on the other hand, is restricted to its implicit occurrence in splits.

Finally, μF denotes the unique fixed-point of the Fun-valued function F, with which we can define recursive functions. The important issue whether a relation-valued function has an unique fixed-point shall not be overlooked. It was shown in [10] that the uniqueness of the fixed-point has close relationship with well-foundness and termination. All recursive definitions in this paper do have unique fixed-points, although it is out of the scope of this paper to verify them.

For example, the concatenation of two lists, $cat :: (List\ A \times List\ A) \to List\ A$, usually written in an ordinary functional language as:

$$\begin{aligned} cat\,([\,], y) &= y \\ cat\,(a : x, y) &= a : cat\,(x, y) \end{aligned}$$

can be written in Fun as:

$$\begin{aligned} cat \;=\; \mu(X{:}\ & (nil^\circ \times id); snd \;\cup \\ & (cons^\circ \times id); assocr; (id \times X); cons) \end{aligned}$$

Here $(nil^\circ \times id)$ and $(cons^\circ \times id)$ act as patterns. The term $(cons^\circ \times id)$ decomposes the first component of a pair into its head and tail, while $(nil^\circ \times id)$ checks whether that component is the empty list. The piping function $assocr :: ((A \times B) \times C) \to (A \times (B \times C))$, distributing the values to the right places before the recursive call, is defined by $assocr = \langle fst; fst, \langle fst; snd, snd\rangle\rangle$.

The function $filter$, defined below, takes a list of items wrapped by either L and R, and keeps only those with R tags.

$$\begin{aligned} filter \;=\; \mu(X{:}\ & nil^\circ; nil \;\cup \\ & cons^\circ; (inl^\circ \times id); snd; X \;\cup \\ & cons^\circ; (inr^\circ \times X); cons) \end{aligned}$$

We will come back to it in Section 5.

For some readers, point-free programs may be difficult to read. We decide to stick with a point-free language because its semantics is much simpler and suitable for inversion. The conversion from ordinary pointwise programs to point-free notation can be done mechanically. It is a tedious transformation involving keeping an environment and searching for the best way to rotate the variables, and will not be dealt with in this paper.

The language Fun is not closed under converse — we can define non-injective functions in Fun, whose converses are not functional. In other words, Fun is powerful enough that it allows the programmer to define functions "unhealthy" under converse. In the next section we will further refine Fun into an injective language.

3 The Injective Language Inv

In the previous section we refined the relations into a functional language Fun. In this section, we will further refine Fun into an injective functional language, Inv, that allows only injective functions.

In Section 3.1 we give a definition of Inv. The discussion will be kept in a semi-formal tune, for example, we do not use the semantics function $[\![\_]\!]$ and instead identify the language with its semantics. The reason is that Inv will be further extended to deal with bidirectional updating in Section 4, where a more complete treatment will be given. Some Inv examples are given in Section 3.2, including standard functions on pairs and lists, as well as Inv counterparts of the "lenses" of [14]. In Section 3.3, we briefly discuss about the compilation of Fun into Inv and the computational power of Inv.

3.1 Language Definition

The problematic constructs in Fun include constant functions, fst, snd, and the split. Constant functions and projections lose information. The split duplicates information and, as a result, in inversion we need to take care of consistency of previously copied data. We

$$\begin{array}{ll} (f;g)^{\smile} \;=\; g^{\smile};f^{\smile} & (f \cup g)^{\smile} \;=\; f^{\smile} \cup g^{\smile} \\ (f \times g)^{\smile} \;=\; (f^{\smile} \times g^{\smile}) & (\mu F)^{\smile} \;=\; \mu(X\colon (F\, X^{\smile})^{\smile}) \end{array}$$

Figure 3: The operator $^{\smile}$ distributes into other constructs. In the last equation F is a function from Inv expressions to Inv expressions.

wish to enforce constrained use of these problematic constructs by introducing more structured constructs in Inv, in pretty much the same spirit how we enforced constrained use of intersection by introducing the split in Fun. The language Inv is defined by:

$$\begin{array}{lcl} X & ::= & X^{\smile} \mid \mathit{nil} \mid \mathit{zero} \mid C \\ & & \mid \delta \mid \mathit{dup}\, P \mid \mathit{cmp}\, B \mid \mathit{inl} \mid \mathit{inr} \\ & & \mid X;X \mid \mathit{id} \mid X \cup X \\ & & \mid X \times X \mid \mathit{assocr} \mid \mathit{assocl} \mid \mathit{swap} \\ & & \mid \mu(V\colon X_V) \\ C & ::= & \mathit{succ} \mid \mathit{cons} \mid \mathit{node} \\ B & ::= & < \mid \leq \mid \neq \mid \geq \mid > \\ P & ::= & \mathit{nil} \mid \mathit{zero} \mid \mathit{str}\ \mathit{String} \mid (S;)^{*}\mathit{id} \\ S & ::= & C^{\smile} \mid \mathit{fst} \mid \mathit{snd} \end{array}$$

where $*$ denotes "a possibly empty sequence of". In Inv we use the "reverse" operator $^{\smile}$ to denote inversion. In this section it coincides with relational converse, but will divert from the latter in Section 4. Each constructs in Inv has its reverse in Inv. Constructors *cons* and *succ* have inverses $\mathit{cons}^{\smile}$ and $\mathit{succ}^{\smile}$. The domain of *nil* and *zero* is restricted to the unit type, to preserve injectivity. The function $\mathit{swap}\,(a,b) = (b,a)$ is its own inverse. That is, $\mathit{swap}^{\smile} = \mathit{swap}$. The function *assocr* has inverse *assocl*.

$$\begin{array}{lcl} \mathit{assocr}\,((a,b),c) & = & (a,(b,c)) \\ \mathit{assocl}\,(a,(b,c)) & = & ((a,b),c) \end{array}$$

The reverse operator promotes into composition, product, union and the fixed-point operator, as shown in Figure 3.

An extra restriction needs to be imposed upon union. To preserve reversibility, in $f \cup g$ we require not only the domains, but the ranges of f and g, to be disjoint. Again, one can ensure the disjointness by always writing unions in the form $p_1;f_1;q_1 \cup p_2;f_2;q_2$, where the p_is and q_is are distinct from each other. However, that excludes some useful functions (such as *snoc*, to be introduced in Section 5.1). Some techniques were introduced in [12] to determine the disjointness in a finer granularity. Since this paper is more about a theoretical framework than implementation techniques, we will not deal with automatic determination of range-disjointness.

The range-disjointness restriction implies that we cannot define tail-recursive functions. We will talk about ways to get around it in Section 3.3.

The operator $\delta_A = \{(n,(n,n)) \mid n \in A\}$, where A is the type δ gets instantiated to, makes a copy of its argument. We restrict A to atomic types (integers, strings, and unit) only, and from now on use n and m to denote values of atomic types. To duplicate a list, we can always use $\mathit{map}\,\delta;\mathit{unzip}$, where *map* and *unzip* are to be introduced in Section 3.2.

In many occasions we may want to duplicate not all but some sub-component of the input. For convenience, we include another Inv construct *dup* which takes a sequence of "labels" and duplicates the selected sub-component. The label is either *fst*, *snd*, $\mathit{cons}^{\smile}$, and $\mathit{node}^{\smile}$. Informally, think of the sequence of labels as the composition of selector functions (*fst* and *snd*) or deconstructors, and *dup* can be understood as:

$$\mathit{dup}\, f\, x \quad = \quad (x, f\, x)$$

Formally, it is defined by:

$$\begin{array}{lcl} \mathit{dup}\ \mathit{id} & = & \delta \\ \mathit{dup}\,(\mathit{fst};P) & = & (\mathit{dup}\,P \times \mathit{id});\mathit{subl} \\ \mathit{dup}\,(\mathit{snd};P) & = & (\mathit{id} \times \mathit{dup}\,P);\mathit{assocl} \\ \mathit{dup}\,(\mathit{cons}^{\smile};P) & = & \mathit{cons}^{\smile};\mathit{dup}\,P;(\mathit{cons} \times \mathit{id}) \\ \mathit{dup}\,(\mathit{node}^{\smile};P) & = & \mathit{node}^{\smile};\mathit{dup}\,P;(\mathit{node} \times \mathit{id}) \end{array}$$

where $\mathit{subl}\,((a,b),c) = ((a,c),b)$, whose definition in terms of Inv constructs is given in Section 3.2. For example, $\mathit{dup}\,(\mathit{fst};\mathit{snd})\,((a,n),b) = (((a,n),b),n)$.

The reverse of $\mathit{dup}\, f$ is a partial function taking a pair (x,n) and returning x unchanged if $f\,x$ equals n. Here n can be safely dropped because we know its value already. We write $(\mathit{dup}\,f)^{\smile}$ as $\mathit{eq}\,f$.

$$\mathit{eq}\,f \quad = \quad (\mathit{dup}\,f)^{\smile}$$

Another functionality of *dup* is to introduce constants. The original input is kept unchanged but paired with a new constant[3]:

$$\begin{array}{lcl} \mathit{dup}\ \mathit{nil}\ a & = & (a,[\,]) \\ \mathit{dup}\ \mathit{zero}\ a & = & (a,0) \\ \mathit{dup}\,(\mathit{str}\ s)\ a & = & (a,s) \end{array}$$

Their reverses eliminate a constant whose value is known. In both directions we lose no information.

