Higher-Order and Symbolic Computation, 21, 89–122 (2008)
© 2008 Kluwer Academic Publishers, Boston. Manufactured in The Netherlands.

A Programmable Editor for Developing Structured Documents Based on Bidirectional Transformations

ZHENJIANG HU hu@nii.ac.jp
Information Systems Architecture Research Division/GRACE Center
National Institute of Informatics
2-1-2 Hitotsubashi, Chiyoda-ku, Tokyo 101-8430, Japan

SHIN-CHENG MU scm@iis.sinica.edu.tw
Institute of Information Science, Academia Sinica
No 128, Academia Road, Section 2, Nankang, Taipei 11529, Taiwan

MASATO TAKEICHI takeichi@mist.i.u-tokyo.ac.jp
Department of Mathematical Informatics, University of Tokyo
7-3-1 Hongo, Bunkyo, Tokyo 113-8656, Japan

Editor: Nevin Heintze, Julia Lawall, Michael Leuschel, Peter Sestoft

Abstract. This paper presents an application of bidirectional transformation to the design and implementation of a novel editor supporting interactive refinement in the development of structured documents. The user performs a sequence of editing operations on a view of the document, and the editor automatically derives an efficient and reliable document source and a transformation that produces the document view. The editor is unique in its programmability, in the sense that transformations can be obtained through editing operations. It uses the view-updating technique developed in the database community and a new bidirectional transformation language that can describe not only the relationship between the document source and its view, but also the data dependency in the view.

Keywords: View updating, Bidirectional transformation, Functional programming, Document Engineering, Structured Editor

1. Introduction

XML [4] has been attracting a tremendous amount of interest as a universal, queryable representation for structured documents. This has in part been stimulated by the growth of the Web and e-commerce, where XML has emerged as the *de facto* standard for structured document representation and information interchange. Many XML editors [22] support the construction of XML documents. While existing XML editors are helpful for *creating* structured documents, they are rather weak in supporting dynamic *refinement* of the documents.

A structured document consists of three components: a schema for defining the document structure, the source data conforming to the schema, and a transformation for displaying the data. Take the document in Figure 1, for example. We start by defining an address book schema (the DTD in the figure), which allows an arbitrary number of entries, each including a name, zero or more email addresses,

and a telephone number. We then construct an XML document conforming to the schema to store address information. Also, we define a transformation (the XSLT program in the figure) to display the address book in a user-friendly *view* (Figure 2), say by sorting the entries according to the names and adding an index of names. In this example, the view is also described as an XML document. In general, the view can be described in HTML or any data structure that can be understood and displayed by a browser. Note that the source XML document in Figure 1 has no redundant information, while the view in Figure 2 does; i.e. each name appears twice.

One would hope that the development of a structured document would follow a perfectly rational process — first design the schema, then prepare the data, and design the transformation. In reality, however, all three components may keep evolving. While preparing the data, one may realise that the schema is too restrictive; changing the schema may make the transformation invalid. It would be nice to have an interactive environment in which all three components can be refined simultaneously, and the system would somehow maintain their consistency. However, existing editors do not support such interactive refinement very well. The three components of a structured document are usually maintained independently with no guarantee of consistency. A big challenge is, therefore, to find an efficient way to maintain consistency of the source document and its view even when there is local data dependency in the view. Considering the view in Figure 2, we would hope that when the user adds or deletes a person, for example, the original document in Figure 1 would be updated correspondingly. Furthermore, the changes should also trigger an update of the index of names in Figure 2. We may even hope that when a name is added to the index, an empty person entry is added to the `person` bodies in both the source document and the view.

Another problem is that the XML and the XSLT transformations are usually hand-written. To be able to design the XML and the XSLT transformation, the users are expected to be capable of programming. This is in contrast to applications such as spreadsheets, in which the user can manipulate the spreadsheet by using an intuitive interface. The user is not aware that it is actually a limited form of programming which assumes little knowledge about types, control flow, etc. Instead of writing code, the user inserts functions through an interface. The ease of use of spreadsheets contributes significantly to their popularity.

In this paper, we propose a novel editor that supports interactive refinement during the development of structured documents: Given a sequence of editing operations on the *view*, both the source document and the transformation from the source document to the view can be automatically derived. Furthermore, we show that if the structure (schema) of the view is given, the editor can systematically obtain the structure (schema) of the source document from the transformation.

The key to our editor is a bidirectional transformation language that describes the relationship between the source data and the view. Our main contributions can be summarized as follows.

- We, as far as we know, are the first to recognize the importance of the view-updating technique for interactive development of structured documents. The

- DTD:

```
<!ELEMENT Addrbook (Person*)>
<!ELEMENT Person (Name, Email*, Tel)>
<!ELEMENT Name (#PCDATA)>
<!ELEMENT Email (#PCDATA)>
<!ELEMENT Tel (#PCDATA)>
```

- XML

```
<Addrbook>
  <Person>
    <Name>Takeichi</Name>
    <Email>takeichi@acm.org</Email>
    <Tel>+81-3-5841-7430</Tel>
  </Person>
  <Person>
    <Name>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>Mu</Name>
    <Email>scm@iis.sinica.edu.tw</Email>
    <Tel>+81-3-5841-7411</Tel>
  </Person>
</Addrbook>
```

- XSLT:

```
<xsl:template match="/Addrbook">
  <IPL>
    <Index>
      <xsl:for-each select="Person">
        <xsl:sort select="Name"/>
        <xsl:copy-of select="Name"/>
      </xsl:for-each>
    </Index>
    <Addrbook>
      <xsl:for-each select="Person">
        <xsl:sort select="Name"/>
        <xsl:copy-of select="."/>
      </xsl:for-each>
    </Addrbook>
  </IPL>
</xsl:template>
```

Figure 1. Example Structured Document (an Address Book)

```
<IPL>
  <Index>
    <Name>Hu</Name>
    <Name>Mu</Name>
    <Name>Takeichi</Name>
  </Index>
  <Addrbook>
    <Person>
      <Name>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>Mu</Name>
      <Email>scm@iis.sinica.edu.tw</Email>
      <Tel>+81-3-5841-7411</Tel>
    </Person>
    <Person>
      <Name>Takeichi</Name>
      <Email>takeichi@acm.org</Email>
      <Tel>+81-3-5841-7430</Tel>
    </Person>
  </Addrbook>
</IPL>
```

Figure 2. A View of the Address Book in XML

view-updating technique, i.e. reflecting view modification back to the original database [1, 2, 6, 11, 19], has been intensively studied in the database community. We have borrowed this technique and adapted it to the design of our editor with an extension not exploited before: editing operations can modify not only the view but also the transformation (from the database to the view). This adapted view updating technique, which departs from traditional formulations of view update, plays an important role in both design and implementation of our editor.

- We designed a programming language, X, for specifying the relationship between the original data and the view. It is inspired by the "lenses" of Greenwald et al. [12] and the language developed by Meertens for constraint maintenance [15]. Both languages, due to their application area, focused their attention on total and surjective relations between the data and the view. We included in X a special construct to duplicate data explicitly for handling data dependency within the view as well as handling non-surjective relations. The language

is powerful enough to describe the common editing operations (insert, delete, move, and copy) as well as other important transformations.

- We have implemented our idea in a prototype editor. The editor is particularly interesting in its programmability, consistency, and has a unified, presentation-oriented (i.e., view-oriented) interface for developing the three components through editing operations on the view.

  – *Programmability*: Transformation programs can be constructed (automatically derived) through interactive editing operations on the view.

  – *Consistency*: The bidirectional language ensures consistency among the source document, the transformation, and the view.

  – *Presentation-oriented* interface: The editor has a uniform view-based editing interface that enables users to easily construct and refine documents.

The rest of the paper is organized as follows. We start by giving a simple definition of structured documents in functional notation in Section 2. After defining the bidirectional transformation language, which plays an important role in our editor, in Section 3, we describe the design principle and implementation techniques in Section 4 and demonstrate how our system can assist the development of structured documents in Section 5. Related work is discussed in Section 6. We conclude in Section 7 with a brief summary and a look at future work.

2. Structured Documents

We formulate a *structured document* as a triple (T, D, X):

- T: schema for defining the structure of source document;
- D: source document;
- X: transformation mapping a source document to a displayed document, which is called *view*.

For instance, the structured document described in the introduction specifies T using a DTD, D using XML, and X using XSLT.

In this paper, for the sake of conciseness, we introduce a more lightweight notation for schemas and documents. For the transformation, we have developed our own language, X, which we describe in detail in Section 3. Throughout this paper, we use Haskell-like [3] notation.

We shall briefly discuss document representation in this section, and defer the discussion of schemas to Section 4. A structured document is a tree, which is viewed as a pair of a label and a list of its subtrees. The *Val* datatype defines the datatypes we will see in the sections to follow. Among them, a document is

represented by *Tree Val*.

$$
\begin{array}{lcl}
Val & ::= & Atom \\
 & & \mid\ *Atom \ \mid\ Val^{+} \ \mid\ Val^{-} \\
 & & \mid\ (Val \times Val) \ \mid\ [Val] \ \mid\ Tree\ Val \\
Atom & ::= & \mathsf{String} \ \mid\ () \\
[a] & ::= & [\,] \ \mid\ a : [a] \\
Tree\ a & ::= & \mathsf{N}\ a\ [Tree\ a]
\end{array}
$$

The atomic types include string and unit, which has only one value (). The *editing tags* $(*(_)$, $(_)^{+}$, and $(_)^{-})$ in the second line of the definition of *Val* are marks the editor leaves in the tree after the user edits something. They are discussed in Section 3.3. Composite types include pairs, lists, and trees. The (:) operator, forming lists, associates to the right. We also follow the common convention of writing the list `"1"` : `"2"` : `"3"` : [ ] as [`"1"`, `"2"`, `"3"`]. Trees are constructed by the constructor N, which builds a new tree from a node and a list of trees.

For example, the document source in the introduction is represented as follows.

```
N "Addrbook"
  [N "Person"
     [N "Name" [N "Takeichi" []],
      N "Email" [N "takeichi@acm.org" []],
      N "Tel" [N "+81-3-5841-7430" []]],
   N "Person"
     [N "Name" [N "Hu" []],
      N "Email" [N "hu@mist.i.u-tokyo.ac.jp" []],
      N "Email" [N "hu@ipl.t.u-tokyo.ac.jp" []],
      N "Tel" [N "+81-3-5841-7411" []]],
   N "Person"
     [N "Name" [N "Mu" []],
      N "Email" [N "scm@iis.sinica.edu.tw" []],
      N "Tel" [N "+81-3-5841-7411" []]]]
```

This very simplified representation omits some features of XML that can be considered simple extensions and some features that are more tricky ones such as IDRefs[1], which we will consider in future work.

3. Bidirectional Transformation Language

This section covers the techniques we have developed for updating and synchronization. In a typical scenario of constructing a structured document, a document designer provides a schema for the source document and a transformation converting it to a view, while the user fills in the data. Our editor unifies the roles of the document designer and the user. It allows the user to create a document and its transformation by editing the view. The user then supplies a schema for the view, from which the editor infers the schema of the source document.

This view-oriented approach requires a mechanism to relate the three components (T, D, and X) through the view. It is similar to *view-updating* [1, 2, 6, 11, 19], as it is known in the database community: given a database and a query that produces a view from the database, the mechanism should reflect view modification back to the database. Two major requirements, however, set our approach apart from the usual applications in the database community.

1. The editor should be able to handle views that contain local data dependencies. For example, in Figure 2 the same names appear twice in the view. This requires synchronization between the view and the source as well as between mutually dependent parts in the view.

