Higher-Order and Symbolic Computation, 1, 1–31 (2006)
© 2006 Kluwer Academic Publishers, Boston. Manufactured in The Netherlands.

A Programmable Editor for Developing Structured Documents Based on Bidirectional Transformations

ZHENJIANG HU hu@mist.i.u-tokyo.ac.jp
SHIN-CHENG MU scm@ipl.t.u-tokyo.ac.jp
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 transformations to 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 the document view, 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 transformation can be obtained through editing operations. The main tricks behind are the utilization of the view-updating technique developed in the database community, and a new bidirectional transformation language that can describe not only relationship between the document source and its view, but also data dependency in the view.

This is an extended version of the paper presented at PEPM 2004.

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

1. Introduction

XML [4] has been attracting a tremendous surge of interest as a universal, queryable representation for structured documents. Everyday a countless number of structured documents in XML are constructed, and so many editors [19] are designed and implemented to support the construction of XML 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 representation of structured documents and information interchange. While the existing XML editors are helpful for the *creation* of the documents, they are rather weak to support development of structured documents in the sense they hardly provide powerful mechanism for dynamic *refinement* of the structured documents.

Let us take a close look at the process of using existing editors to develop a simple address book (Figure 1). It basically includes three steps: designing a suitable document type, constructing an XML document with the designed type for storing information, and defining a transformation for viewing the document. Concretely, we may start by defining an address book type (i.e., DTD), which allows

```
<!ELEMENT addrbook (person*)>
<!ELEMENT person (name, email*, tel)>
<!ELEMENT name (#PCDATA)>
<!ELEMENT email (#PCDATA)>
<!ELEMENT tel (#PCDATA)>
```

DTD

```
<xsl:template match="/addrbook">
  <addrbook>
    <index>
      <xsl:for-each select="person">
        <xsl:sort select="name"/>
        <xsl:value-of select="name"/>
      </xsl:for-each>
    </index>
    <xsl:for-each select="person">
      <xsl:sort select="name"/>
      <xsl:value-of select="person"/>
    </xsl:for-each>
  </addrbook>
</xsl:template>
```

XSLT

```
<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>
```

XML

Figure 1. Address Book: A Structure Document

an arbitrary number of people's addresses including a name, some email addresses if there are, and a telephone number. Then, we construct an XML document (i.e., XML) of this type to store address information. And finally, we define a transformation (i.e., XSLT) to view the address book in a friendly way (Figure 2), say by sorting persons according to the last names and adding an index of names. The result of this development is a structured document with three components: a data type definition, an XML document representing the source data, and a transformation for viewing the data. Two remarks are worth making on the view. First, to simplify the presentation, we consider the view as another XML data, which is possible to be presented in a more friendly format with a suitable stylesheet description. Second, besides difference in their structures between the two XML documents, the original XML document in Figure 1 and the view in Figure 2, the original document has no redundant information, while the view does; e.g., the same names appear twice in the view.

During the development of a structured document, none of the three components is always fixed. Instead, they all keep evolving. It has been observed that document development follows a life-cycle similar to the development of computer programs, in which the document is repeatedly refined. However, the existing editors do not support interactive refinement very well:

```
<addrbook>
  <index>
    <name> Zhenjiang Hu </name>
    <name> Shin-Cheng Mu </name>
    <name> Masato Takeichi </name>
  </index>
  <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>
  <person>
    <name> Masato Takeichi </name>
    <email> takeichi@acm.org </email>
    <tel> +81-3-5841-7430 </tel>
  </person>
</addrbook>
```

Figure 2. A View of the Address Book in XML

- First, they treat the three components of a structured document independently, which makes it hard to keep them consistent with each other. Take the address book example, if we want to make a change on the data type by splitting the telephone number (`tel`) into two parts, country code (`ccode`) and local code (`tel`), to share the country code, we may refine the document type definition in Figure 1 to the following.

  ```
  <!ELEMENT addrbook (ccode, person*)>
  <!ELEMENT ccode (#PCDATA)>
  <!ELEMENT person (name, email*, tel)>
  <!ELEMENT name (#PCDATA)>
  <!ELEMENT email (#PCDATA)>
  <!ELEMENT tel (#PCDATA)>
  ```

  This refinement requires corresponding changes on the XML document and the transformation, which is difficult.

- Second, users are supposed to be very familiar with XML, knowing DTD, XML, and XSLT for the construction of the three components of structured documents. This may be rather disappointing to those who know very little about XML (for example, those possessing merely some basic knowledge of HTML), but still want to create structured documents in their daily work. In fact, more and more people nowadays want to be able to create their structured documents in a user-friendly manner, pretty much like how spreadsheets are created. The intuitive interface of the latter contributes a lot to its 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* together with a data type definition for the final view, an efficient and reliable structured document with the three basic components can be derived automatically.

One challenge in design and implementation of such editing systems is to find an efficient way for maintaining consistency of the source document and its view even when there is local data dependency in the view. Consider the view in Figure 2, we would wish that when the user, for example, adds or deletes a person, the original document in Figure 1 be updated correspondingly. Further more, the changes should also trigger an update of the index of names in Figure 2. We may even wish that when an additional name is added to the index, a fresh, empty person will be added to the person bodies in both the source document and the view.