The *cmp* construct takes a pair of values, and lets them go through only if they satisfy one of the five binary predicates given by non-terminal B.

Apparently every program in Inv is invertible, since no information is lost in any operations. Every operation has its inverse in Inv. The question, then, is can we actually define useful functions in this quite restrictive-looking language?

[3]With *dup* we do not need *nil* and *zero*. For example, $\mathit{nil}^{\smile};\mathit{zero}$ can be written as $\mathit{dup}\ \mathit{zero};\mathit{swap};\mathit{eq}\ \mathit{nil}$. We keep them in the language because they are more familiar for the readers.

3.2 Programming Examples in Inv

To convince the reader that Inv is actually quite useful, it is time to show some example programs.

3.2.1 Functions on Pairs

All functions that move around the components in a pair can be defined in terms of products, *assocr*, *assocl*, and *swap*. We find the following functions useful:

$$\begin{array}{lcl} \mathit{subr} & = & \mathit{assocl}; (\mathit{swap} \times \mathit{id}); \mathit{assocr} \\ \mathit{subl} & = & \mathit{assocr}; (\mathit{id} \times \mathit{swap}); \mathit{assocl} \\ \mathit{trans} & = & \mathit{assocr}; (\mathit{id} \times \mathit{subr}); \mathit{assocl} \end{array}$$

Their semantics, after expanding the definition, is given below:

$$\begin{array}{lcl} \mathit{subr}\,(a,(b,c)) & = & (b,(a,c)) \\ \mathit{subl}\,((a,b),c) & = & ((a,c),b) \\ \mathit{trans}\,((a,b),(c,d)) & = & ((a,c),(b,d)) \end{array}$$

3.2.2 List Processing

The function *map* applies a function to all elements of a list; the function *unzip* takes a list of pairs and splits it into a pair of lists. They can be defined as fixed-points as below:

$$\begin{array}{lcl} \mathit{map}\,f & = & \mu(X: \mathit{nil}^{\smile}; \mathit{nil} \cup \\ & & \quad \mathit{cons}^{\smile}; (f \times X); \mathit{cons}) \\ \mathit{unzip} & = & \mu(X: \mathit{nil}^{\smile}; \delta; (\mathit{nil} \times \mathit{nil}) \cup \\ & & \quad \mathit{cons}^{\smile}; (\mathit{id} \times X); \mathit{trans}; (\mathit{cons} \times \mathit{cons})) \end{array}$$

The branches starting with $\mathit{nil}^{\smile}$ are the base cases, matching $[\,]$, while $\mathit{cons}^{\smile}$ matches non-empty lists. From its semantics we also have $(\mathit{map}\,f)^{\smile} = \mathit{map}\,f^{\smile}$.

List concatenation is not injective. However, the following function *lcat* (labelled concatenation)

$$\mathit{lcat}\,(a,(x,y)) = (a, x +\!\!+ [a] +\!\!+ y)$$

is injective if its domain is restricted to tuples where a does not appear in x. Its definition in Inv can be written as a fixed-point:

$$\begin{array}{lcl} \mathit{lcat} & = & \mu(X: (\delta \times \mathit{swap}; \mathit{eq}\,\mathit{nil}); \mathit{assocr}; (\mathit{id} \times \mathit{cons}) \cup \\ & & \quad (\mathit{id} \times (\mathit{cons}^{\smile} \times \mathit{id}); \mathit{assocr}); \\ & & \quad \mathit{neq}; \mathit{subr}; (\mathit{id} \times X); \mathit{subr}; (\mathit{id} \times \mathit{cons})) \end{array}$$

where $\mathit{neq} = \mathit{assocl}; (\mathit{cmp}\,(\neq) \times \mathit{id}); \mathit{assocr}$.

The function *merge* takes a pair of sorted lists and merges them into one. However, by doing so we lost information necessary to split them back to the original pair. Therefore, we tag the elements in the merged list with labels indicating where they were from. In usual pointwise notation, the function could have been written:

$$\begin{array}{lcl} \mathit{merge}\,(x,[\,]) & = & \mathit{map}\,\mathit{inl}\,x \\ \mathit{merge}\,([\,],y) & = & \mathit{map}\,\mathit{inr}\,y \\ \mathit{merge}\,(a{:}\,x, b{:}\,y) & = & \mathit{inl}\,a : \mathit{merge}\,(x, b{:}\,y), \text{ if } a \leq b \\ & = & \mathit{inr}\,b : \mathit{merge}\,(a{:}\,x, y), \text{ if } a > b \end{array}$$

For example, $\mathit{merge}\,([1,4,7],[2,5,6])$ equals $[L\,1, R\,2, L\,4, R\,5, R\,6, L\,7]$. The above definition is translated to Inv as below:

$$\begin{array}{l} \mu(X: \mathit{eq}\,\mathit{nil}; \mathit{map}\,\mathit{inl} \cup \\ \quad \mathit{swap}; \mathit{eq}\,\mathit{nil}; \mathit{map}\,\mathit{inr} \cup \\ \quad (\mathit{cons}^{\smile} \times \mathit{cons}^{\smile}); \mathit{trans}; \\ \quad\; ((\mathit{leq} \times \mathit{id}); \mathit{assocr}; (\mathit{id} \times \mathit{subr}; (\mathit{id} \times \mathit{cons}); X); \\ \quad\quad (\mathit{inl} \times \mathit{id}) \cup \\ \quad\; (\mathit{gt}; \mathit{swap} \times \mathit{id}); \mathit{assocr}; (\mathit{id} \times \mathit{assocl}; (\mathit{cons} \times \\ \quad\quad \mathit{id}); X); (\mathit{inr} \times \mathit{id})); \mathit{cons}) \end{array}$$

where $\mathit{leq} = \mathit{cmp}\,(\leq)$ and $\mathit{gt} = \mathit{cmp}\,(>)$. Note that we use *eq nil* to test the nullity of the lists.

3.2.3 Lenses

In [14], a set of “lenses” were given to define tree transforms. All the lenses, or their equivalent, can be defined in Inv. A “view” in [14] can be think of as a list of rose trees. Three of the injective lenses can be defined below:

$$\begin{array}{lcl} \mathit{plunge}\,n & = & \mathit{dup}\,(\mathit{str}\,n); \mathit{swap}; \mathit{node} \\ \mathit{hoist}\,n & = & (\mathit{plunge}\,n)^{\smile} \\ \mathit{pivot}\,n & = & \mathit{cons}^{\smile}; (\mathit{hoist}\,n; \mathit{wrap}^{\smile} \times \mathit{id}); \mathit{node} \end{array}$$

where $\mathit{wrap} = \mathit{dup}\,\mathit{nil}; \mathit{cons}$, wrapping an item into a singleton list.

The Inv function *is s* checks whether the input string equals s, while $s \mapsto t$ is a partial function mapping only s to t:

$$\begin{array}{lcl} \mathit{is}\,s & = & \delta; \mathit{eq}\,(\mathit{str}\,s) \\ s \mapsto t & = & \mathit{dup}\,(\mathit{str}\,t); \mathit{swap}; \mathit{eq}\,(\mathit{str}\,s) \end{array}$$

Let f_i be a collection of Inv expressions, and s_i and t_i be a collection of names such that $s_i \neq s_j$ and $t_i \neq t_j$ for $i \neq j$. The expression

$$\mathit{map}\,(\textstyle\bigcup_i (\mathit{node}^{\smile}; (\mathit{is}\,s_i \times f_i); \mathit{node}))$$

takes a list of trees and applies f_i to those with root label s_i. It corresponds to the lense also called *map* in [14], but having a simpler semantics due to its injective nature. On the other hand, the following expression

$$\mathit{map}\,(\textstyle\bigcup_i (\mathit{node}^{\smile}; (s_i \mapsto t_i \times \mathit{id}); \mathit{node}))$$

is the Inv counterpart of the *rename* lense. It takes a list of trees and renames their root labels according to the injective mapping between s_is and t_is.

The lense *xfork* performs filtering and its injective counterpart is bound to return some extra information. We will talk about filters in Section 5.3.

3.3 Logging Translation and the Reversible Turing Machine

Given a transform written in Inv, we will discuss in Section 4 how to assign it an alternative semantics to deal with inconsistencies arise from user editing. To have an effective transform for this purpose, we argue that it is better to code the transform in Inv in the first place. The situation reminds us of the argument in [5] that we need a special language for reversible computers.

In this section, however, we will still briefly talks about compiling Fun programs to Inv programs. We will also discuss the relationship between Inv and reversible Turing machines. Readers who are interested only in the bidirectional updating can skip this section.

Still, there are lots of things we cannot do in Inv. We cannot add two numbers, we cannot concatenate two lists. In short, we cannot construct non-injective functions. However, given any program $f :: A \to B$ in Fun, we can always construct a $f_I :: A \to (B \times H)$ in Inv such that $f_I; fst = f$ — that is, $f\ a = b$ if and only if there exists some h satisfying $f_I\ a = (b, h)$.

Such a f_I may not be unique, but always exists: you can always take $H = A$ and simply copy the input to the output. However, it is not immediately obvious how to construct such a $f_I :: A \to (B \times A)$ in Inv (since not every function can be an argument to *dup*). Nor is it immediately obvious how to compose two transformed functions without throwing away the intermediate result.

In [19], we presented what we call the "logging" translation. It basically works by pairing the result of the computation together with a history, where each choice of branch and each disposed piece of data is recorded, so one can always trace the computation back. It is similar to a translation for procedural languages described in [24].