2. The editor should allow the user to modify not only the source but also the transformation. Therefore, the editor serves as a user interface for designing the transformation. In contrast, the transformation (query) in the context of database view-updating is usually assumed to be fixed.

In this section we address the first requirement and describe the techniques for synchronization, assuming a fixed transformation given by a document designer. The second requirement is discussed in Section 4, where we formulate an editor in which both the source and transformation can be altered by the user. We will introduce two programming languages, X and Inv. X is the interface language for the users of the editor to describe tree transformations, and it can be embedded into Inv, a general bidirectional language, to gain its bidirectionality. Section 3.1 describes how the formulation of synchronization and updating is split into two languages. To give a feel of how a document designer would use our system, we give an overview of the forward direction of X in Section 3.2, before talking about the semantics of Inv and how X is embedded in Inv in Section 3.3. We discuss the specification of our editor in Section 4.

3.1. From Injective Functions to Bidirectional Transformations

The document designer in our system specifies a *forward* transformation from the source to a view. In addition, we need a *backward* updating for mapping a possibly updated view back to the source. Ideally, the document designer should only need to specify the former. The latter can be left for the system to derive, at least semi-automatically. This task resembles program inversion — given program p, derive another program that maps the output of p to the input of p. To achieve a clearer formulation of backward updating, we initially restrict ourselves to cases where the transformation is an injective function whose inverse can be derived trivially. In our earlier work [18], we defined a programming language, Inv, in which one can define only injective functions.

However, being able to invert injective functions is not enough. For example, every address book produced by the XSLT transformation in Figure 1 contains an index whose entries are consistent with the names appearing in the main body. Assume that the user changes one of the names in the index but not in the body. The

altered, inconsistent view is not associated with any source. In general, the ability to duplicate information enables a programmer to define non-surjective functions, which opens the possibility that the altered view may not be in the range of the transformation. Still, it would be nice if the system were smart enough to guess the user's intention and update the source accordingly, before performing another forward transformation to produce a new, consistent view. Fortunately, a non-injective function can always be simulated by an injective function that pairs the result of the functions with a copy of the the input (hence, it is always injective). However, defining functions this way is error-prone, because it requires explicit manipulation of the copied input.

To resolve this problem, we have designed the interface language X with which the users of our editor can define transformations. The language X enables the document designer to define non-injective functions, as well as providing primitives designed for tree manipulation. Each X construct from A to B, injective or non-injective, can be embedded in Inv as an injective function of type $A \rightarrow (A \times B)$, where the A in the output is a copy of the input. The relation induced by the embedding defines the desired updating behavior.

*3.2. Language **X***

Language X is influenced by similar languages proposed by Greenwald, Moore et al. [12] and Meertens [15]. At the language level, the main extension of X over these languages is a new construct, Dup, which performs duplication and thus describes data dependency inside the view.

3.2.1. An Informal Introduction A document designer, when programming in X, only needs to consider the forward (source-to-view) transformation. The backward updating is left to the system. To give the reader a feel of what it is like to program in X, in this section we informally summarize its semantics in the forward direction.

The syntax of X is given in Figure 3. Primitive transformations are denoted by non-terminal B. Compound transformations are formed by sequencing, product, conditionals, Map, or Fold.

A graphical explanation of the X constructs is given in Figure 4. User-defined primitive bidirectional transformations are introduced either by the construct GFun or by the construct NFun. The construct GFun takes a pair of functions f and g, while NFun takes only one function f. Their forward transformations are given by

$$\begin{array}{lcl} \mathsf{GFun}\ (f,g) & = & f \\ \mathsf{NFun}\ f & = & f. \end{array}$$

Their difference is in the backward updating. In GFun (f,g), function g is the inverse of f *in the range of* f, and vice versa, as specified by the two properties:

$$\begin{array}{l} \mathrm{INV1}:\ f;g;f = f \\ \mathrm{INV2}:\ g;f;g = g, \end{array}$$

$$
\begin{array}{llll}
X & ::= & B & \{\text{ primitives }\} \\
 & | & X \fatsemi X & \{\text{ sequencing }\} \\
 & | & X \otimes X & \{\text{ product }\} \\
 & | & \mathsf{If}\ P\ X\ X & \{\text{ conditional branches }\} \\
 & | & \mathsf{Map}\ X & \{\text{ apply to all children }\} \\
 & | & \mathsf{Fold}\ X\ X & \{\text{ fold }\} \\
 & & & \\
B & ::= & \mathsf{GFun}\ (f, g) & \{\text{ Galois function pairs }\} \\
 & | & \mathsf{NFun}\ f & \{\text{ a simple function }\} \\
 & | & \mathsf{Dup} & \{\text{ duplication }\}
\end{array}
$$

Figure 3. Language X for Specifying Bidirectional Transformations

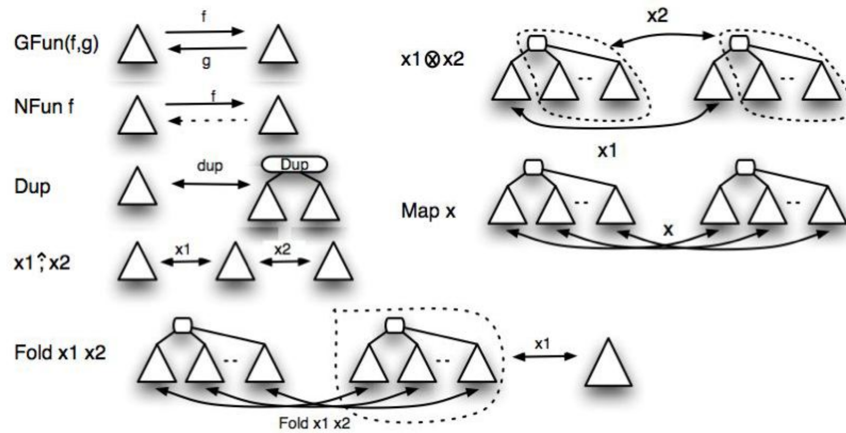

Figure 4. Intuitive Explanation of X Constructs

where the semicolon denotes (reverse) function composition, defined by $(f; g)\ a = g\ (f\ a)$. These two properties are satisfied by all Galois-connected pairs of functions, thus the name GFun. Function f is used for forward transformation, and g is used for backward updating. The construct NFun, on the other hand, takes any function as a forward transformation. The resulting view is assumed to be read-only. The construct NFun is usually used together with Dup (duplication, introduced below) to display information derived from other parts of the document. The derived information is "locked" from any editing action and can be updated only by editing the location from which the information was taken.

A number of useful transformations can be defined in terms of GFun and NFun. The identity transformation is simply

$$\mathsf{idX} \;=\; \mathsf{GFun}\ (id, id).$$

More examples will be given in the next section.

Construct Dup takes a tree and produces two copies of the input, connected by a special node labelled "Dup".

$$\mathsf{Dup}\ t \;=\; \mathsf{N}\ \texttt{"Dup"}\ [t, t]$$

The Dup operator enables one to describe in X value dependency among different parts of the view — when one of the copies is edited by the user, the other should change as well.

Given two bidirectional transformations, x_1 and x_2, the transformation $x_1 \,⨟\, x_2$ informally means "do x_1, then do x_2" (for readability, we use reverse composition, where information flows from left to right):

$$(x_1 \,⨟\, x_2)\ t \;=\; x_2\ (x_1\ t).$$

The product construct $x_1 \otimes x_2$ behaves similarly to products in ordinary functional languages, except that it works on trees rather than pairs. The input tree is sliced into two parts: the left-most child (immediate subtree) of the root, and the root plus the other subtrees. Transform x_1 is applied to the left-most child of the root, while x_2 is applied to the rest. The results are then combined:

$$\begin{array}{lcl}(x_1 \otimes x_2)\ (\mathsf{N}\ n\ (t : ts)) & = & \mathsf{let}\ (\mathsf{N}\ n'\ ts') \;=\; x_2\ (\mathsf{N}\ n\ ts) \\ & & \mathsf{in}\ \mathsf{N}\ n'\ (x_1\ t : ts').\end{array}$$

Combinator If $p\ x_1\ x_2$ applies transform x_1 to the input if the input satisfies predicate p. Otherwise, x_2 is applied.

$$\mathsf{If}\ p\ x_1\ x_2\ t = \begin{cases} x_1\ t, & \text{if } p\ t \\ x_2\ t, & \text{if not } p\ t \end{cases}$$

It may puzzle the reader why If is invertible. The answer is that in its Inv embedding we still keep a copy of the input, so we can simply perform the test again to determine which branch was taken. The details are given in Section 3.3.3.

The forward transform of Map x applies transformation x to all subtrees of the given tree, leaving the root label unchanged:

$$\mathsf{Map}\ x\ (\mathsf{N}\ n\ ts) \;=\; \mathsf{N}\ n\ (map\ x\ ts),$$

where the function map is defined by

$$\begin{array}{lcl} map\ x\ [\,] & = & [\,] \\ map\ x\ (t : ts) & = & x\ t : map\ x\ ts. \end{array}$$

In Fold $x_1\ x_2$, x_2 is applied to the leaves, x_1 is applied to the internal nodes. Its forward transformation is given by

$$\begin{array}{lcl} \mathsf{Fold}\ x_1\ x_2\ (\mathsf{N}\ n\ [\,]) & = & x_2\ n \\ \mathsf{Fold}\ x_1\ x_2\ (\mathsf{N}\ n\ ts) & = & x_1\ (\mathsf{N}\ n\ (map\ (\mathsf{Fold}\ x_1\ x_2)\ ts). \end{array}$$

3.2.2. Examples More X primitives defined using GFun are given in Figure 5. Some take an extra index (i or n) as an argument. In such cases, they can be simply considered as macros defining a collection of functions since our syntax forbids the indices to be results of other X constructs. Another interesting transformation is given by

$$\mathsf{sortX} \;=\; \mathsf{GFun}\ (\mathit{sortT}, \mathit{sortT}),$$

where the function sortT sorts the subtrees of the root based on the contents of the first child of of each subtree. While it is clear that sortT is not invertible, sortT and sortT do satisfy properties INV1 and INV2.

Transformation constX v maps any source to a constant view. Transformation numberX computes the number of the children (immediate subtrees) of the root of the source tree. They are defined, respectively, by

$$\begin{array}{lcl} \mathsf{constX}\ t & = & \mathsf{NFun}\ (\lambda x.\, t) \\ \mathsf{numberX} & = & \mathsf{NFun}\ (\mathit{show} \circ \mathit{length} \circ \mathit{children}), \end{array}$$

where function *children* extracts the list of children of the root of a tree, *length* computes the length of a list and *show* converts a number into a string. Transformation Dup $⨾$ (numberX $\otimes$ idX) maps the input tree to a new tree with two children, the first one being the number of the children of the input tree, while the second one a copy of the input. Although the number in the new tree is not editable because transformation numberX is defined in terms of NFun, its value is updated if we remove a child from or add a child to the copied tree in the view.