An important feature missing in Inv is the ability to define loops. In Fun, one can write a loop as a tail recursive function $\mu(X : \mathit{term} \cup \mathit{body}; X)$ where *term* and *body* have disjoint domains. However, the range of $\mathit{body}; X$ contains that of *term*, which is not allowed in Inv — when we ran the loop back-wards we do not know whether to terminate the loop now or execute the body again.

Loops come handy when we want to show that Inv is computationally as powerful as Bennett's reversible Turing machine [7], since the simulation of a Turing machine is best described as a loop. Luckily, we can still (with the logging translation) implement a loop with a counter in Inv. Using a trick invented in [7], we can eliminate the counter as long as the loop body is injective. As a corollary, Inv is computationally equivalent to a reversible Turing machine. More details are given in [19].

4 Bidirectional Updating

Now consider the scenario of an editor, where a source document is transformed, via an Inv program, to a view editable by the user. Consider the transform $\mathit{idx} = \mathit{map}\ (\mathit{dup}\ \mathit{fst}); \mathit{unzip}$, we have:

$$\begin{aligned}&\mathit{idx}\ [(1, \text{“}a\text{”}), (2, \text{“}b\text{”}), (3, \text{“}c\text{”})] = \\ &\quad ([(1, \text{“}a\text{”}), (2, \text{“}b\text{”}), (3, \text{“}c\text{”})], [1, 2, 3])\end{aligned}$$

Think of each pair as a contact and the numbers as names, the function *idx* is a simplified version of the generation of an index of names.

Through a special interface, there are several things the user can do: changing the value of a node, inserting a new node, or deleting a node. Assume that the user changes the value 3 in the "index" to 4:

$$([(1, \text{“}a\text{”}), (2, \text{“}b\text{”}), (3, \text{“}c\text{”})], [1, 2, 4])$$

Now we try to perform the transform backwards. Applying the reverse operator to *idx*, we get $(\mathit{map}\ (\mathit{dup}\ \mathit{fst}); \mathit{unzip})^{\smile} = \mathit{unzip}^{\smile}; \mathit{map}\ (\mathit{eq}\ \mathit{fst})$. Applying it to the modified view, $\mathit{unzip}^{\smile}$ maps the modified view to:

$$[((1, \text{“}a\text{”}), 1), ((2, \text{“}b\text{”}), 2), ((3, \text{“}c\text{”}), 4)]$$

pairing the names and entries together, to be processed by $\mathit{map}\ (\mathit{eq}\ \mathit{fst})$. However, $((3, \text{“}c\text{”}), 4)$ is not in the domain of $\mathit{eq}\ \mathit{fst}$ because the equality check fails. We wish that $\mathit{eq}\ \mathit{fst}$ would return $(4, \text{“}c\text{”})$ in this case, answering the user's wish to change the name.

Now assume that the user inserts a new section title in the table of contents, resulting in

$$([(1, \text{“}a\text{”}), (2, \text{“}b\text{”}), (3, \text{“}c\text{”})], [1, 2, 4, 3])$$

This time the changed view cannot even pass $\mathit{unzip}^{\smile}$, because the two lists have different lengths. We wish that $\mathit{unzip}^{\smile}$ would somehow know that the two 3's should go together and the zipped list should be

$$[((1, \text{“}a\text{”}), 1), ((2, \text{“}b\text{”}), 2), (\perp, 4), ((3, \text{“}c\text{”}), 3)]$$

where $\perp$ denotes some unconstrained value, which would be further constrained by $\mathit{map}\ (\mathit{eq}\ \mathit{fst})$ to $(4, \perp)$. The Inv construct $\mathit{eq}\ \mathit{fst}$ should also recognise $\perp$ and deal with it accordingly.

In short, we allow the programmer to write Inv transforms that are not surjective. Therefore it is very likely that a view modified by the user may fall out of the range of the transform. This is in contrast to the approach taken in [16] where all transforms are bi-total and duplication is not allowed.

The two problems we discussed just now are representative of the view-updating problem. There are basically two kinds of dependencies we have to deal with: element-wise dependency, stating that two pieces of primary-typed data have the same value, and structural dependency, stating that two pieces of data have the same shape. In Inv, element-wise dependency is specified solely by the δ construct, making them easier to deal with. Structural dependency, on the other hand, is specified by the *unzip* function.

One possible solution is to provide an alternative semantics [18] that extends the ranges of Inv constructs in a reasonable way, so that the unmodified, or barely modified programs can deal with the changes. We will discuss this in detail in this section.

4.1 Get and Put

We will need some labels in the view, indicating "this part has been modified by the user." We extend the *View* data type as below:

$$\begin{array}{lcl} \mathit{View} & ::= & \ldots \mid *\mathit{Int} \mid *\mathit{String} \\ \mathit{List}\ a & ::= & \ldots \mid a \oplus \mathit{List}\ a \mid a \ominus \mathit{List}\ a \end{array}$$

Here the $*$ mark applies to atomic types only, indicating that the values has been changed. The view $a \oplus x$ denotes a list $a\colon x$ whose head a was freshly inserted by the user, while $a \ominus x$ denotes a list x which used to have a head a but was deleted. The deleted value a is still cached for future use. The two operators associate to the right, like the cons operator (:). A rose tree is basically a pair consisting of a label and a list of subtrees, and can be dealt with accordingly. It will be a future work to derive the tags given any polymorphic datatype.

The original semantics of each Inv program is an injective function. When the tags are involved, however, we lost the injectivity. Multiple views may be mapped to the same source. In cases where we need to generate fresh new values, an Inv program might not even be a function. The semantic function $[\![\_]\!]$ thus maps X programs to binary relations on *View*s. The semantics of Inv will be discussed in the sections to come, and finally be summarised in Figure 6.

We define two auxiliary functions *notag*? and *ridtag*. The former is a partial function letting through the input view unchanged if it contains no tags. The latter gets rid of the tags in a view, producing a normal form. Their definitions are given in Figure 4. The behaviour of the editor, on the other hand, is specified using two functions get_X and put_X, both parameterised by an Inv program X:

$$\begin{array}{lcl} get_X & = & \mathit{notag}?; [\![X]\!] \\ put_X & \dot{\subseteq} & [\![X^{\smile}]\!]; \mathit{ridtag} \end{array}$$

The function get_X maps the source to the view by calling X. The domain of X, however, is restricted to un-

$$\begin{array}{lcl}
\mathit{notag}?\ x & = & x, \text{ if } \mathit{notag}\ x \\
\mathit{notag}\ (a, b) & = & \mathit{notag}\ a \wedge \mathit{notag}\ b \\
\mathit{notag}\ (a : x) & = & \mathit{notag}\ a \wedge \mathit{notag}\ x \\
\mathit{notag}\ (\mathit{Node}\ a\ x) & = & \mathit{notag}\ a \wedge \mathit{notag}\ x \\
\mathit{notag}\ (L\ a) & = & \mathit{notag}\ a \\
\mathit{notag}\ (R\ a) & = & \mathit{notag}\ a \\
\mathit{notag}\ a & = & \mathit{True} \\
\quad \text{if } a \in \mathit{Int} \cup \mathit{String} \cup \{[\,]\} \cup \{()\} & & \\
\mathit{notag}\ a & = & \mathit{False} \\
 & & \\
\mathit{ridtag}\ *a & = & a \\
\mathit{ridtag}\ (a, b) & = & (\mathit{ridtag}\ a, \mathit{ridtag}\ b) \\
\mathit{ridtag}\ (a\colon b) & = & \mathit{ridtag}\ a\colon \mathit{ridtag}\ b \\
\mathit{ridtag}\ (\mathit{Node}\ a\ x) & = & \mathit{Node}\ (\mathit{ridtag}\ a)\ (\mathit{ridtag}\ x) \\
\mathit{ridtag}\ (L\ a) & = & L\ (\mathit{ridtag}\ a) \\
\mathit{ridtag}\ (R\ a) & = & R\ (\mathit{ridtag}\ a) \\
\mathit{ridtag}\ (a \ominus x) & = & \mathit{ridtag}\ x \\
\mathit{ridtag}\ (a \oplus x) & = & \mathit{ridtag}\ a : \mathit{ridtag}\ x
\end{array}$$

Figure 4: Functions *notag*? and *ridtag*.

tagged views. With this restriction, get_X falls back to the original, injective semantics introduced in Section 3.1, as will be shown in Section 4.4.

The function put_X, on the other hand, maps the (possibly edited) view back to the source by letting it go though $X^{\smile}$ and removing the tags in the result. Here $\dot{\subseteq}$ denotes "functional refinement". It relates a function and a relation, defined by $f \dot{\subseteq} R$ if and only if $f \subseteq R$ and $\mathit{dom}\, f = \mathit{dom}\, R$. In general, $[\![X^{\smile}]\!]; \mathit{ridtag}$ is not a function since $[\![X^{\smile}]\!]$ may leave some values unspecified. However, any functional refinement of $[\![X^{\smile}]\!]; \mathit{ridtag}$ would satisfy the properties we want. The implementation can therefore, for example, choose an "initial value" for each unspecified value according to its type. The initial view is obtained by a call to get_X. When the user performs some editing, the editor applies put_X to the view, obtaining a new source, before generating a new view by calling get_X again.

4.2 Generalised Equality Test

In the original semantics, the $^{\smile}$ operator coincides with relational converse. Therefore, the relation $[\![\delta_A^{\smile}]\!]$, where A is an atomic type, is given by the set $[\![\delta_A]\!]^{\circ} = \{((n, n), n) \mid n \in A\}$. To deal with editing, we generalise its semantics to:

$$\begin{array}{ll} [\![\delta^{\smile}]\!]\,(n, n) = n & [\![\delta^{\smile}]\!]\,(*n, *n) = *n \\ [\![\delta^{\smile}]\!]\,(*n, m) = *n & [\![\delta^{\smile}]\!]\,(m, *n) = *n \end{array}$$

When the two values are not the same but one of them was edited by the user, the edited one gets precedence and goes through. Therefore $(*n, m)$ is mapped to $*n$. If both values are edited, however, they still have to be the same. Note that the semantics of δ does not change. Also, we are still restricted to atomic types. One will have to call $\mathit{map}\,\delta; \mathit{unzip}$ to duplicate a list, thereby separate the value and structural dependency.

4.3 Insertion and Deletion

Recall *unzip* defined in Section 3.2. Its reverse, unzip$^{\smile}$, according to the distributivity of $^{\smile}$, is given by:

$$\begin{array}{rcl} \mathit{zip} & = & \mu(X : (\mathit{nil}^{\smile} \times \mathit{nil}^{\smile}); \delta^{\smile}; \mathit{nil} \cup \\ & & \quad (\mathit{cons}^{\smile} \times \mathit{cons}^{\smile}); \mathit{trans}; (\mathit{id} \times X); \mathit{cons}) \end{array}$$

The puzzle is: how to make it work correctly in the presence of $\ominus$ and $\oplus$ tags? We introduce several new additional operators and types:

- two new Inv operators, *del* and *ins*, both parameterised by a view. The function $\mathit{del}\ a :: \mathit{List}\ A \to \mathit{List}\ A$ introduces an $(a \ominus\ )$ tag, while $\mathit{ins}\ a :: \mathit{List}\ A \to \mathit{List}\ A$ introduces an $(a \oplus\ )$ tag.
- two kinds of pairs in *View*: positive $(a, b)^{+}$ and negative $(a, b)^{-}$. They are merely pairs with an additional label. They can be introduced only by the reverse of $\mathit{fst}_b^{\pm}$ and $\mathit{snd}_a^{\pm}$ functions to be introduced below. The intention is to use them to denote pairs whose components are temporary kept there to be processed. We will see their use in the derivation later.
- six families of functions $\mathit{fst}_a^{\square}$ and $\mathit{snd}_a^{\square}$, where $\square$ can be either $+$, $-$, or nothing, defined by

  $$\begin{array}{rcl} \mathit{fst}_b^{\square}\ (a, b)^{\square} & = & a \\ \mathit{snd}_a^{\square}\ (a, b)^{\square} & = & b \end{array}$$

  That is, fst_b^{+} eliminates the second component of a positive pair only if it equals b. Otherwise it fails. Similarly, snd_a eliminates the first component of an ordinary pair only of it equals a. When interacting with existing operators, they should satisfy the algebraic rules in Figure 5. In order to shorten the presentation, we use $\square$ to match $+$, $-$ and nothing, while $\pm$ matches only $+$ and $-$. The $\square$ and $\pm$ in the same rule must match the same symbol.

With the new operators and types, an extended *zip* capable of dealing with deletion can be extended from the original *zip* by (here "…" denotes the original two branches of *zip*):

$$\begin{array}{l} \mu(X : \ldots \cup \forall a, b \cdot \\ \quad ((\mathit{ins}\ a)^{\smile} \times (\mathit{ins}\ b)^{\smile}); X; \mathit{ins}\ (a, b) \cup \\ \quad ((\mathit{ins}\ a)^{\smile} \times \mathit{isList}); X; \mathit{ins}\ (a, b) \cup \\ \quad (\mathit{isList} \times (\mathit{ins}\ b)^{\smile}); X; \mathit{ins}\ (a, b) \cup \\ \quad ((\mathit{del}\ a)^{\smile} \times (\mathit{del}\ b)^{\smile}); X; \mathit{del}\ (a, b) \cup \\ \quad ((\mathit{del}\ a)^{\smile} \times \mathit{cons}^{\smile}; \mathit{snd}_b^{-}); X; \mathit{del}\ (a, b) \cup \\ \quad (\mathit{cons}^{\smile}; \mathit{snd}_a^{-} \times (\mathit{del}\ b)^{\smile}); X; \mathit{del}\ (a, b)) \end{array}$$

where a and b are universally quantified, and $\mathit{isList} = \mathit{nil}^{\smile}; \mathit{nil} \cup \mathit{cons}^{\smile}; \mathit{cons}$, a subset of *id* letting through only lists having no tag at the head.

$$\begin{array}{rcl} (f \times g); \mathit{fst}_{(g\ b)}^{\square} & = & \mathit{fst}_b^{\square}; f,\ \text{if } g \text{ is total} \\ (f \times g); \mathit{snd}_{(f\ a)}^{\square} & = & \mathit{snd}_a^{\square}; g,\ \text{if } f \text{ is total} \\ \mathit{swap}; \mathit{snd}_a^{\square} & = & \mathit{fst}_a^{\square} \\ \mathit{snd}_a^{\square\,\smile}; \mathit{eq}\ \mathit{nil} & = & (\lambda\,[\,] \to a) \\ \mathit{assocl}; (\mathit{fst}_b^{\square} \times \mathit{id}) & = & (\mathit{id} \times \mathit{snd}_b^{\square}) \\ \mathit{assocl}; (\mathit{snd}_a^{\square} \times \mathit{id}) & = & (\mathit{snd}_a^{\square} \cup \mathit{snd}_a) \\ \mathit{assocl}; \mathit{snd}_{(a,b)}^{\square} & = & \mathit{snd}_a^{\square}; (\mathit{snd}_b^{\square} \cup \mathit{snd}_b) \end{array}$$

Figure 5: Algebraic rules. Here $(\lambda\,[\,] \to a)$ is a function mapping only empty list to a.

Look at the branch starting with $((\mathit{ins}\ a)^{\smile} \times (\mathit{ins}\ b)^{\smile})$. It says that, given a pair of lists both starting with insertion tags $a\oplus$ and $b\oplus$, we should deconstruct it, pass the tails of the lists to the recursive call, and put back an $((a, b)\oplus)$ tag. If only the first of them is tagged (matching the branch starting with $((\mathit{ins}\ a)^{\smile} \times \mathit{isList})$), we temporarily remove the $a\oplus$ tag, recursively process the lists, and put back a tag $(a, b)\oplus$ with a freshly generated b. It is non-deterministic which b is chosen, and might be further constrained when *zip* is further composed with other relations.

The situation is similar with deletion. In the branch starting with $((\mathit{del}\ a)^{\smile} \times \mathit{cons}^{\smile}; \mathit{snd}_b^{-})$ where we encounter a list with an a deleted by the user, we remove an element in the other list and remember its value in b. Here universally quantified b is used to match the value — all the branches with different b's are unioned together, with only one of them resulting in a successful match. After processing it recursively, we add a tag $(a, b)\ominus$ indicating that a pair (a, b) was removed from the resulting list.

It would be very tedious if the programmer has to explicitly write down these extra branches for all functions (let alone that we did not provide the construct for universal quantification.) We wish that *del*, *ins*, *fst* and *snd* do not appear in the programs, but the system can somehow derive the additional branches. Luckily, these additional branches can be derived automatically using the rules in Figure 5.

In the derivations later we will omit the semantics function $[\![\,]\!]$ and use the same notation for the language and its semantics, where no confusion would occur. This is merely for the sake of brevity.

In place of ordinary *cons*, we define two constructs addressing the dependency of structures. Firstly, the *bold* **cons** is defined by::

$$\begin{array}{rcl} \mathbf{cons} & = & \mathit{cons} \cup \\ & & \bigcup_{a::A}(\mathit{snd}_a^{-}; \mathit{del}\ a) \ \cup\ \bigcup_{a::A}(\mathit{snd}_a^{+}; \mathit{ins}\ a) \end{array}$$

Secondly, we define the following *sync* operator:

$$\begin{array}{lcl}
sync & = & (cons \times cons) \\
sync^{\smallsmile} & = & (cons^{\smallsmile} \times cons^{\smallsmile}) \\
\multicolumn{3}{l}{\quad \cup \bigcup_{a,b\in A}(((del\ a)^{\smallsmile}; snd^{-}_{a}{}^{\smallsmile} \times (del\ b)^{\smallsmile}; snd^{-}_{b}{}^{\smallsmile})} \\
\multicolumn{3}{l}{\qquad\qquad\qquad \cup\, ((del\ a)^{\smallsmile}; snd^{-}_{a}{}^{\smallsmile} \times cons^{\smallsmile}; snd_b; snd^{-}_{b}{}^{\smallsmile})} \\
\multicolumn{3}{l}{\qquad\qquad\qquad \cup\, (cons^{\smallsmile}; snd_a; snd^{-}_{a}{}^{\smallsmile} \times (del\ b)^{\smallsmile}; snd^{-}_{b}{}^{\smallsmile}))} \\
\multicolumn{3}{l}{\quad \cup \bigcup_{a,b\in A}(((ins\ a)^{\smallsmile}; snd^{+}_{a}{}^{\smallsmile} \times (ins\ b)^{\smallsmile}; snd^{+}_{b}{}^{\smallsmile})} \\
\multicolumn{3}{l}{\qquad\qquad\qquad \cup\, ((ins\ a)^{\smallsmile}; snd^{+}_{a}{}^{\smallsmile} \times isList; snd^{+}_{b}{}^{\smallsmile})} \\
\multicolumn{3}{l}{\qquad\qquad\qquad \cup\, (isList; snd^{+}_{b}{}^{\smallsmile} \times (ins\ b)^{\smallsmile}; snd^{+}_{b}{}^{\smallsmile}))}
\end{array}$$

In the definition of *zip*, we replace every singular occurence of *cons* with **cons**, and every (*cons* × *cons*) with *sync*. The definition of $sync^{\smallsmile}$ looks very complicated but we will shortly see its use in the derivation. Basically every produce correspond to one case we want to deal with: when both the lists are cons lists, when one or both of them has a ⊖ tag, or when one or both of them has a ⊕ tag.