Transformation keepX returns the first child of a tree:

$$\mathsf{keepX} \;=\; (\mathsf{idX} \otimes \mathsf{constX}\ (\mathsf{N}\ \texttt{"tmp"}\ [\,])) \fatsemi \mathsf{hoistX}\ \texttt{"tmp"},$$

where hoistX is as defined in Figure 5. We adopt the convention in Haskell that application binds tighter than infix operators. Therefore the definition above should be parsed as (idX $\otimes$ (constX (N "tmp" [ ]))) $⨾$ (hoistX "tmp")

The following X programs define some useful transformations on trees. Transformation insertFstX v inserts document v as the leftmost child of the root, transformation deleteFstX deletes the leftmost child, and transformation modifyRootX n overwrites the root label to n.

$$\begin{array}{lcl} \mathsf{insertFstX}\ v & = & \mathsf{insertHoleX} \fatsemi (\mathsf{replaceHoleX}\ v \otimes \mathsf{idX}) \\ \mathsf{deleteFstX} & = & (\mathsf{constX}\ \Omega \otimes \mathsf{idX}) \fatsemi \mathsf{deleteHoleX} \\ \mathsf{modifyRootX}\ n & = & \mathsf{insertFstX}\ (\mathtt{N}\ n\ [\,]) \fatsemi \mathsf{exchangeX} \fatsemi \mathsf{deleteFstX} \end{array}$$

To manipulate an arbitrary subtree, we introduce the concept of *paths*. A path is a sequence of non-negative integers $[i_1, i_2, \ldots, i_n]$, denoting the subtree obtained by going into the i_1-th child of the root, then into the i_2-th child, and so on. For example, [ ] denotes the root node (or the entire tree), and [0] denotes the first child

- fromPivotX i moves the leftmost subtree rightwards such that it ends up being the ith subtree (counting from 0). toPivotX i does the inverse.

$$\begin{array}{l} \mathsf{fromPivotX}\ i \ = \ \mathsf{GFun}\ (f, f^{-1}) \\ \mathsf{toPivotX}\ i \ = \ \mathsf{GFun}\ (f^{-1}, f) \\ \quad \textbf{where}\ \ f\ (\mathtt{N}\ n\ (t{:}\ ts)) \ = \ \mathtt{N}\ n\ (\mathit{take}\ i\ ts +\!\!+ [t] +\!\!+ \mathit{drop}\ i\ ts) \end{array}$$

- sinkPivotX i moves the leftmost subtree one level down, such that it becomes the first child of the ith subtree in the result. liftPivotX i does the inverse.

$$\begin{array}{l} \mathsf{sinkPivotX}\ i \ = \ \mathsf{GFun}\ (f, f^{-1}) \\ \mathsf{liftPivotX}\ i \ = \ \mathsf{GFun}\ (f^{-1}, f) \\ \quad \textbf{where}\ \ f\ (\mathtt{N}\ n\ (t{:}\ ts)) \ = \ \mathtt{N}\ n\ (\mathit{take}\ i\ ts \\ \qquad\qquad\qquad\qquad +\!\!+ [\mathtt{N}\ m\ (t{:}\ us)] +\!\!+ \mathit{drop}\ (i+1)\ ts) \\ \qquad \textbf{where}\ \ \mathtt{N}\ m\ us \ = \ ts!!i \end{array}$$

- hoistX n: If the root has label n and single child t, then the result is t. newRoot n makes the current tree the single child of a new root with label n.

$$\begin{array}{l} \mathsf{hoistX}\ n \ = \ \mathsf{GFun}\ (f^{-1}, f) \\ \mathsf{newRootX}\ n \ = \ \mathsf{GFun}\ (f, f^{-1}) \\ \quad \textbf{where}\ f\ t = \mathtt{N}\ n\ [t] \end{array}$$

- exchangeX exchanges the root label with the label of the leftmost child (which is assumed to have no children).

$$\begin{array}{l} \mathsf{exchangeX} \ = \ \mathsf{GFun}\ (f, f) \\ \quad \textbf{where}\ \ f\ (\mathtt{N}\ n\ (\mathtt{N}\ m\ [\,] : ts)) \ = \ \mathtt{N}\ m\ (\mathtt{N}\ n\ [\,] : ts) \end{array}$$

- insertHoleX inserts Ω, a special tree denoting a hole, as the leftmost child of the root. deleteHoleX deletes the hole appearing as the leftmost child of the root.

$$\begin{array}{l} \mathsf{insertHoleX} \ = \ \mathsf{GFun}\ (f, f^{-1}) \\ \mathsf{deleteHoleX} \ = \ \mathsf{GFun}\ (f^{-1}, f) \\ \quad \textbf{where}\ \ f\ (\mathtt{N}\ n\ ts) \ = \ \mathtt{N}\ n\ (\Omega : ts) \end{array}$$

- replaceHoleX t replaces the hole with tree t.

$$\begin{array}{l} \mathsf{replaceHoleX}\ t \ = \ \mathsf{GFun}\ (f, f^{-1}) \\ \quad \textbf{where}\ \ f\ \Omega \ = \ t \end{array}$$

Figure 5. Some useful X primitives defined using GFun. We adopt Haskell precedence rules, where application binds tighter than infix operators such as $(+\!\!+)$, $(:)$, etc.

of the root. We define a collection of functions indexed by paths as follows.

$$
\begin{array}{lcl}
\mathsf{applyX}\ [\,]\ x & = & x \\
\mathsf{applyX}\ (i : is)\ x & = & \mathsf{toPivotX}\ i \fatsemi (\mathsf{applyX}\ is\ x \otimes \mathsf{idX}) \fatsemi \mathsf{fromPivotX}\ i \\
\mathsf{fromPivotXP}\ [i] & = & \mathsf{fromPivotX}\ i \\
\mathsf{fromPivotXP}\ (i : is) & = & \mathsf{sinkPivotX}\ i \fatsemi \mathsf{applyX}\ [i]\ (\mathsf{fromPivotXP}\ is) \\
\mathsf{toPivotXP}\ [i] & = & \mathsf{toPivotX}\ i \\
\mathsf{toPivotXP}\ (i : is) & = & \mathsf{applyX}\ [i]\ (\mathsf{toPivotXP}\ is) \fatsemi \mathsf{liftPivotX}\ i
\end{array}
$$

Transformation applyX p x applies transformation x to the subtree at path p while leaving the rest of the tree unchanged. Transformations fromPivotXP and toPivotXP are like fromPivotX and toPivotX but take paths as arguments. Transformation fromPivotXP p moves the pivot, the first child of the root, to path p, while toPivotXP moves the subtree at path p to the pivot position. Although they are written as curried functions accepting paths and thus not part of X, they are intended as macros yielding X terms. With them, transformation moveX p_1 p_2, which moves the subtree at p_1 such that it ends up at path p_2, can be defined by

$$\mathsf{moveX}\ p_1\ p_2 = \mathsf{toPivotXP}\ p_1 \fatsemi \mathsf{fromPivotXP}\ p_2.$$

Similarly, we can define other editing functions; insertX p v inserts document v to the tree at path p, transformation deleteX p deletes the subtree at path p.

$$
\begin{array}{lcl}
\mathsf{insertX}\ p\ v & = & \mathbf{let}\ p' +\!\!+ [i] = p\ \mathbf{in}\ \mathsf{applyX}\ p'\ (\mathsf{insertFstX}\ v \fatsemi \mathsf{fromPivotX}\ i) \\
\mathsf{deleteX}\ p & = & \mathsf{toPivotXP}\ p \fatsemi \mathsf{deleteFstX}
\end{array}
$$

Recall the transformation that converts the source in Figure 1 to the view in Figure 2, where the main difference between the view and the source is that the entries in the view are sorted, and the view has an additional index of names. This transformation can be coded in X as follows.

```
sortX ⨟ Dup ⨟
applyX [0] (modifyRootX "Index" ⨟ Map keepX) ⨟
moveX [0] [0,0] ⨟ modifyRootX "IPL"
```

We start by sorting the address book based on the names by sortX and duplicating the address book. We then keep only the name (first child) for each person in the duplicated address book using Map keepX, and change the root name to "Index" using modifyRootX "Index". Finally, we move the index of names next to the body, and modify the root label to "IPL".

3.3. Language Inv *and* ***X*** *Embedding*

The language Inv was designed by the authors [17] to study bidirectional updating. Expressions in Inv define binary relations over *Val* that are used to handle bidirectional updating. In this section we will introduce Inv, before giving an embedding of X in Inv, thereby giving X a precise semantics. The purpose is that, through the embedding, X constructs gain bidirectionality too.

$$\begin{aligned}\mathsf{Inv} ::= & \ \mathsf{Inv}^{\smile} \mid nil \mid cons \mid node \mid P? \mid V \\ & \mid \delta \mid dupNil \mid dupStr\ String \\ & \mid \mathsf{Inv}; \mathsf{Inv} \mid id \mid \mathsf{Inv} \cup \mathsf{Inv} \\ & \mid \mathsf{Inv} \times \mathsf{Inv} \mid assocr \mid assocl \mid swap \\ & \mid \mu(V \colon \mathsf{Inv}) \\ & \mid prim\ (f, g)\end{aligned}$$

Figure 6. Language Inv.

In Section 3.3.1 we briefly introduce Inv, and in Section 3.3.2 we discuss how it handles duplication and aligns items in different lists. The discussion here is a summary of the authors' previous work [17]. The embedding is presented in Section 3.3.3. For a transformation to be used in an editor, it has to satisfy a set of constraints, called *bidirectionality*, given in Section 3.3.4.

3.3.1. A Quick Overview of Inv Recall the datatype *Val* defined in Section 2.

$$Val ::= \ \ldots \mid \ *Atom \mid Val^{+} \mid Val^{-} \mid \ldots$$

We assume that our backward updating is done in an *online* setting – the user makes changes to a view using an interface provided by the system, and the system performs the corresponding update immediately after each change is made. It is in contrast to *off-line* updating [12], where the user is given a copy of the data, allowed to alter it in all the ways she likes with whatever tools, before executing a command to integrate the data back to the system. In the online scenario, the system is able to record what the user just did by using the *editing tags* $*(_)$, $(_)^{+}$, and $(_)^{-}$. The $*(_)$ tag applies to atomic values (strings) only. When the user changes the value of a string, the editor marks the string with the $*(_)$ tag. The $(_)^{+}$ tag indicates that the tagged element is newly inserted by the user. When the user deletes an element, it is wrapped by a $(_)^{-}$, indicating that it should be deleted but is temporarily left there for further processing. Values containing any of the tags are called *tagged*.

Figure 6 shows the syntax of the subset of Inv we need for this article. The non-terminal P denotes predicates (boolean-valued functions on *Val*), and V denotes a finite set of variable names. Although there are variables, Inv is basically a point-free language in the sense that programs are defined directly on functions, since there is no *let* expression and the only way to introduce a variable is in a fixed-point operator (to be discussed later). The semantics of Inv is given in Figure 7. The semantic function $[\![_]\!]_\eta$ takes an environment η and maps an Inv construct to a binary relation on *Val*. If we restrict the domain and range to non-tagged values, each construct reduces to a partial injective function on *Val*. The environment is a mapping from variables to relations. We write $[\![_]\!]$ when the environment is empty.

A list is built by constructors *nil* and *cons*, where the input of *nil* is restricted to unit type. Constructor *node* produces a tree from a pair consisting of a label and a list of subtrees. One can also produce an empty list or a string using *dupNil* or *dupStr*.

Function *id* is the identity function, the unit of composition. Functions *swap*, *assocl*, and *assocr* distribute the components of the input pair.

Several constructs are defined using their corresponding operations on relations. Relation composition and product are defined by

$$\begin{array}{lcl} f;g & = & \{(a,c) \mid (a,b) \in f \wedge (b,c) \in g\} \\ (f \times g) & = & \{((a,c),(b,d)) \mid (a,b) \in f \wedge (c,d) \in g\}. \end{array}$$

Union of relations is usually used to model conditional branches. However, we need to put another restriction on the two branches — that their domains and ranges, when restricted to non-tagged values, must be disjoint. Therefore we define a $\uplus$ operator:

$$\begin{array}{lcl} f \uplus g & = & f \cup g, \text{ if } (dom_{nt}\ f \cap dom_{nt}\ g) \ = \ (ran_{nt}\ f \cap ran_{nt}\ g) \ = \ \emptyset \\ dom_{nt}\ f & = & \{a \mid (a,b) \in f \ \wedge \ a \text{ not tagged}\} \\ ran_{nt}\ f & = & \{b \mid (a,b) \in f \ \wedge \ b \text{ not tagged}\}, \end{array}$$

where $\emptyset$ denotes the empty set, and $f \uplus g$ is only defined for f and g satisfying the side condition in the definition. The definition of union in Figure 7 is in terms of $\uplus$, which ensures that the union does define an injective function when restricted to non-tagged values. The composition, product, and union constructs in Inv make use of these definitions.

All functions that move the components in a pair around can be defined in terms of *assocr*, *assocl*, *swap*, and the product. We find the following functions useful for defining other Inv programs.

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

For non-tagged values, we have $subr\ (a,(b,c)) = (b,(a,c))$, and $trans\ ((a,b),(c,d)) = ((a,c),(b,d))$.

The δ operator duplicates its argument. We restrict the use of δ to strings.

Only injective functions have inverses. The *relational converse* is a generalisation of the concept of inverses to relations. The converse $^\circ$ of relation R is defined by

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

In Inv we have a *reverse* operator, $(_)^{\smile}$, which, when the input is restricted to non-tagged values, coincides with the relational converse. The reverse of *cons*, for example, decomposes a non-empty list into the head and the tail. The reverse of *nil* matches only the empty list and maps it to the unit value. The reverse of *swap* is itself, and *assocr* and *assocl* are reverses of each other. The reverse of δ, when the input contains no tags, is a partial function accepting only pairs of identical elements. Therefore, the inverse of duplication is an equality test.

However, when the input contains editing tags, the reverse is more than the converse. Tagged values are not in the range of the transformation, and the reverse operator is responsible for looking for a suitable source for these values. When the user edits an atomic value, for example, the action is marked with a $*(_)$ tag. When the two values passed to $\delta^{\smile}$ are not the same and one of them was edited by the user, the edited one should get precedence and be passed along. Therefore, $(*n, m)$ is mapped to $*n$ by $\delta^{\smile}$. If both values are edited, they have to be the same.

How the reverse operator distributes into composition, products, union, and fixed-points is defined by the rules in Figure 7.

The fixed-point $\mu(Y : expr)$, where *expr* is an Inv expression possibly containing a variable Y, is used to define recursion. The variable Y here is use to "tie the knot" — the semantics of $\langle Y : expr\rangle$ is a function mapping relations to relations. To compute the fixed-point, we have to be sure that $\langle Y : expr\rangle$ is continuous on a CPO, but for purpose of this paper, we will not repeat the standard domain theoretic construction. A number of list processing functions can be defined using the fixed-point operator. The function *map* applies the given function to each element of the input list. The function $unzip :: [(a \times b)] \rightarrow ([a] \times [b])$ transposes a list of pairs into a pair of lists, for example, mapping $[(1, \mathsf{a}), (2, \mathsf{b}), (3, \mathsf{c})]$ to $([1, 2, 3], [\mathsf{a}, \mathsf{b}, \mathsf{c}])$. Their definitions in Inv are exactly the point-free (that is, omitting variables and using function composition instead) counterparts of their usual definitions:

$$
\begin{aligned}
map\ f &= \mu(Y : nil^{\smile}; nil\ \cup \\
&\qquad\quad cons^{\smile}; (f \times Y); cons) \\
unzip &= \mu(Y : nil^{\smile}; \delta; (nil \times nil)\ \cup \\
&\qquad\quad cons^{\smile}; (id \times Y); trans; sync).
\end{aligned}
$$

The *sync* operator is equivalent to $(cons \times cons)$ on non-tagged values. Its role in backward updating is essential, and we elaborate in the next section on its interaction with the reverse operator.

Finally, both the forward and backward transform of the *prim* construct are given explicitly. This is used to define some basic X constructs.

3.3.2. Duplication and Alignment We explain how Inv handles propagation of editing tags with a minimal amount of annotation in reverse computation. Consider the function *zip*, well-known by functional programmers, which takes a pair of lists and returns a list of pairs by pairing up elements at corresponding positions. For example it maps $([1, 2, 3], [\mathsf{a}, \mathsf{b}, \mathsf{c}])$ to $[(1, \mathsf{a}), (2, \mathsf{b}), (3, \mathsf{c})]$. We use *zip* as the key function that makes the address book example (in Figure 1 and 2) work. Consider the pair of lists $([1, 2, 3], [\mathsf{a}, \mathsf{b}^-, \mathsf{c}])$. Imagine that the left component is in fact the index of names, the right component is the list of entries, and the $(_)^-$ tag denotes that the user just deleted an entry. In this case *zip* should map the input to $[(1, \mathsf{a}), (2, \mathsf{b})^-, (3, \mathsf{c})]$ — the corresponding name is deleted as well. If we insert a name in the index, say $([1, 2, 4^+, 3], [\mathsf{a}, \mathsf{b}, \mathsf{c}])$, *zip* should map it to $[(1, \mathsf{a}), (2, \mathsf{b}), (4, d)^+, (3, \mathsf{c})]$ for some arbitrary d. Still, we wish to be able to simply define $zip = unzip^{\smile}$. Furthermore we hope that, through carefully designed semantics for primitives like *cons* and *sync*, all the Inv functions we define will be able to

handle the editing tags with a minimal amount of annotation (for example, using *sync* in place of $(cons \times cons)$). Note that this annotation is only necessary at the level of Inv. The X constructs are designed such that all annotations are hidden in the embedding.

The discussion to follow is a simplified version of that in our previous work [17], from which certain aspects (in particular, algebraic rules for reasoning about Inv expressions) have been omitted.

We define several auxiliary functions. The partial functions cn, cn^{+}, and cn^{-} take an element and a list and construct a list only if the element is not tagged, tagged by $(_)^{+}$, and by $(_)^{-}$ respectively (thus $[\![cons]\!]_{\eta} = cn \cup cn^{+} \cup cn^{-}$). A partial function, $notag = (cn^{\circ}; cn) \cup [\![nil^{\circ}; nil]\!]$, passes along only the lists whose first element is not tagged. Functions *del* and *ins* introduce an $(_)^{+}$ or a $(_)^{-}$, respectively, if the input is not tagged.

Expanding the definition of *zip* and distributing the reverse operator inwards, the construct $sync^{\smile}$ indicates that the two lists passed to *zip* must be synchronised. The forward direction of the *sync* operator is defined by $[\![sync]\!]_{\eta} = [\![(cons \times cons)]\!]_{\eta}$. Its reverse, $sync^{\smile}$, takes a pair of lists and extracts their first elements. When one or both of the lists have a tagged element in its head, $sync^{\smile}$ is responsible for tagging the corresponding element in another list or delaying the extracting of the element, such that *zip* behaves as described above.

The first line of the definition of *sync* in Figure 7 says that when the first elements of both lists are not tagged or wrapped by the same tag, we simply extract both of them. The second line says that when only one of them has a $(_)^{-}$ tag, we have to add a $(_)^{-}$ tag to the other one. The third line is for the case when only one of them has a $(_)^{+}$ tag. We use the converse of *snd* to introduce an arbitrary element. For example, $[\![sync^{\smile}]\!]$ maps $(\mathtt{a}^{-} : x, \mathtt{b} : y)$ to $((\mathtt{a}^{-}, x), (\mathtt{b}^{-}, y))$ and maps $(x, b^{+} : y)$ to $((c^{+}, x), (\mathtt{b}^{+}, y))$ for some arbitrary c, if the head of x is not tagged. The tags generated by $sync^{\smile}$ are then processed by the *trans* and products in *zip*. The relation $[\![zip]\!]$ constructed in this way performs the mapping described at the beginning of this section.

The δ operator copies (and in the reverse direction, unifies) atomic values only. To unify structural data, we have to synchronise their shapes as well. Here *zip* plays an essential role. We can use it to define a generic duplication operator. Let dup_a be a type-indexed collection of functions, each having type $a \rightarrow (a \times a)$:

$$
\begin{array}{lcl}
dup_{String} & = & \delta \\
dup_{(a \times b)} & = & (dup_a \times dup_b); trans \\
dup_{[a]} & = & (map\ dup_a); unzip \\
dup_{Tree} & = & \mu(Y : node^{\smile}; (dup_{String} \times map\ Y); trans; (node \times node)).
\end{array}
$$

The definition for $dup_{[a]}$ says that to duplicate a list we should duplicate each element and unzip the resulting list of pairs. The use of *unzip* synchronises the shapes of the two lists in the backward updating. For example, when the input to $dup_{[String]}{}^{\smile}$ is the pair $([\mathtt{a}, \mathtt{c}], [\mathtt{a}, \mathtt{b}^{+}, \mathtt{c}])$, $unzip^{\smile} = zip$, now a relation on non-tagged values, non-deterministically maps the input to $[(\mathtt{a}, \mathtt{a}), (d, \mathtt{b}^{+}), (\mathtt{c}, \mathtt{c})]$ for an arbitrary d. The list of pairs is then processed using $(map\ dup_{String})^{\smile}$. Since

Inv Constructs

$$
\begin{array}{lcl}
[\![nil]\!]_\eta\ () & = & [\,] \\
[\![cons]\!]_\eta\ (a,x) & = & a : x \\
[\![node]\!]_\eta\ (a,x) & = & \mathsf{N}\ a\ x \\
[\![id]\!]_\eta\ a & = & a \\
[\![isStr]\!]_\eta\ n & = & n,\ n \text{ a string} \\
[\![X]\!]_\eta & = & \eta(X),\ X \in V \\
[\![p?]\!]_\eta\ x & = & x,\ p\ x
\end{array}
$$

$$
\begin{array}{lcl}
[\![prim\ (f,g)]\!]_\eta & = & f \\
[\![(prim\ (f,g))^{\smile}]\!]_\eta & = & g
\end{array}
$$

$$
\begin{array}{lcl}
[\![dupNil]\!]_\eta\ a & = & (a,[\,]) \\
[\![dupStr\ s]\!]_\eta\ a & = & (a,s)
\end{array}
$$

$$
\begin{array}{lcl}
[\![swap]\!]_\eta\ (a,b) & = & (b,a) \\
[\![assocr]\!]_\eta\ ((a,b^-),c^-) & = & (a,(b,c)^-) \\
[\![assocr]\!]_\eta\ ((a,b^+),c^+) & = & (a,(b,c)^+) \\
[\![assocr]\!]_\eta\ ((a,b)^-,c) & = & (a^-,(b^-,c)) \\
[\![assocr]\!]_\eta\ ((a,b)^+,c) & = & (a^+,(b^+,c)) \\
[\![assocr]\!]_\eta\ ((a,b),c) & = & (a,(b,c)), \\
\quad b,c \text{ not tagged} & & \\
[\![assocl]\!]_\eta = [\![assocr]\!]_\eta^\circ & &
\end{array}
$$

$$
\begin{array}{lcl}
[\![f;g]\!]_\eta & = & [\![f]\!]_\eta ; [\![g]\!]_\eta \\
[\![f \times g]\!]_\eta & = & ([\![f]\!]_\eta \times [\![g]\!]_\eta) \\
[\![f \cup g]\!]_\eta & = & [\![f]\!]_\eta \uplus [\![g]\!]_\eta \\
[\![\mu(X\colon F)]\!]_\eta & = & fix\ \langle X\colon F\rangle_\eta
\end{array}
$$