After the substitution, all the branches can be derived by algebraic reasoning. The rules we need are listed in Figure 5. Only rules for *assocl* are listed. Free identifiers are universally quantified. The rules for *assocr* can be obtained by pre-composing *assocr* to both sides and use $asscor; assocl = id$. To derive the first branch for insertion, for example, we reason:

$$\begin{array}{cl}
 & zip \\
\supseteq & \quad \{\text{fixed-point}\} \\
 & sync^{\smallsmile}; trans; (id \times zip); \mathbf{cons} \\
\supseteq & \quad \{\text{since } sync^{\smallsmile} \supseteq ((ins\ a)^{\smallsmile}; snd^{+}_{a}{}^{\smallsmile} \\
 & \qquad \times (ins\ b)^{\smallsmile}; snd^{+}_{b}{}^{\smallsmile}) \text{ for all } a, b\} \\
 & ((ins\ a)^{\smallsmile} \times (ins\ b)^{\smallsmile}); (snd^{+}_{a}{}^{\smallsmile} \times (ins\ b)^{\smallsmile}); \\
 & trans; (id \times zip); \mathbf{cons} \\
\supseteq & \quad \{\text{claim: } (snd^{+}_{a}{}^{\smallsmile} \times snd^{+}_{b}{}^{\smallsmile}); trans \supseteq (snd^{+}_{(a,b)})^{\smallsmile}\} \\
 & ((ins\ a)^{\smallsmile} \times (ins\ b)^{\smallsmile}); (snd^{+}_{(a,b)})^{\smallsmile}; (id \times zip); \mathbf{cons} \\
= & \quad \{\text{since } (f \times g); snd^{+}_{f\ a} = snd^{+}_{a}; g \text{ for total } f\} \\
 & ((ins\ a)^{\smallsmile} \times (ins\ b)^{\smallsmile}); zip; (snd^{+}_{(a,b)})^{\smallsmile}; \mathbf{cons} \\
\supseteq & \quad \{\text{since } \mathbf{cons} \supseteq snd^{+}_{(a,b)}; ins\,(a, b)\} \\
 & ((ins\ a)^{\smallsmile} \times (ins\ b)^{\smallsmile}); \\
 & zip; (snd^{+}_{(a,b)})^{\smallsmile}; snd^{+}_{(a,b)}; ins\,(a, b) \\
= & \quad \{\text{since } snd^{+}_{x}{}^{\smallsmile}; snd^{+}_{x} = id\} \\
 & ((ins\ a)^{\smallsmile} \times (ins\ b)^{\smallsmile}); zip; ins\,(a, b)
\end{array}$$

We get the first branch. The claim that $trans^{\smallsmile}; (snd^{\square}_{a} \times snd^{\square}_{b}) = snd^{\square}_{(a,b)}$ can be verified by the rules in Figure 5 and is left as an exercise. The use of snd^{+} prevents the head of the expression from being reduced to $(del\,(a, b))^{\smallsmile}$ in the last two steps. As another example, we show how to derive one of the branches for deletion:

$$\begin{array}{cl}
 & sync^{\smallsmile}; trans; (id \times zip); \mathbf{cons} \\
\supseteq & \quad \{\text{since } sync^{\smallsmile} \supseteq ((del\ a)^{\smallsmile}; snd^{-}_{a}{}^{\smallsmile} \times \\
 & \qquad cons^{\smallsmile}; snd_b; snd^{-}_{b}{}^{\smallsmile}) \text{ for all } a, b\} \\
 & ((del\ a)^{\smallsmile}; snd^{-}_{a}{}^{\smallsmile} \times cons^{\smallsmile}; snd_b; snd^{-}_{b}{}^{\smallsmile}); \\
 & trans; (id \times zip); \mathbf{cons} \\
\supseteq & \quad \{\text{claim: } (snd^{+}_{a}{}^{\smallsmile} \times snd^{+}_{b}{}^{\smallsmile}); trans \supseteq snd^{+}_{(a,b)}{}^{\smallsmile}\} \\
 & ((del\ a)^{\smallsmile} \times cons^{\smallsmile}; snd_b); (snd^{-}_{(a,b)})^{\smallsmile}; \\
 & (id \times zip); \mathbf{cons} \\
= & \quad \{\text{since } (f \times g); snd^{-}_{f\ a} = snd^{-}_{a}; g \text{ for total } f\} \\
 & ((del\ a)^{\smallsmile} \times cons^{\smallsmile}; snd_b); zip; (snd^{-}_{(a,b)})^{\smallsmile}; \mathbf{cons} \\
\supseteq & \quad \{\text{since } \mathbf{cons} \supseteq snd^{-}_{(a,b)}; del\,(a, b) \\
 & \qquad \text{and } (snd^{-}_{(a,b)})^{\smallsmile}; snd^{-}_{(a,b)} = id\} \\
 & ((del\ a)^{\smallsmile} \times cons^{\smallsmile}; snd_b); zip; del\,(a, b)
\end{array}$$

In a similar fashion, all the branches can be derived dynamically.

The algebraic rules can be applied both forwards and backwards, which may cause problems for automatic transformation. Luckily, it is possible to integrate these rules in an Inv interpreter. The details are given in [18].

4.4 The Put-Get-Put Property and Galois Connection

A *valid* Inv program is one that does not use $fst^{\square}_{a}$ and $snd^{\square}_{b}$ apart from in **cons** and *sync*. The domain of get_X, for a valid X, is restricted to tag-free views, so is its range. In fact, $notag?; [\![X]\!]$ reduces to the injective function defined by the original semantics. Therefore, $get_X; get_X{}^{\circ} = dom\, get_X$. An outline of the proof is given in Appendix A. Furthermore, $notag?; ridtag = notag?$. As a result, for all valid Inv programs X we have the following *get-put* property:

$$get_X; put_X \quad = \quad dom\, get_X \qquad (1)$$

This is a desired property for our editor: mapping an unedited view back to the source always gives us the same source document.

On the other hand, $put_X; get_X \subseteq id$ is not true. For example, $(put_\delta; get_\delta)\,(*a, b) = (a, a) \neq (*a, b)$. This is one of the main difference between our work and that of [16] and [14], where the *put-get* property $put_X; get_X = id$ is supposed to hold. It also implies that duplication cannot be allowed in the language.

Instead, we have a weaker property. First of all, for all valid X we have $dom\, get_X \subseteq ran\, put_X$. That is, every valid source input to get_X must be a result of put_X for at least one view, namely, the view the source get mapped to under the original semantics. Pre-composing $put\ X$ to (1) and use $put_X; dom\, get_X \subseteq put_X; ran\, put_X = put_X$, we get the following *put-get-put* property:

$$put_X; get_X; put_X \quad \subseteq \quad put_X \qquad (2)$$

$$\begin{aligned}
\llbracket \mathit{nil} \rrbracket\,() &= [\,] \\
\llbracket \mathit{zero} \rrbracket\,() &= 0 \\
\llbracket \mathit{succ} \rrbracket\,n &= n+1 \\
\llbracket \mathit{cons} \rrbracket\,(a,x) &= a\colon x \\
\llbracket \mathit{node} \rrbracket\,(a,x) &= \mathit{Node}\;a\;x \\
\llbracket \mathit{inl} \rrbracket\,a &= L\;a \\
\llbracket \mathit{inr} \rrbracket\,a &= R\;a \\
\llbracket \mathit{id} \rrbracket\,a &= a
\end{aligned}$$

$$\begin{aligned}
\llbracket \mathit{swap} \rrbracket\,(a,b)^{\Box} &= (b,a)^{\Box} \\
\llbracket \mathit{assocr} \rrbracket\,((a,b)^{\pm},c)^{\pm} &= (a,(b,c)^{\pm})^{\pm} \\
\llbracket \mathit{assocr} \rrbracket\,((a,b)^{\pm},c) &= (a,(b,c)^{\pm}) \\
\llbracket \mathit{assocr} \rrbracket\,((a,b),c)^{\pm} &= (a,(b,c))^{\pm} \\
\mathit{assocl} &= \mathit{assocr}^{\smile}
\end{aligned}$$

$$(f^{\smile})^{\smile} = f$$

$$\begin{aligned}
\llbracket \delta \rrbracket\,n &= (n,n) \\
\llbracket \delta^{\smile} \rrbracket\,(n,n)^{\Box} &= n \\
\llbracket \delta^{\smile} \rrbracket\,(*n,*n)^{\Box} &= *n \\
\llbracket \delta^{\smile} \rrbracket\,(*n,m)^{\Box} &= *n \\
\llbracket \delta^{\smile} \rrbracket\,(m,*n)^{\Box} &= *n
\end{aligned}$$

$$\begin{aligned}
\llbracket \mathit{dup}\;\mathit{nil} \rrbracket\,a &= (a,[\,]) \\
\llbracket (\mathit{dup}\;\mathit{nil})^{\smile} \rrbracket\,(a,[\,])^{\Box} &= a \\
\llbracket \mathit{dup}\;\mathit{zero} \rrbracket\,a &= (a,0) \\
\llbracket (\mathit{dup}\;\mathit{zer}0)^{\smile} \rrbracket\,(a,0)^{\Box} &= a \\
\llbracket \mathit{dup}\;(\mathit{str}\;s) \rrbracket\,a &= (a,s) \\
\llbracket (\mathit{dup}\;(\mathit{str}\;s))^{\smile} \rrbracket\,(a,s)^{\Box} &= a
\end{aligned}$$

$$\begin{aligned}
\mathbf{cons} = {} & \mathit{cons} \\
& \cup \textstyle\bigcup_{a::A}(\mathit{snd}_a^{-};\mathit{del}\;a) \\
& \cup \textstyle\bigcup_{a::A}(\mathit{snd}_a^{+};\mathit{ins}\;a)
\end{aligned}$$

$$\begin{aligned}
\llbracket \mathit{cmp}\;\trianglelefteq \rrbracket\,(a,b)^{\Box} &= (a,b)^{\Box}, \text{ if } a \trianglelefteq b \\
\llbracket f;g \rrbracket\,x &= \llbracket g \rrbracket\,(\llbracket f \rrbracket\,x) \\
\llbracket f \times g \rrbracket\,(a,b)^{\Box} &= (\llbracket f \rrbracket\,a, \llbracket g \rrbracket\,b)^{\Box} \\
\llbracket f \cup g \rrbracket &= \llbracket f \rrbracket \cup \llbracket g \rrbracket, \\
&\quad \text{if } \mathit{dom}\,f \cap \mathit{dom}\,g = \mathit{ran}\,f \cap \mathit{ran}\,g = \emptyset \\
\llbracket \mu F \rrbracket &= \llbracket F\;\mu F \rrbracket
\end{aligned}$$

$$\begin{aligned}
\llbracket f^{\smile} \rrbracket &= \llbracket f \rrbracket^{\circ} \\
\llbracket f;g^{\smile} \rrbracket &= \llbracket g^{\smile} \rrbracket;\llbracket f^{\smile} \rrbracket \\
\llbracket (f \times g)^{\smile} \rrbracket &= \llbracket (f^{\smile} \times g^{\smile}) \rrbracket \\
\llbracket (f \cup g)^{\smile} \rrbracket &= \llbracket f^{\smile} \rrbracket \cup \llbracket g^{\smile} \rrbracket \\
\llbracket \mu F^{\smile} \rrbracket &= \llbracket \mu(X \to (F\;X^{\smile})^{\smile} \rrbracket
\end{aligned}$$

$$\begin{aligned}
\llbracket \mathit{fst}_a^{\Box} \rrbracket\,(a,b)^{\Box} &= b \\
\llbracket \mathit{snd}_b^{\Box} \rrbracket\,(a,b)^{\Box} &= a \\
\llbracket \mathit{del}\;a \rrbracket\,(a \ominus x) &= (a,x)^{-} \\
\llbracket \mathit{ins}\;a \rrbracket\,(a \oplus x) &= (a,x)^{+}
\end{aligned}$$

$$\begin{aligned}
\mathit{sync} &= (\mathit{cons} \times \mathit{cons}) \\
\mathit{sync}^{\smile} &= (\mathit{cons}^{\smile} \times \mathit{cons}^{\smile}) \\
\cup \textstyle\bigcup_{a,b\in A} & (((\mathit{del}\;a)^{\smile};\mathit{snd}_a^{-\,\smile} \times (\mathit{del}\;b)^{\smile};\mathit{snd}_b^{-\,\smile}) \\
& \cup ((\mathit{del}\;a)^{\smile};\mathit{snd}_a^{-\,\smile} \times \mathit{cons}^{\smile};\mathit{snd}_b;\mathit{snd}_b^{-\,\smile}) \\
& \cup (\mathit{cons}^{\smile};\mathit{snd}_a;\mathit{snd}_a^{-\,\smile} \times (\mathit{del}\;b)^{\smile};\mathit{snd}_b^{-\,\smile})) \\
\cup \textstyle\bigcup_{a,b\in A} & (((\mathit{ins}\;a)^{\smile};\mathit{snd}_a^{+\,\smile} \times (\mathit{ins}\;b)^{\smile};\mathit{snd}_b^{+\,\smile}) \\
& \cup ((\mathit{ins}\;a)^{\smile};\mathit{snd}_a^{+\,\smile} \times \mathit{isList};\mathit{snd}_b^{+\,\smile}) \\
& \cup (\mathit{isList};\mathit{snd}_b^{+\,\smile} \times (\mathit{ins}\;b)^{\smile};\mathit{snd}_b^{+\,\smile}))
\end{aligned}$$

Figure 6: Summary of the alternative semantics. The patterns should be matched from top to bottom. Some cases of *dup* was given in Section 3.1 and thus omitted.

When the user edits the view, the editor calls the function $put\,X$ to calculate an updated source, and then calls get_X to update the view as well. For example, $(*a, b)$ is changed to (a, a) after $put_\delta; get_\delta$. But now that the view is changed, do we need another application of put_X to update the source again? With the *put-get-put* property we know that another put_X is not necessary, because it is not going to change the view — the effect of $put_X; get_X; put_X$, if it yields anything, is the same as put_X.

It is desirable to have $put_X; get_X; put_X = put_X$. However, this is not true, and $dom\,get_X \neq ran\,put_X$. For a counter-example, take $X = (\delta \times id); assocr; (id \times \delta^{\smile})$. The function get_X takes only pairs with equal components and returns it unchanged. Applying put_X to $(*b, a)$ results in (b, a), which is not in the domain of get_X. Such a result is theoretically not satisfactory, but does not cause a problem for our application. The editor can signal an error to the user, saying that such a modification is not allowed, when the new source is not in the domain of get_X. The domain check is not an extra burden since we have to call get_X anyway.

A Galois connection is a pair of functions $f :: a \to b$ and $g :: a \to b$ satisfying

$$f\,x \trianglelefteq y \quad \equiv \quad x \preceq g\,y \tag{3}$$

Galois connected functions satisfy a number of properties, including $f; g; f = f$. For those X that $dom\,get_X = ran\,put_X$ do hold, get_X and put_X satisfy (3), if we take $\preceq$ to be equality on tag-free *Views* and $\trianglelefteq$ to be $(put_X; get_X)^{\circ}$. That is, $s \preceq s'$ if and only if the two sources s and s' are exactly the same, while a view v is no bigger than v' under $\trianglelefteq$ if there exists a source s such that $v = get_X\,s$ and $s = put\,v'$. For example, (n, n) is no bigger than $(*n, m)$, $(m, *n)$, $(*n, *n)$, and (n, n) itself under $\trianglelefteq$, when the transform is δ. The proof is given in Appendix B. The only glitch here is that $\trianglelefteq$ is not reflexive! In fact it is reflexive only in the range of get_X — the set of tag-free views. However, this is enough for get_X and put_X to satisfy most properties of a Galois connection.

5 Examples of Transforms

In this section we will show more transforms useful for displaying a document in an editor, as well as how they react to user editing.

5.1 Snoc

The function $snoc :: (A \times List\,A) \to List\,A$, appending an element to the end of a list, can be defined recursively as:

$$\begin{aligned}
snoc \;=\; & \mu(X : eq\;nil; dup\;nil; \mathbf{cons} \cup \\
& \qquad (id \times \mathbf{cons}^{\circ}); subr; (id \times X); \mathbf{cons})
\end{aligned}$$

For example $[\![\mathit{snoc}]\!]\,(4,[1,2,3]) = [1,2,3,4]$. Conversely, $\mathit{snoc}^{\smile}$ extracts the last element of a list. But what is the result of extracting the last element of a list whose last element was just removed? We expand the base case:

$$
\begin{array}{ll}
 & \mathit{snoc}^{\smile} \\
\supseteq & \quad \{\text{fixed-point}\} \\
 & \mathbf{cons}^{\smile}; \mathit{eq\ nil}; \mathit{dup\ nil} \\
\supseteq & \quad \{\text{specialising } \mathbf{cons} \supseteq \mathit{snd}^{-}_{a}; \mathit{del\ a}\} \\
 & (\mathit{del\ a})^{\smile}; \mathit{snd}^{-\smile}_{a}; \mathit{ea\ nil}; \mathit{dup\ nil} \\
= & \quad \{\text{since } \mathit{snd}^{-\smile}_{a}; \mathit{eq\ nil} = (\lambda[\,] \to a)\} \\
 & (\mathit{del\ a})^{\smile}; (\lambda[\,] \to a); \mathit{dup\ nil} \\
= & \quad \{\text{since } \mathit{snd}^{-\smile}_{a}; \mathit{eq\ nil} = (\lambda[\,] \to a) \Rightarrow \mathit{snd}^{-\smile}_{a} = \\
 & \qquad (\lambda[\,] \to a); \mathit{dup\ nil}\} \\
 & (\mathit{del\ a})^{\smile}; \mathit{snd}^{-\smile}_{a}
\end{array}
$$

That is, for example, $\mathit{snoc}^{\smile}\,(4 \ominus [\,]) = \mathit{snd}^{-\smile}_{4}\,[\,]$. Inductively, we have $\mathit{eval}\,\mathit{snoc}^{\smile}\,(1:2:3:4 \ominus [\,]) = \mathit{snd}^{-\smile}_{4}\,(1:2:3:[\,])$, which is reasonable enough: by extracting the last element of a list whose last element, 4, is missing, we get a pair whose first element should not have been there.