$$
\begin{array}{lcl}
[\![\delta]\!]_\eta\ n & = & (n,n) \\
[\![\delta^{\smile}]\!]_\eta\ (n,n) & = & n, \\
\quad n \text{ not tagged or tagged by } {}^+,{}^- & & \\
[\![\delta^{\smile}]\!]_\eta\ (*n,*n) & = & *n \\
[\![\delta^{\smile}]\!]_\eta\ (*n,m) & = & *n,\ m \text{ not tagged} \\
[\![\delta^{\smile}]\!]_\eta\ (m,*n) & = & *n,\ m \text{ not tagged}
\end{array}
$$

$$
\begin{array}{lcl}
[\![(f;g)^{\smile}]\!]_\eta & = & [\![g^{\smile}]\!]_\eta ; [\![f^{\smile}]\!]_\eta \\
[\![(f \times g)^{\smile}]\!]_\eta & = & [\![(f^{\smile} \times g^{\smile})]\!]_\eta \\
[\![(f \cup g)^{\smile}]\!]_\eta & = & [\![f^{\smile}]\!]_\eta \uplus [\![g^{\smile}]\!]_\eta \\
[\![\mu(X\colon F)^{\smile}]\!]_\eta & = & [\![\mu(Y\colon (F[Y^{\smile}/X])^{\smile})]\!] \\
\quad \text{where } Y \text{ is not free in } F & & \\
[\![(f^{\smile})^{\smile}]\!]_\eta & = & [\![f]\!]_\eta \\
[\![f^{\smile}]\!]_\eta & = & [\![f]\!]_\eta^\circ, \\
\quad f \text{ not matching other clauses} & & \\
[\![f]\!]_\eta\ x^+ & = & ins\ ([\![f]\!]_\eta\ x),\ f \neq \delta \\
[\![f]\!]_\eta\ x^- & = & del\ ([\![f]\!]_\eta\ x),\ f \neq \delta \\
[\![f]\!]_\eta\ *x & = & touch\ ([\![f]\!]_\eta\ x),\ f \neq \delta
\end{array}
$$

$$
\begin{array}{lcl}
[\![sync]\!]_\eta & = & [\![(cons \times cons)]\!]_\eta \\
[\![sync^{\smile}]\!]_\eta & = & (cn^\circ \times cn^\circ)\ \cup\ (cn_-^\circ \times cn_-^\circ)\ \cup\ (cn_+^\circ \times cn_+^\circ) \\
 & & \cup\ (cn_-^\circ \times (cn^\circ;(del \times [\![id]\!]_\eta)))\ \cup\ ((cn^\circ;(del \times [\![id]\!]_\eta)) \times cn_-^\circ) \\
 & & \cup\ (cn_+^\circ \times (notag;snd^\circ;(ins \times [\![id]\!]_\eta)))\ \cup\ ((notag;snd^\circ;(ins \times [\![id]\!]_\eta)) \times cn_+^\circ)
\end{array}
$$

Auxiliary Definitions

$$
\begin{array}{lcl}
cn\ (a,x) & = & a : x,\ a \text{ not tagged} \\
cn_+\ (a^+,x) & = & a^+ : x \\
cn_-\ (a^-,x) & = & a^- : x \\
notag & = & (cn^\circ;cn) \cup [\![nil^\circ;nil]\!]
\end{array}
\qquad
\begin{array}{lcl}
del\ a & = & a^-,\ a \text{ not tagged} \\
ins\ a & = & a^+,\ a \text{ not tagged} \\
touch\ a & = & *n,\ n \text{ not tagged} \\
snd\ (a,b) & = & b
\end{array}
$$

$$\langle X\colon F\rangle_\eta\ h\ =\ [\![F]\!]_{\eta'}, \quad \text{where } \eta'(X) = h;\ \eta'(Y) = \eta(Y) \text{ if } Y \neq X$$

$$fix\ h\ =\ \textstyle\bigcup_i h^i(\emptyset), \quad \text{where } \emptyset \text{ is the empty relation}$$

Figure 7. Semantics of Inv. Each Inv construct defines a binary relation on *Val*.

${dup_{String}}^{\smile} = \delta^{\smile}$ has cases only for pairs of input both tagged by $(_)^{+}$ (the first case for $[\![\delta^{\smile}]\!]_{\eta}$ in Figure 7), d is constrained to b^{+}. The only output of $(map\ dup_{String})^{\smile}$ is thus $[\mathsf{a}, \mathsf{b}^{+}, \mathsf{c}]$. In the discussion below, we sometimes omit the type subscript a in dup_a.

3.3.3. Embedding X in Inv As mentioned above, each X transformation is embedded in Inv as an expression having type $S \to (S \times V)$ — the program takes the source and returns a copy of the original source together with the generated view. With the help of Inv, we merely need to define each X construct in Inv in the most obvious way, considering its forward transformation. The rest is guided by the restriction of Inv that no information can be discarded — there are no information-losing constructs like *fst*, *snd*, etc. Therefore, the input often has to be decomposed, passed to auxiliary functions, and composed again. To actually get rid of an entity, we simply run the computation used to create the entity in reverse (for example, in the embedding of $(\,\hat{;}\,)$). The information flow enforced by Inv determines how the modified view is compared against the previously cached source in the backward updating.

We denote by $\lceil _ \rceil$ the embedding of X into Inv. Primitives GFun and NFun are implemented using the backdoor *prim*:

$$\begin{array}{lcl} \lceil \mathsf{GFun}(f, g) \rceil & = & dup_a; (id \times prim\ (f, g)) \\ \lceil \mathsf{NFun}\ f \rceil & = & dup_a; (id \times prim\ (f, \Pi)) \end{array}$$

where a denotes the source type of f, and Π is the total relation that maps anything to anything of the same type. In both cases the input is first duplicated by *dup*. One of the duplicated trees is then passed to either $prim\ (f, g)$ or $prim\ (f, \Pi)$. The forward transformation is given by f in both cases. For GFun, the backward transformation is specified by the programmer. For NFun, we simply discard the edited view by mapping it to an arbitrary value using relation Π. The reverse of dup_a then picks the one that is the same as the cached source.

The Dup construct, for duplicating the input (of type a), is defined by

$$\begin{array}{l} \lceil \mathsf{Dup} \rceil \ = \ dup_a; (id \times (dup_a; futatsu; mkRoot)) \\ \quad \textbf{where} \ \ futatsu \ = \ (id \times (dupNil; cons)); cons \\ \qquad\qquad\quad mkRoot \ = \ dupStr\ \texttt{"Dup"}; swap; node \end{array} .$$

Here, dup_a is called twice, the first time to produce the cached input and the second time to do the actual duplication. The expression *futatsu*; *mkRoot* simply produces a tree with a root labelled "Dup" from a pair of trees ("futatsu" means "two of" in Japanese).

Assume that we have two embedded transformations $\lceil f \rceil :: A \to (A \times B)$ and $\lceil g \rceil :: B \to (B \times C)$. How should we produce their embedded composition of type $A \to (A \times C)$? The Inv expression $\lceil f \rceil; (id \times \lceil g \rceil)$ applies $\lceil f \rceil$ to the input and $\lceil g \rceil$ to the second part of the result, resulting in $(A \times (B \times C))$. We now need to discard the intermediate value of type B, but there are no information-discarding

constructs in Inv. The solution is to notice that the reverse of $\lceil f \rceil$ takes the pair $(A \times B)$ and reduces it to A. The composition of X programs is thus defined by

$$\lceil f \mathbin{\hat{;}} g \rceil \;=\; \lceil f \rceil; (id \times \lceil g \rceil); assocl; (\lceil f \rceil^{\smile} \times id).$$

It is instructive to look at the effect of $\lceil f \rceil^{\smile}$ during backward upating. Consider $\lceil f \mathbin{\hat{;}} g \rceil^{\smile} = (\lceil f \rceil \times id); assocr; (id \times \lceil g \rceil^{\smile}); \lceil f \rceil^{\smile}$. Let (a, c) be the output of the forward transform, and assume that c is changed to c'. To get the updated source a', we first apply $(\lceil f \rceil \times id)$ to (a, c'), yielding $((a, b), c')$ to reproduce the hidden intermediate value b. After some swapping, $\lceil g \rceil^{\smile}$ is applied to (b, c'), yielding an updated b' if the changes made to c is a valid one (otherwise (b, c') is not in the domain of $\lceil g \rceil^{\smile}$ and the system signals an error). Finally, we send (a, b') to $\lceil f \rceil^{\smile}$, computing the updated a'. This is exactly how backward updating for composition was defined by previous researchers [12, 15], and it follows naturally from our definition.

If the definition of composition above were taken as the implementation, it seems very slow to apply $\lceil f \rceil^{\smile}$ and $\lceil f \rceil$ many times when we have a sequence of compositions. However, for a certain class of f (for example those composed out of NFun and GFun whose two functions are inverses of each other, which include all the examples in Figure 5), we are allowed to perform the optimisation of discarding the intermediate value.

Products in X are defined by

$$\begin{array}{l} \lceil f \otimes g \rceil \;=\; slice; (\lceil f \rceil \times \lceil g \rceil); trans; (slice^{\smile} \times slice^{\smile}) \\ \quad \textbf{where} \quad slice \;=\; node^{\smile}; (id \times cons^{\smile}); subr; (id \times node) \;\; . \end{array}$$

The $slice$ function implements $slice\ (\mathtt{N}\ a\ (t : x)) = (t, \mathtt{N}\ a\ x)$. Its role is simply to deconstruct the tree and rearrange the subtrees for further processing. Embedded transformations f and g are applied to the sliced parts, before the results are combined by $trans$ and $(slice^{\smile} \times slice^{\smile})$.

The conditional combinator is defined by

$$\mathsf{If}\ p\ f\ g \;=\; p?; \lceil f \rceil; (p? \times id) \;\cup\; (\neg p)?; \lceil g \rceil; ((\neg p)? \times id).$$

Since we have a copy of the input, in the backward direction we can simply perform the same test to determine which branch was taken. This is done when we run $(p? \times id)$ and $((\neg p)? \times id)$ backwards.

Let f be an X transformation. Let $trList\ f$ and $trSubtrees\ f$ be macros that apply f to every element of a list and to the list of children of a tree, respectively, while keeping a copy of the input.

$$\begin{array}{lcl} trList\ f & = & map\ f; unzip \\ trSubtrees\ f & = & node^{\smile}; (dup \times f); trans; (node \times node) \end{array}$$

In $trList$, a call to $map\ f$ produces a list of pairs consisting of a copy of the element and the transformed result before it is unzipped. In the reverse direction, $unzip^{\smile}$ takes care of the alignment when the user adds to or removes from the transformed list. With the two helper macros, Map can be defined as

$$\lceil \mathsf{Map}\ f \rceil \;=\; trSubtrees\ (trList\ \lceil f \rceil).$$

Finally, Fold is defined recursively:

$$\begin{array}{lcl} \lceil \mathsf{Fold}\ f\ e \rceil & = & \mu(Y : \mathit{isleaf}; \lceil e \rceil \ \cup\ \mathit{isnode}; \lceil \mathsf{Map}\ Y \ \hat{;}\ f \rceil) \\ \mathit{isleaf} & = & \mathit{node}^{\smile}; \mathit{dupNil}^{\smile}; \mathit{dupNil}; \mathit{node} \\ \mathit{isnode} & = & \mathit{node}^{\smile}; (\mathit{id} \times (\mathit{cons}^{\smile}; \mathit{cons})); \mathit{node}. \end{array}$$

If the input is a leaf (a node with no children), we apply e. Otherwise, we recursively apply Fold f e to the subtrees using Map, before transforming the result using f. To compose embedded transformations, we use the X composition $(_ \ \hat{;}\ _)$.

3.3.4. Bidirectionality To model the behaviour of our editor, for every transform x in X we define a pair of functions $\phi_x :: S \to V$ and $\lhd_x :: (S \times V) \to S$ such that:

$$\begin{array}{lcl} \phi_x s & = & \mathit{snd}\ (\llbracket \lceil x \rceil \rrbracket\ s) \\ s \lhd_x v' & = & \llbracket \lceil x \rceil^{\smile} \rrbracket\ (s, v'), \end{array}$$

where $\mathit{snd}\ (s, v) = v$. The function ϕ_x defines the transformation from the source to the view. The function $\lhd_x$ takes the original source and an edited view and returns an updated source. Greenwald et al. call these operators *get* and *put* respectively [12]. The editor starts with a source document and uses ϕ_x to produce an initial view. After each editing action, $\lhd_x$ is called (with a cached copy of the source) to produce an updated source. The editor then calls ϕ_x to produce a new view. After a *put-get* cycle, we remove the editing tags.

For ϕ_x and $\lhd_x$ to be used in the editor, however, there are certain properties we would like them to satisfy. First of all, we would like to be sure that, after the *put-get* cycle described above, the source and the view are in a stable state, therefore we do not need another *put-get* cycle.

Definition 1 (Bi-idempotence). A pair of functions $\phi :: S \to V$ and $\lhd :: (S \times V) \to S$ is said to be *bi-idempotent* if the following two properties hold:

$$\begin{array}{ll} \text{GET-PUT-GET}: & \phi(s \lhd v) = v, \quad \text{where } v = \phi s; \\ \text{PUT-GET-PUT}: & s' \lhd (\phi s') = s', \ \text{where } s' = s \lhd v \text{ for some } v. \end{array}$$

The PUT-GET-PUT property says that if s' is a recently updated source, mapping it to its view and immediately performing the backward update does not change its value. Note that this property only needs to hold for those s' in the range of $\lhd$. For an arbitrary s, we impose the GET-PUT-GET requirement instead. Let v be the view of s. Updating s with v and taking its view, we get v again. Using the two properties together ensures that if the user alters the view, we need to perform only one *put* followed by one *get*. No further updating is necessary. The authors previously showed that a certain class of Inv expressions, which covers our embedding, is bi-idempotent [17].

Bi-idempotence merely guarantees that the system enters a stable state. Defining $s \lhd v = s$ would satisfy bi-idempotence as well. However, we would like modifications

on the view to actually affect the backward updating, if the altered value was not generated by, for example, NFun. We call this property *update preservation*. There are several possible ways to define update preservation, and we are currently investigating what the best approach would be. In this paper we will focus on bi-idempotence, but give an outline of one possible formulation of update preservation in Section 7.

Finally we define bidirectionality:

Definition 2 (Bidirectionality). A pair of functions $\phi :: S \to V$ and $\lhd :: (S \times V) \to S$ that is both bi-idempotent and update preserving is called *bidirectional*.

Remark: In the work of both Greenwald and Moore [12] and Meertens [15], the following GET-PUT and PUT-GET properties are required for arbitrary v and s':

GET-PUT: $s' \lhd (\phi s') = s'$, for any source s'
PUT-GET: $\phi(s \lhd v) = v$, for any view v.

For our application, the PUT-GET property does not hold for general v, as seen in the address book example. The GET-PUT property implies our PUT-GET-PUT property, but we specify only the weaker constraint in the definition of bi-idempotence. (**End of remark**)

4. XEditor: A Programmable Editor

Our editor provides a presentation-oriented (view-oriented) environment for interactively developing structured documents. It enables users to construct structured documents in a WYSIWYG (what you see is what you get) manner through a sequence of editing operations on the view, and automatically produces a document source (without any data dependency) and a transformation mapping the source document to the view. In addition, we will show that it is possible to derive the schema of the source document from the editing sequence if the schema for the view is given.

4.1. Editing Operations

Figure 8 summarizes the operators the editor provides for users to edit the document view. There are two kinds of operators, view modifiers and view transformers, allowing users to change either the source document or the transformation. Modification on the view caused by a view modifier must be reflected back to the source document while keeping the transformation that maps from the source and the view, whereas modification on the view by a view transformer should keep the source as it is but reflect the change on the transformation. There are five view modifiers that are found in most editors InsertE p t inserts tree t as the first child of the node at path p. DeleteE p deletes the subtree at path p. CopyE p_1 p_2 inserts at path p_2 a copy of the subtree at path p_1. MoveE p_1 p_2 removes the subtree at path

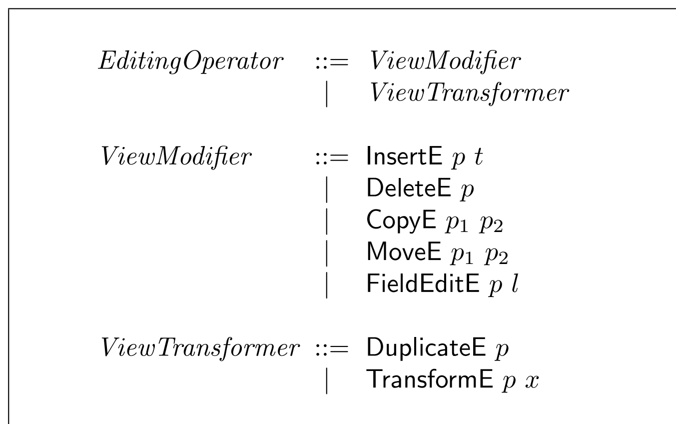

Figure 8. Editing Operations

p_1 and inserts it at path p_2. FieldEditE p l replaces the content of the node at path p by l. The view transformers are unique to this editor; they are used to remember the view modification in the transformation. DuplicateE p duplicates the subtree at path p, making a new node (labeled `"Dup"`) having the two duplicates that are kept identical. It is often used to introduce dependency in the view. TransformE p x applies bidirectional transformation x, defined by an X term, to the tree at path p, which enables use of domain-specific transformations.

4.2. Deriving Structured Documents

This section explains how to obtain the three components for a structured document from a sequence of editing operations on the document view and the schema of the final view. As described in Section 2, the three components of a structured document are the document schema, the document source, and the transformation. We show that the document source and the transformation can be automatically derived from a sequence of editing operations, In addition, we explain how the schema of the source document can be systematically derived from that of the final view for a given transformation.

4.2.1. Automatic Derivation of Document Source and Transformation We define the state of the editor as a triple:

$$\mathcal{S} = (s, x, v),$$

where s and v denote the source and view, respectively, and x denotes a bidirectional transformation. An editor state, $\mathcal{S} = (s, x, v)$, is said to be *stable* if the following *SYNC* property is true:

$$v = \phi_x\ s \text{ and } s = s \lhd_x v.$$

$$
\begin{array}{ll}
\mathsf{InsertE}\ p\ t: & (x,\ v) \mapsto (x,\ \mathit{insert}\ p\ t\ v) \\
\mathsf{DeleteE}\ p: & (x,\ v) \mapsto (x,\ \mathit{delete}\ p\ v) \\
\mathsf{CopyE}\ p_1\ p_2: & (x,\ v) \mapsto (x,\ \mathit{copy}\ p_1\ p_2\ v) \\
\mathsf{MoveE}\ p_1\ p_2: & (x,\ v) \mapsto (x,\ \mathit{move}\ p_1\ p_2\ v) \\
\mathsf{FieldEditE}\ p\ l: & (x,\ v) \mapsto (x,\ \mathit{fieldEdit}\ p\ l\ v) \\
\mathsf{DuplicateE}\ p: & (x,\ v) \mapsto ((x \mathbin{\hat{;}} \mathsf{applyX}\ p\ \mathsf{Dup}),\ v) \\
\mathsf{TransformE}\ p\ x': & (x,\ v) \mapsto ((x \mathbin{\hat{;}} \mathsf{applyX}\ p\ x')),\ v)
\end{array}
$$

where

$$
\begin{array}{lll}
\mathit{insert}\ p\ t\ v & = & \phi_{(\mathsf{insertX}\ p\ t^{+})}\ v \\
\mathit{delete}\ p\ v & = & \mathbf{let}\ t = \phi_{(\mathsf{toPivotXP}\ p \mathbin{\hat{;}} \mathsf{keepX})}\ v \\
 & & \mathbf{in}\ \mathit{insert}\ p\ t^{-}\ (\phi_{(\mathsf{deleteX}\ p)}\ v) \\
\mathit{copy}\ p_1\ p_2\ v & = & \mathit{insert}\ p_2\ (\phi_{(\mathsf{toPivotXP}\ p_1 \mathbin{\hat{;}} \mathsf{keepX})}\ v)\ v \\
\mathit{move}\ p_1\ p_2\ v & = & \mathbf{let}\ t = \phi_{(\mathsf{toPivotXP}\ p_1 \mathbin{\hat{;}} \mathsf{keepX})}\ v \\
 & & \mathbf{in}\ \mathit{insert}\ p_2\ t^{-}\ (\mathit{delete}\ p_1\ v) \\
\mathit{fieldEdit}\ p\ l\ v & = & \phi_{(\mathsf{applyX}\ p\ (\mathsf{modifyRootX}\ *l))}\ v
\end{array}
$$

Figure 9. Semantics of Editing Operations

The editor starts with the initial state ($\mathsf{N}\ \texttt{root}\ [], \mathsf{idX}, \mathsf{N}\ \texttt{root}\ []$), which is obviously a stable state.

Each editing operation changes either x or v, as summarized in Figure 9. The first five editing operations that update the document through the view are defined through auxiliary functions *insert*, *delete*, *copy*, *move* and *fieldEdit*, that only update the view but keep the transformation unchanged. The function *insert* adds into v a new tree t^{+} at the path p, while *delete* marks the subtree at path p with a $(_)^{-}$ tag. Function *fieldEdit* changes the label of the subtree at path p to $*l$. Functions *move* and *copy* can be defined in terms of *insert* and *delete*. Editing operations DuplicateE and TransformE update the transformation but keep the view unchanged.

After each editing operation, we restore the stability of the state:

- if v is changed to v', the new stable state is $(s \lhd_x v', x, \phi_x(s \lhd_x v'))$;
- if x is changed to x', the new stable state is $(s \lhd_{x'} (\phi_{x'} s), x', \phi_{x'}(s \lhd_{x'} (\phi_{x'} s)))$.

The stability of the new state is guaranteed by bi-idempotence. We assume that all inputs to the (ϕ) and $(\lhd)$ operations are in their domains. Otherwise either the editing action on the view is invalid, or the new transformation is not applicable to the current view. In both cases, the editor signals an error to the user and goes back to the state before the editing operation.

It is worth noting the difference between the two forms of editing operations that may appear in our editor: editing operations that directly manipulate views

and editing operations that are formalized as bidirectional transformations. For instance, *EditOperator* may have two different notions of insertion: InsertE p t and TransformE p (insertX [] t). The former inserts a tree into the view and propagates this change to the source, which may trigger further changes in the view. The latter performs an *independent* insertion in the view without altering the source. Note that not every editing sequence is valid in our system. A subtree produced by, for example, NFun, is not editable with InsertE. On the other hand, the user can apply TransformE p (insertX [] t) to the subtree.