5.2 Reverting a List

The ubiquitous fold function on lists can be defined by

$$\mathit{fold}\,f\,g \;=\; \mu(X : \mathit{nil}^{\smile}; g \;\cup\; \mathbf{cons}^{\smile}; (\mathit{id} \times X); f)$$

The function *reverse*, reverting a list, can be defined in terms of fold as $\mathit{reverse} = \mathit{fold\ snoc\ nil}$. Unfolding its definition, we can perform the following refinement:

$$
\begin{array}{ll}
 & \mathit{reverse}^{\smile} \\
\supseteq & \quad \{\text{unfolding the definitions}\} \\
 & \mathit{snoc}^{\smile}; (\mathit{id} \times \mathit{reverse}^{\smile}); \mathbf{cons} \\
\supseteq & \quad \{\text{since } \mathit{snoc}^{\smile} \supseteq (\mathit{del\ a})^{\smile}; \mathit{snd}^{-\smile}_{a}\} \\
 & (\mathit{del\ a})^{\smile}; \mathit{snd}^{-\smile}_{a}; (\mathit{id} \times \mathit{reverse}^{\smile}); \mathbf{cons} \\
= & \quad \{\text{since } (f \times g); \mathit{snd}^{-}_{f\ a} = \mathit{snd}^{-}_{a}; g \text{ for total } f\} \\
 & (\mathit{del\ a})^{\smile}; \mathit{reverse}^{\smile}; \mathit{snd}^{-\smile}_{a}; \mathbf{cons} \\
\supseteq & \quad \{\mathbf{cons} \supseteq \mathit{snd}^{-}_{a}; \mathit{del\ a} \text{ and } \mathit{snd}^{-\smile}_{a}; \mathit{snd}^{-}_{a} = \mathit{id}\} \\
 & (\mathit{del\ a})^{\smile}; \mathit{reverse}^{\smile}; \mathit{del\ a}
\end{array}
$$

which shows that $\mathit{reverse}^{\smile}$ regenerates the $\ominus$ tags (and, similiarly, $\oplus$ tags) upon receipt of the "partial" pairs returned by *snoc*. For example, we have $\mathit{reverse}\,(1:2:3 \ominus 4:[\,]) = 4:3 \ominus 2:1:[\,]$ which is exactly what we want. A lesson is that to deal with lists, we have to first learn to deal with pairs.

5.3 Filtering

Filtering is an often needed feature. For example, in a list of $(\mathit{author}, \mathit{article})$ pairs we may want to extract the articles by a chosen author. If we apply the logging transform to the function *filter* defined in Section 2.3, we get the following Inv program[4]:

$$
\begin{array}{lll}
\mathit{split} & = & \mu(X\colon \mathit{nil}^{\circ}; \delta; (\mathit{nil} \times \mathit{nil}) \cup \\
 & & \quad \mathit{cons}^{\circ}; (\mathit{inl}^{\circ}; \mathit{inl} \times X); \mathit{subr}; (\mathit{id} \times \mathit{cons}) \cup \\
 & & \quad \mathit{cons}^{\circ}; (\mathit{inr}^{\circ}; \mathit{dup\ unit} \times X); \mathit{trans}; \\
 & & \quad\; (\mathit{id} \times (\mathit{inr} \times \mathit{id})); (\mathit{cons} \times \mathit{cons}))
\end{array}
$$

The function *split* returns a pair of lists, the first one being the filtered list. The second list, supposed to be hidden from the user, records the removed element, as well as holes for elements of the first list. For example, $\mathit{split}\,[R\,1, L\,2, R\,3]$ yields $([1,3], [R\,(), L\,2, R\,()])$. Simply replacing $(\mathit{cons} \times \mathit{cons})$ by *sync* synchronises the filtered list with the list of holes.

One shall not, however, expect the logging transformation to always yield a *useful* injective function from an arbitrary non-injective one. Transforming the function *cat*, for example, yields a function returning the concatenated list together with a history isomorphic to the length of the first list. This is a reasonable translation, but for displaying a document it is not as useful as, for example, the *lcat* function defined in Section 3.2. Naively transforming the Fun expression $\langle \mathit{map\,fst}, \mathit{id} \rangle$ yields an Inv program that works but produces a redundant history we do not actually need.

We believe that it is another evidence for the argument in [5] that transforming an ordinary program to an information-preserving and yet usable one is hard, and we need a language designed for reversible programming. Nevertheless, it is a challenging topic to see how well we can compile and optimise Fun programs to Inv.

5.4 Merging

Recall the function *merge* defined in Section 3.2, merging two sorted lists into one, while marking the elements with labels remembering where they were from:

$$\mathit{merge}\,([1,4,7],[2,5,6]) \;=\; [L\,1, R\,2, L\,4, R\,5, R\,6, L\,7]$$

A common scenario of filtering is when we have a list of *sorted* items to filter. For example, the articles in the database may be sorted by the date of creation, and splitting the list retains the order. If we simplify the situation a bit further, it is exactly the converse of what *merge* does, if we think of L and R as true and false!

To make *merge* work with editing tags, we simply

[4] In fact the resulting history is not exactly a list but a recursive datatype isomorphic to the list of sums. The operator *sync* can be made to deal with arbitrary datatype rather than just lists, but that is out of the scope of this paper.

replace every occurrence of *cons* with **cons**:

$$\begin{array}{l} \mathit{merge} = \\ \mu(X : \mathit{eq}\ \mathit{nil}; \mathit{map}\ \mathit{inl} \cup \\ \quad \mathit{swap}; \mathit{eq}\ \mathit{nil}; \mathit{map}\ \mathit{inr} \cup \\ \quad (\mathbf{cons}^{\smile} \times \mathbf{cons}^{\smile}); \mathit{trans}; \\ \quad\ ((\mathit{leq} \times \mathit{id}); \mathit{assocr}; (\mathit{id} \times \mathit{subr}; (\mathit{id} \times \mathbf{cons}); X); \\ \quad\quad (\mathit{inl} \times \mathit{id}) \cup \\ \quad\quad (\mathit{gt}; \mathit{swap} \times \mathit{id}); \mathit{assocr}; (\mathit{id} \times \mathit{assocl}; (\mathbf{cons} \times \\ \quad\quad\ \mathit{id}); X); (\mathit{inr} \times \mathit{id})); \mathbf{cons}) \end{array}$$

Notice that we did not use *sync* because, in this case, we certainly do not want to delete or invent elements in one list when the user edits the other! This is an example showing that changing *cons* to **cons** and *sync* cannot be done entirely mechanically without knowing the intention of the programmer. The function *merge* above deals with editing tags as how we would expect. For example, when an element is added to the split list:

$$\begin{array}{l} \mathit{merge}\,(1 : 3 \oplus 4 : 7 : [\,], [2, 5, 6]) \ = \\ \quad L\,1 : R\,2 : L\,3 \oplus [L\,4, R\,5, R\,6, L\,7] \end{array}$$

the new element is inserted back to the original list as well.

6 Related Work

Bidirectional Updating

Bidirectional view-updating: to correctly reflect the modification on the view back to the database [6, 9, 13, 20, 1], is an old problem in the database community. It has recently attracted many interests [16, 14], each took a slightly different approach according to their target applications.

In [14], a semantic foundation and a programming language (the "lenses") for bidirectional transformations are given. They form the core of the data synchronisation system Harmony [21]. Another very much related language was given by Meertens [16] to specify constraints in the design of user-interfaces. Due to their intended applications, not much efforts were put on describing either element-wise or structural dependency inside the view.

Complementarily, we choose to based our formalisation of bidirectional updating on injective mapping. The function *put* computes a source solely from the view, which contributes to the simplicity of its semantics. The extension to deal with duplication and structural changes are thus easier to cope with.

An earlier attempt [15] of extending the languages in [16] and [14] for presentation-oriented editors was not satisfactory, and we believe that it is because the languages do not reveal enough structure of the computation. Still, the programmer may feel more at ease coding in a non-injective languages. One of our future work would be to investigate whether these two approaches can be reconciled.

Program Inversion

Previous researches [2, 11, 17] showed that program inversion is not only helpful for manual algorithm derivation but also for solving practical problems. This work demonstrates yet another application.

The pair of operators *dup* and *eq* in Inv were inspired by [11], where it was pointed out that duplication and equality check are inverses of each other. Our injective language Inv is an extension of that in [19] with a new semantics for dealing with duplications and structural changes. It is basically the same as that in [18], though the focus there was on algebraic reasoning of bidirectional behavior rather than on a new framework for bidirectional updating as in this paper. We make this clearer in Section 3 that Inv is expressive enough to specify those bidirectional transformations in [16, 14].