4.2.2. Systematic Derivation of Document Schema We turn to the issue of how to derive the document schema. Although one could make use of the existing techniques [5, 9] to extract schemas from documents, we adopt another approach. Noticing that with our editor users perform editing operations on the view and usually have in mind a clear picture of the final view, it is natural to ask them to provide the schema for the final view (see example in Section 5). In this section, we show that we can infer the schema of the document source from the transformation based on the backward type inference algorithm [16].

For the sake of simplicity, we consider a simple form of schema for defining the tree structure of documents:

$$\begin{array}{lcll} T & ::= & \mathsf{Node}\ \mathit{Name}\ \mathit{Children} & \{\ \text{Tree}\ \} \\ \mathit{Children} & ::= & T_1, T_2 & \{\ \text{Sequence}\ \} \\ & \mid & T_1 \mid T_2 & \{\ \text{Choice}\ \} \\ & \mid & [T] & \{\ \text{List}\ \} \\ & \mid & \mathsf{String} & \{\ \text{Text}\ \} \end{array}$$

where *Name* denotes a collection of label names. The structure of a document is a tree, where each node has a label. The children of a node may be a sequence of documents with different schemas, a document with choice of different schemas, a list of documents of the same type, or a simple text node. For instance, the document schema for the address book in the introduction can be defined as follows.

$$\begin{array}{lcl} \mathit{ADDR} & = & \mathsf{Node}\ \mathit{Addrbook}\ [\mathsf{Node}\ \mathit{Person}\ (\mathit{NAME}, [\mathit{EMAIL}], \mathit{TEL})] \\ \mathit{NAME} & = & \mathsf{Node}\ \mathit{Name}\ \mathsf{String} \\ \mathit{EMAIL} & = & \mathsf{Node}\ \mathit{Email}\ \mathsf{String} \\ \mathit{TEL} & = & \mathsf{Node}\ \mathit{Tel}\ \mathsf{String} \end{array}$$

Our inference system, as summarized in Figure 10, is actually a special case of the inverse inference algorithm [16], which shows how the source type can be automatically derived from the view type in a more general setting. Thanks to the simplicity of X and our schema, our inference system turns out of be simpler and more efficient. In Figure 10, we write $x : S \leftrightarrow V$ to denote a judgment which has two meanings: the forward transformation of x maps the source of schema S to the view of schema V (i.e., $x : S \rightarrow V$), and the backward transformation of x returns the source of schema S from the view of schema V (i.e., $x : S \leftarrow V$). To deal with

$$\frac{f : S \to V}{\mathsf{NFun}\ f : S \leftrightarrow V} \quad \text{(NFun-I)}$$

$$\frac{f : S \to V \quad g : V \to S}{\mathsf{GFun}\ (f,\ g) : S \leftrightarrow V} \quad \text{(GFun-I)}$$

$$\mathsf{Dup} : V \leftrightarrow \mathsf{Node}\ \mathit{Dup}\ (V, V) \quad \text{(Dup-I)}$$

$$\frac{x_1 : S \leftrightarrow U \quad x_2 : U \leftrightarrow V}{x_1 \mathbin{⨾} x_2 : S \leftrightarrow V} \quad \text{(Seq-I)}$$

$$\frac{\begin{array}{c} x_1 : S_1 \leftrightarrow V_1 \\ x_2 : \mathsf{Node}\ n\ S_2 \leftrightarrow \mathsf{Node}\ n'\ V_2 \end{array}}{x_1 \otimes x_2 : \mathsf{Node}\ n\ (S_1 \bowtie S_2) \leftrightarrow \mathsf{Node}\ n'\ (V_1 \bowtie V_2)} \quad \text{(Prod-I)}$$

$$\frac{x_1 : S \leftrightarrow V \quad x_2 : S \leftrightarrow V \quad p : S \to \mathit{Bool}}{\mathsf{If}\ p\ x_1\ x_2 : S \leftrightarrow V} \quad \text{(Cond-I)}$$

$$\frac{x : S \leftrightarrow V}{\mathsf{Map}\ x : \mathsf{Node}\ n\ [S] \leftrightarrow \mathsf{Node}\ n\ [V]} \quad \text{(Map-I)}$$

$$\frac{\begin{array}{c} x_2 : \mathsf{String} \leftrightarrow V \\ x_1 : \mathsf{Node}\ m\ [\mathsf{Node}\ n\ V] \leftrightarrow V \\ S = \mathsf{String} \mid \mathsf{Node}\ m\ [S] \end{array}}{\mathsf{Fold}\ x_1\ x_2 : S \leftrightarrow V} \quad \text{(Fold-I)}$$

Figure 10. A Schema Inference System

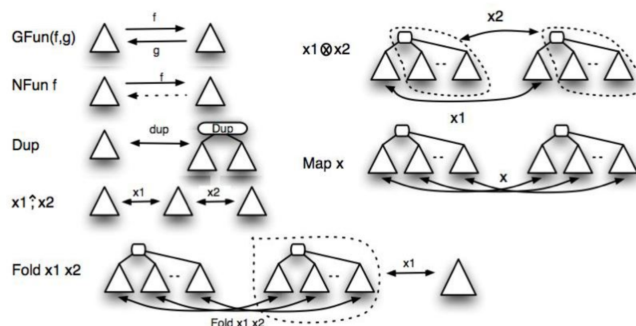

Figure 11. A Snapshot of the Prototyped Editor

ambiguity in the use of sequences and lists, we define

$$T_1 \bowtie T_2 \;=\; \begin{cases} [T_1], & \text{if } T_2 = [T_1] \\ T_1, T_2, & \text{otherwise} \end{cases}$$

and represent the sequence T_1, T_2 by $T_1 \bowtie T_2$ in our inference system. Rules (NFun-I) and (GFun-I) are two base cases showing that the schemas of the source document and the view are determined by the types of the functions used in defining the primitive transformations. Rules (Dup-I) and (Seq-I) are straightforward. Rule (Prod-I) deals with the product transformation, where we use $\bowtie$ to deal with ambiguity in the use of sequences and lists. Rules (Cond-I) and (Map-I) are obvious. In Rule (Fold-I), we define a recursive type (schema), S, for the document source.

This inference system enables us to derive a source schema S from a transformation x and a view type V such that $x : S \leftrightarrow V$. Such S is what we want in the sense that any source data of schema S can be transformed by x to a view that are valid against the schema [16].

5. Editing = Document Developing

We have implemented in Haskell a prototype editing system for supporting this development. Since its purpose is for testing our idea, the editor has a simple user interface. As in Figure 11, the left side is the source, the middle part is the view on which editing operations can be applied, the right side is a set of editing buttons, and the bottom shows the transformation mapping the source to the view. The left source and the bottom transformation do not really need to shown to users; we show them to simplify the testing.

We demonstrate how our editor works by going through the development of the address book example shown in the introduction. We will construct, by interacting with the editor, a source document and a transformation that can produce the view in Figure 2.

The editor starts with an empty view with only one node, labeled `"Root"`.

```
N "Root" []
```

The node or subtree in focus, on which the user performs editing operations, is selected using the cursor. Here, for simplicity, we use a path to denote the selected subtree. The complete list of operations the user can perform on the focused subtree has been given in Section 4. We show below how these operations are used for developing the address book.

We first change the label `"Root"` to `"Addrbook"` by using the FieldEditE operation,

```
N "Addrbook" []
```

then, by using the InsertE operation, we insert a name and contact information as a subtree of the root (the node at position []), which could be done by inserting nodes one by one.

```
N "Addrbook"
 [N "Person"
   [N "Name" [N "Takeichi" []],
    N "Email" [N "takeichi@acm.org" []],
    N "Tel" [N "+81-3-5841-7430" []]]]
```

We continue adding more names and contact information by copying the subtree rooted at path [1] by using the CopyE operation. The copied tree becomes a sibling of the original:

```
N "Addrbook"
 [N "Person"
   [N "Name" [N "Takeichi" []],
    N "Email" [N "takeichi@acm.org" []],
    N "Tel" [N "+81-3-5841-7430" []]],
  N "Person"
   [N "Name" [N "Takeichi" []],
    N "Email" [N "takeichi@acm.org" []],
    N "Tel" [N "+81-3-5841-7430" []]]]
```

We then change the values at the nodes to the second person's name and contact information:

```
N "Addrbook"
 [N "Person"
   [N "Name" [N "Takeichi" []],
    N "Email" [N "takeichi@acm.org" []],
    N "Tel" [N "+81-3-5841-7430" []]],
  N "Person"
   [N "Name" [N "Hu" []],
    N "Email" [N "hu@mist.i.u-tokyo.ac.jp" []],
    N "Tel" [N "+81-3-5841-7411" []]]]
```

Note that we are editing both the source document and the view, though we are not quite aware of this fact yet. Transformation X, is currently simply the identity transformation idX. Now suppose we want to sort the entries by name. We do this by selecting all the people and applying the sortX transformation to their names by using editing operation TransformE. The result looks like

```
N "Addrbook"
 [N "Person"
   [N "Name" [N "Hu" []],
    N "Email" [N "hu@mist.i.u-tokyo.ac.jp" []],
    N "Tel" [N "+81-3-5841-7411" []]],
  N "Person"
   [N "Name" [N "Takeichi" []],
    N "Email" [N "takeichi@acm.org" []],
    N "Tel" [N "+81-3-5841-7430" []]]]
```

Next, suppose we want to make an index of the names. We first duplicate the address book using the DuplicateE operation:

```
N "Dup"
[N "Addrbook"
   [N "Person"
     [N "Name" [N "Hu" []],
      ... ],
    N "Person"
     [N "Name" [N "Takeichi" []],
      ...]],
 N "Addrbook"
   [N "Person"
     [N "Name" [N "Hu" []],
      ... ],
    N "Person"
     [N "Name" [N "Takeichi" []],
      ...]]]
```

and we then apply transformation keepX by using TransformE to keep *only* the names from the duplicated address book (and change the tag `"Addrbook"` to `"Index"`):

```
N "Dup"
[N "Index"
   [N "Name" [N "Hu" []],
    N "Name" [N "Takeichi" []]]
 N "Addrbook"
   [N "Person"
     [N "Name" [N "Hu" []],
      ... ],
    N "Person"
```

```
        [N "Name" [N "Takeichi" []],
         ...]]]
```

This duplication shows a key feature of our system. It differs from the copy operation we performed to add a new person to the address book. There the copied data are independent. but here the duplicate operation keeps the subtree and its duplicate synchronized. In this example, deletion, insertion, or modification of a person's information on one side causes corresponding changes on the other side, unless we explicitly instruct the editor to perform the editing operations independently. The keepX transformation used in the TransformE operation, for example, is such an independent transformation. If it is applied to the subtree at path [0] to extract the names, the main address book at path [1] remains unchanged. In contrast, if we insert the following entry (by using the InsertE operation)

```
N "Person"
  [N "Name" [N "Mu" []],
   N "Email" [N "scm@iis.sinica.edu.tw" []],
   N "Tel" [N "+81-3-5841-7411" []]]
```

into the `"Addrbook"` subtree at path [1] as its last child, the name `"Mu"` will automatically appear in the index of names, resulting in:

```
N "Dup"
[N "Index"
   [N "Name" [N "Hu" []],
    N "Name" [N "Mu" []]],
    N "Name" [N "Takeichi" []],
 N "Addrbook"
   [N "Person"
     [N "Name" [N "Hu" []],
      ... ],
    N "Person"
     [N "Name" [N "Mu" []],
      ...]],
    N "Person"
     [N "Name" [N "Takeichi" []],
      ...]]]
```

Note that although the entry is inserted (by the user) as the last child of the `"Addrbook"` in the view, the resulting view has both the entries under `"Addrbook"` and the names under `"Index"` sorted.

Finally, after renaming the root to `IPL`, we are done. Note that it is possible to go further to edit the view by applying complex transformations through TransformE. Consider, as an example, that we want to display the email address in the following way: if the email address is ended with `".jp"`, it will be displayed as it is and allow later modification, otherwise it will be displayed as `"***"` and cannot be modified any more. To do so, we may apply the following transformation to the subtree at

path [1] via TransformE.

```
dispPersonEmail =
    Map (toPivotX 1 ;
        (If addressEndedWithJP idX (modifyRootX "***")) ⊗ idX ;
        fromPivotX 1)
```

Here users have to write X code to perform transformations that are more complicated. To simplify such coding in practice, it would be better to prepare a library of useful transformations.

In summary, the important features of our programmable editor are as follows.

- It is presentation-oriented (view-oriented), meaning the developer can directly edit the view, the exact display of the document. This WYSIWYG style is more user-friendly than existing editors whose view is changed either through source editing or by transformation modification.
- It enables simple description of data dependency in the view by using the DuplicateE operations, and provides an efficient way to maintain data consistency in the view. As far as we are aware, this is the first structured document editor with local data synchronization.
- It automatically keeps consistency among the source data, the view, and the bidirectional transformation that links the source data and the view. The source data and the transformation are gradually built while the user edits the view.

6. Related Work

Many XML editors [22] have been designed and implemented for supporting development of structured documents in XML. Most of them, such as XMLSpy [14], force the user to develop structured documents in the order of DTD, document content, and presentation. These kinds of tools cannot effectively support interactive document development, which is considered to be very important in document engineering [8, 24]. Moreover, these tools require developers to have in-depth knowledge about DTD, XML and XSLT. In contrast, our editor provides a single, integrated, WYSIWYG interface and requires less knowledge about XML.

The most closely related system to ours is Proxima [13, 21], a presentation-oriented generic editor designed for all kinds of XML documents and presentations. It is very similar to our system; it enables the description of transformations and of computations on views through editing operations. However, for each transformation and computation, users must prepare two functions to explicitly express the two-way transformation. In contrast, we provide a bidirectional language with the view-updating technique, facilitating bidirectional transformation. Another similar system is TreeCalc [23], a simple tree version of a spreadsheet system. However, TreeCalc does not support structure modification on the view.

Our representation of the editor state by a triple (document source, transformation, and view) is inspired by the work on view-updating [1, 2, 6, 11, 19] in the

database community, where modification on the view can be reflected back to the original database. We borrow this technique with a significant extension: editing operations can modify not only the view but also the query, which has not been exploited before. Since our transformation language does not have the JOIN operator, the problem of costly propagation of deletions and annotations through views [20] does not arise.

During the design of our bidirectional transformation language, X, we learned much from the lens combinators [7, 12], which give a semantic foundation and a core programming language for bidirectional transformations on tree-structured data. The current lens combinators can clearly specify dependence between the source data and the view, but cannot describe dependency *inside a view*. This is not a problem for data synchronization, but it has to be remedied in our view-oriented editor. It would be interesting to see whether the lens combinators can be enriched with duplication by relaxing the requirement in the "PUT-GET" and "GET-PUT" properties. In contrast, our language with duplication clarifies dependencies. Another very much related language is that proposed by Meertens [15], which is designed for specifying constraints in the design of user interfaces. Again, the language cannot deal with dependency inside a view.

Our idea of duplication in X is greatly influenced by the invertible language of Glück and Kawabe [10], where duplication is considered to be the inverse of equality check and vice versa. In inverse computation, an inverse function computes an input merely from the output, but in bidirectional transformation, backward updating can use both the output and the old input to compute a new input. Therefore, adding duplication to a bidirectional language needs a more complex equality check mechanism. It would be interesting to see if inverse transformation with duplication can support view updating and to compare these two approaches [17, 18].

7. Conclusion and Future Work

We have described a presentation-oriented editor suitable for interactive development of structured documents. The novel use of the view-updating technique in the editor, the duplication construct in our bidirectional language, and the mechanism of changing the transformation through editing operations play key roles in the design of our editor. Our prototype system with automatic view updating shows the promise of this approach; structured documents with *equal dependency* (one part of the document equals to another) can be fully captured and interactively developed with our system.

We have several research directions for the future. First, we have not addressed in detail the optimization of bidirectional transformations in X, which would play an important role in large practical applications. In fact, the compositional style of X may introduce many unnecessary intermediate results, which would make backward transformation very costly.

Second, to complete the discussion on bidirectionality, we have yet to define update preservation. One possible approach is to define an ordering ($\preceq$) on *Val* indi-

cating how "updated" a value is. For example, we have

$$
\begin{aligned}
& \texttt{N "Dup" [N "a" ["b", "c"], N "a" ["b", "c"]]} \\
\prec\ & \texttt{N "Dup" [N "a" ["b", "d"}^{+}\texttt{, "c"], N "a" ["b", "c"]]} \\
\prec\ & \texttt{N "Dup" [N "a" ["b", "d"}^{+}\texttt{, "c"], N "a" ["b", "d"}^{+}\texttt{, "c"]]}.
\end{aligned}
$$

We require that, after a *put-get* cycle, a view does not "fall back" to a less updated one. That is, a pair of functions ϕ and $\lhd$ is *update preserving* with respect to $(\preceq)$ if $v \preceq \phi(s \lhd v)$. Furthermore, we have to extend *Val* with yet another tag to keep track of values generated by *dupStr* and NFun, such that those values with the tag are always related to each other by $(\preceq)$. Other possible approaches have also been suggested, and we are currently looking into them and studying their properties.

Third, the schema inference system needs more refinement and should be discussed more formally. The schema language in this paper is much simpler than the existing schema languages like DTD and Schema. It would be interesting to see how to extend the schema inference system in this paper to obtain a more practical schema derivation algorithm.

Acknowledgments

We wish to thank Atsushi Ohori for introducing us the work on the view updating technique, which initially motivated this work. We should thank Dongxi Liu, Yasushi Hayashi, Keisuke Nakano, and Shingo Nishioka, the PSD project members in University of Tokyo, for stimulating discussions on the design and implementation of this editor, and thank our students Kento Emoto, Kazutaka Matsuda, and Akimasa Morihata for helping us to implement the prototype system. We would also like to thank anonymous referees and Julia Lawall for many useful comments and suggestions on improving the paper's structure and presentation.

This work is partly sponsored by Comprehensive Development of e-Society Foundation Software Program of the Ministry of Education, Culture, Sports, Science and Technology of Japan, and by the National Natural Science Foundation of China under Grant No. 60528006.

Notes

1. In XML, an attribute having type ID is an unique name for the element, while an attribute having type IDRef is a reference to an ID.

References

1. Serge Abiteboul. On views and XML. In *Proceedings of the 18th ACM SIGPLAN-SIGACT symposium on Principles of Database Systems*, pages 1–9. ACM Press, 1999.
2. Francois Bancilhon and Nicolas Spyratos. Updating semantics of relational views. *ACM Transactions on Database Systems*, 6(4):557–575, 1981.
3. Richard Bird. *Introduction to Functional Programming using Haskell.* Prentice Hall, 1998.
4. Tim Bray, Jean Paoli, and C.M. Sperberg-McQueen. Extensible markup language (XML) 1.0. 1998.

5. Boris Chidlovskii. Schema extraction from XML collections. In *Proceedings of the second ACM/IEEE-CS joint conference on Digital libraries*, pages 291–292. ACM Press, 2002.
6. Umeshwar Dayal and Philip A. Bernstein. On the correct translation of update operations on relational views. *ACM Transactions on Database Systems*, 7(3):381–416, 1982.
7. J. Nathan Foster, Michael B. Greenwald, Jonathan T. Moore, Benjamin C. Pierce, and Alan Schmitt. Combinators for bi-directional tree transformations: a linguistic approach to the view update problem. In *ACM SIGPLAN–SIGACT Symposium on Principles of Programming Languages (POPL), Long Beach, California*, pages 233–246, 2005.
8. Richard Furuta, Vincent Quint, and Jacques André. Interactively editing structured documents. *Electronic Publishing Origination, Dissemination, and Design*, 1(1):19–44, 1988.
9. Minos Garofalakis, Aristides Gionis, Rajeev Rastogi, S. Seshadri, and Kyuseok Shim. Xtract: a system for extracting document type descriptors from XML documents. In *Proceedings of the 2000 ACM SIGMOD international conference on Management of data*, pages 165–176. ACM Press, 2000.
10. Robert Glück and Masahiko Kawabe. A program inverter for a functional language with equality and constructors. In Atsushi Ohori, editor, *Programming Languages and Systems. Proceedings*, volume 2895 of *Lecture Notes in Computer Science*, pages 246–264. Springer-Verlag, 2003.
11. Georg Gottlob, Paolo Paolini, and Roberto Zicari. Properties and update semantics of consistent views. *ACM Transactions on Database Systems*, 13(4):486–524, 1988.
12. Michael B. Greenwald, Jonathan T. Moore, Benjamin C. Pierce, and Alan Schmitt. A language for bi-directional tree transformations. Technical Report Technical Report MS-CIS-03-08, Department of Computer and Information Science University of Pennsylvania, August 2003.
13. Johan Jeuring. Implementing a generic editor. In *2nd Workshop on Programmable Structured Documents*, February 2004.
14. Larry Kim. *The Official XMLSPY Handbook*. John Wiley & Sons, 2002.
15. Lambert Meertens. Designing constraint maintainers for user interaction. `http://www.cwi.nl/~lambert`, June 1998.
16. Tova Milo, Dan Suciu, and Victor Vianu. Typechecking for XML transformers. In *Proceedings of the Nineteenth ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems*, pages 11–22. ACM, 2000.
17. Shin-Cheng Mu, Zhenjiang Hu, and Masato Takeichi. An algebraic approach to bi-directional updating. In *Second ASIAN Symposium on Programming Languages and Systems(APLAS 2004)*, pages 2–18, Taipei, Taiwan, November 2004. Springer, LNCS 3302.
18. Shin-Cheng Mu, Zhenjiang Hu, and Masato Takeichi. An injective language for reversible computation. In *Seventh International Conference on Mathematics of Program Construction (MPC 2004)*, pages 289–313, Stirling, Scotland, July 2004. Springer, LNCS 3215.
19. Atsushi Ohori and Keishi Tajima. A polymorphic calculus for views and object sharing. In *Proceedings of ACM Symposium on Principles of Database Systems (PODS)*, pages 255–266, 1994.
20. Peter Buneman and Sanjeev Khanna and Wang-Chiew Tan. On Propagation of Deletion and Annotation Through Views. In *Proceedings of ACM Symposium on Principles of Database Systems (PODS)*, Wisconsin, Madison, June 2002.
21. Martijn M. Schrage and Johan Jeuring. Xprez: A declarative presentation language for XML. available at `http://www.cs.uu.nl/research/projects/proxima/`, 2003.
22. XML Software. A list of XML editors. See `http://www.xmlsoftware.com/editors.html`, 2004.
23. Masato Takeichi, Zhenjiang Hu, Kazuhiko Kakehi, Yasushi Hayashi, Shin-Cheng Mu, and Keisuke Nakano. Treecalc : Towards programmable structured documents. In *JSSST Conference on Software Science and Technology*, September 2003.
24. Lionel Villard, Cecile Roisin, and Nabil Layada. An XML-based multimedia document processing model for content adaptation. In *8th International Conference on Digital Documents and Electronic Publishing, LNCS 2023*, September 2000.