Presentation-oriented Editors

The original motivation of this work was to build a theoretical foundation for presentation-oriented editors supporting interactive development of XML documents [3, 23, 22]. Proxima [22] is a presentation-oriented generic editor, to which one can "plug-in" their own editors for different types of documents and representations. However, it requires explicit specification of both forward and backward updating. Our goal is to specify only the forward transform and derive the backward updating automatically, at the cost of introducing editing tags. We plan to integrate the result to [23] and [22].

7 Conclusion

To deal with the bidirectional updating problem — maintaining the consistency of two pieces of data related by a transform X, we propose to describe X in our injective language Inv and decorate the program with **cons** and *sync* constructs when necessary. The Inv program is then given an alternative, relational semantics capable of dealing with editing actions. The behaviour of such programs can be derived by algebraic reasoning. The use of an injective language made the semantics much simpler, while making its properties, limitation, and possibly ways to overcome the limitation, much easier exposed. Therefore we believe that it is a suitable theoretical model for such tasks.

The language Inv is capable of describing most transformations one would ever need to display a structured document. In fact, Inv is computationally equivalent to the reversible Turing machine. Whatever one can do in an ordinary functional language, there is a hand-coded Inv program doing the same, possibly returning some history.

We argue that ordinary functional programs do not reveal enough structure of the computation, therefore it is not easy to adapt such programs to bidirectional updating. There is a mechanical way to compile all non-injective functions into Inv. How to optimise the compiled program, on the other hand, still remains a challenging task.

The main advantage of our approach is being able to deal with duplication and structural changes – at the expense of introducing extra tags. This is acceptable for our target application, an editor for structured documents, and in contrast to [16, 14], whose target applications, such as bookmark synchronisation, do not allow extra tags. The lack of which causes difficulty for a proper treatment of duplication and structural changes.

Acknowledgements

The idea of using algebraic rules and program transformation to guide the processing of editing tags was proposed by Lambert Meertens during the first author's visit to Kestrel Institute, CA. The authors would like to thank the members of the Programmable Structured Document in Information Processing Lab, Tokyo University for valuable discussions. This research is partly supported by the e-Society Infrastructure Project of the Ministry of Education, Culture, Sports, Science and Technology, Japan.

References

[1] S. Abiteboul. On views and XML. In *Proceedings of the 18th ACM SIGPLAN-SIGACT-SIGART Symposium on Principles of Database Systems*, pages 1–9. ACM Press, 1999.

[2] S. M. Abramov and R. Glück. The universal resolving algorithm and its correctness: inverse computation in a functional language. *Science of Computer Programming*, 43:193–299, May-June 2002.

[3] Altova Co. Xmlspy. `http://www.xmlspy.com/products_ide.html`.

[4] R. C. Backhouse, P. de Bruin, G. Malcolm, E. Voermans, and J. van der Woude. Relational catamorphisms. In B. Möller, editor, *Proceedings of the IFIP TC2/WG2.1 Working Conference on Constructing Programs*, pages 287–318. Elsevier Science Publishers, 1991.

[5] H. G. Baker. NREVERSAL of fortune–the thermodynamics of garbage collection. In *Proc. Int'l Workshop on Memory Mgmt*, number 637 in Lecture Notes in Computer Science, St. Malo, France, September 1992.

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

[7] C. H. Bennett. Logical reversibility of computation. *IBM Journal of Research and Development*, 17(6):525–532, 1973.

[8] R. S. Bird and O. de Moor. *Algebra of Programming*. International Series in Computer Science. Prentice Hall, 1997.

[9] U. Dayal and P. A. Bernstein. On the correct translation of update operations on relational views. *ACM Transactions on Database Systems*, 7(3):381–416, September 1982.

[10] H. Doornbos and R. C. Backhouse. Reductivity. *Science of Computer Programming*, 26:217–236, 1996.

[11] R. Glück and M. Kawabe. A program inverter for a functional language with equality and constructors. In A. Ohori, editor, *Programming Languages and Systems. Proceedings*, number 2895 in Lecture Notes in Computer Science, pages 246–264. Springer-Verlag, 2003.

[12] R. Glück and M. Kawabe. Derivation of deterministic inverse programs based on LR parsing (extended abstract). In Y. Kameyama and P. J. Stuckey, editors, *Proceedings of Functional and Logic Programming*, number 2998 in Lecture Notes in Computer Science, pages 291–306, Nara, Japan, 2004. Springer-Verlag.

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

[14] M. B. Greenwald, J. T. Moore, B. C. Pierce, and A. Schmitt. A language for bi-directional tree transformations. Technical Report, MS-CIS-03-08, University of Pennsylvania, August 2003.

[15] Z. Hu, S.-C. Mu, and M. Takeichi. A programmable editor for developing structured documents based on bidirectional transformations. In *Proceedings of ACM SIGPLAN 2004 Symposium on Partial Evaluation and Program Manipulation*, Verona, Italy, August 2004. ACM Press.

[16] L. Meertens. Designing constraint maintainers for user interaction. `ftp://ftp.kestrel.edu/pub/papers/meertens/dcm.ps`, 1998.

[17] S.-C. Mu and R. S. Bird. Inverting functions as folds. In E. Boiten and B. Möller, editors, *Sixth International Conference on Mathematics of Program Construction*, number 2386 in Lecture Notes in Computer Science, pages 209–232. Springer-Verlag, July 2002.

[18] S.-C. Mu, Z. Hu, and M. Takeichi. An algebraic approach to bi-directional updating, 2004. Submitted to Asian Programming Languages and Systems.

[19] S.-C. Mu, Z. Hu, and M. Takeichi. An injective language for reversible computation. In *Seventh International Conference on Mathematics of Program Construction*. Springer-Verlag, July 2004.

[20] A. Ohori and K. Tajima. A polymorphic calculus for views and object sharing. In *Proceedings of the 13th ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems*, pages 255–266. ACM Press, 1994.

[21] B. C. Pierce, A. Schmitt, and M. B. Greenwald. Bringing harmony to optimism: an experiment in synchronizing heterogeneous tree-structured data. Technical Report, MS-CIS-03-42, University of Pennsylvania, March 18, 2004.

[22] M. M. Schrage. *Proxima - A presentation-oriented editor for structured documents*. PhD thesis, Utrecht University, The Netherlands, 2004.

[23] M. Takeichi, Z. Hu, K. Kakehi, Y. Hayashi, S.-C. Mu, and K. Nakano. TreeCalc:towards programmable structured documents. In *The 20th Conference of Japan Society for Software Science and Technology*, September 2003.

[24] P. Zuliani. Logical reversibility. *IBM Journal of Research and Development*, 46(6):807–818, 2001. Available online at `http://www.research.ibm.com/journal/rd45-6.html`.

A Proof for Injectivity of get_X

The aim is to prove that $get_X ; get_X{}^\circ = dom\ get_X$ for all valid X. We will merely sketch an outline here. First of all we need this lemma:

$$notag?; [\![X]\!] \quad = \quad notag?; [\![X]\!]; notag? \qquad (4)$$

meaning that $[\![X]\!]$ always map tag-free views to tag-free views. The proof is a routine inductive proof. It also takes merely a routine inductive proof to show $get_X ; get_X{}^\circ = dom\ get_X$. We show only one of the cases.

$$\begin{array}{ll}
 & get_{X;Y} \\
= & \quad \{\text{definition}\} \\
 & notag?; [\![X]\!]; [\![Y]\!]; [\![Y]\!]^\circ; [\![X]\!]^\circ; notag? \\
= & \quad \{(4),\ notag?^\circ = notag?\} \\
 & notag?; [\![X]\!]; notag?[\![Y]\!]; [\![Y]\!]^\circ; notag?; [\![X]\!]^\circ; notag? \\
= & \quad \{\text{induction}\} \\
 & notag?; [\![X]\!]; dom\ get_Y ; [\![X]\!]^\circ; notag? \\
= & \quad \{\text{domain calculus}\} \\
 & dom\ (notag?; [\![X]\!]; get_Y); notag?; [\![X]\!]; [\![X]\!]^\circ; notag? \\
= & \quad \{\text{induction}\} \\
 & dom\ (get_X ; get_Y); dom\ get_X \\
= & \quad \{\text{domain calculus}\} \\
 & dom\ (get_X ; get_Y) \\
= & \quad \{\text{definition}\} \\
 & dom\ get_{X;Y}
\end{array}$$

B Proof for Galois Connectivity

Let $\preceq$ be equality on tag-free views. To derive $\trianglelefteq$ we reason:

$$\begin{array}{ll}
 & x = put_X\ y \\
\equiv & \quad \{\text{set-theoretical notation}\} \\
 & (y, x) \in put_X \\
\equiv & \quad \{\text{since } ran\ put_X = dom\ get_X = get_X ; get_X{}^\circ\} \\
 & (y, x) \in put_X ; get_X ; get^\circ \\
\equiv & \quad \{\text{set theory, since } get_X \text{ is a function}\} \\
 & (y, get_X\ x) \in put_X ; get_X \\
\equiv & \quad \{\text{let } \trianglelefteq = (put_X ; get_X)^\circ\} \\
 & get_X\ x \trianglelefteq y
\end{array}$$

In other words, get_X and put_X satisfy (3) if we take $\trianglelefteq$ to be $(put_X ; get_X)^\circ$. Due to the notation we choose, when we write a relation in infix position, the one on the left-hand side is the input, which is different from some existing literature. In the conventional notation, we would say that we choose $\trianglelefteq$ to be *put*; *get*.