The main trick behind our editor is a new bidirectional transformation language to describe the relationship between the source data and the view. Our main contributions can be summarized as follows.

- We, as far as we are aware, are the first to recognize the importance of the view-updating technique for interactive development of structured documents. The view-updating technique [2, 5, 9, 16, 1] has been intensively studied in the database community, where modification on the view can be reflected back to the original database. We borrow this technique and use it in the design of our editor with a significant extension not exploited before: editing operations can modify not only the view but also the transformation (from the database to the view).
- We have designed a powerful language for the specification of the relationship between the original data and the view. Our language is similar to that in [13, 10], extended with a special construct to duplicate data. Lots of efforts were put into handling data dependency within the view. The language is powerful enough to describe the editing operations (insert, delete, move, and copy) as well as other important transformations.
- We have successfully implemented our idea in a prototype editor. The editor is particularly interesting in its programmability and a unified, presentation-oriented interface for developing the three components through editing operations on the view.
  - *Presentation-oriented*: the editor has a uniform view-based editing interface for users to construct and refine their documents.
  - *Programmable*: transformations can be constructed through interactive editing operations. In fact, thanks to the bidirectional language, the three basic components of structured documents can be automatically derived, after editing the view.

The rest of the paper is organized as follows. We start by giving a simple definition of structured documents in functional notations in Section 2. After defining the

bidirectional transformation language that plays an important role in our editor in Section 3, we propose the design principle and implementation technique in Section 4, and demonstrate how our system can assist development of structured documents in Section 5. Related work and conclusions are given in Sections 6 and 7 respectively.

2. Structured Documents

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

- T: the type of the source document;
- D: the source document;
- X: the transformation mapping the source document to another document for display. The document displayed to the user is called the *view*.

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

In this paper, however, for the sake of conciseness we will introduce a lighter-weight notation for types and documents, and our own transformation language X which supports bidirectional transformations. The language will be discussed in detailed in Section 3. In this section we will briefly talk about the document type and document representation. Throughout this paper, we choose a Haskell-like [3] notation to express our idea.

We use the following language to define schemes (types) of documents.

$$
\begin{array}{lcll}
T & = & \mathsf{Node}\ \mathit{Name}\ \mathit{Children} & \{\ \text{Tree}\ \} \\
\mathit{Children} & = & (T_1, T_2, \ldots, T_n) & \{\ \text{Tuple}\ \} \\
 & | & [T] & \{\ \text{List}\ \} \\
 & | & \mathsf{String} & \{\ \text{Text}\ \} \\
\mathit{Name} & = & \mathsf{String} & \{\ \text{Label name associated to the node}\ \}
\end{array}
$$

A document is basically a tree, whose nodes, being associated with a label name (tag name), have children that may be a tuple of documents with different types, a list of documents with the same type, or a simple text node. For instance, the document type for the address book in the introduction can be defined as follows.

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

Here we use *Addrbook*, *Name*, *Email*, and *Tel* to denote special constant types for corresponding tag names. The document source in the introduction can be accordingly represented as the following data of type *ADDR*.

```
Node Addrbook
  [Node Person
     (Node Name "Masato Takeichi",
      [Node Email "takeichi@acm.org"],
      Node Tel "+81-3-5841-7430"),
   Node Person
     (N Name "Zhenjiang Hu",
      [Node Email "hu@mist.i.u-tokyo.ac.jp",
       Node Email "hu@ipl.t.u-tokyo.ac.jp"],
      Node Tel "+81-3-5841-7411"),
   Node Person
     (Node Name "Shin-Cheng Mu",
      [Node Email "scm@mist.i.u-tokyo.ac.jp"],
      Node Tel "+81-3-5841-7430")]
```

To support transiently illegal document structures during interactive development of structured documents, we also introduce a generic (untyped) document type that is only used during interactive document development:

$$G \;=\; \mathsf{N}\ \mathit{Name}\ [G]$$

Different from a type in T, G treats every subtree the same way. For example, the document source in the introduction is represented as follows.

```
N "Addrbook"
  [N "Person"
     [N "Name" [N "Masato Takeichi" []],
      N "Email" [N "takeichi@acm.org" []],
      N "Tel" [N "+81-3-5841-7430" []]],
   N "Person"
     [N "Name" [N "Zhenjiang 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 "Shin-Cheng Mu" []],
      N "Email" [N "scm@mist.i.u-tokyo.ac.jp" []],
      N "Tel" [N "+81-3-5841-7430" []]]]
```

Now we have two representations corresponding to a document, one for the final result and one for develop process. These two representations are *bidirectional* in the sense that for instance, given $ADDR$, a document in type $ADDR$ can be mapped to that of type G, and vice versa. The forward mapping is simply done by ignoring

tuples and text nodes:

$$
\begin{array}{lcl}
t2g & :: & T \rightarrow G \\
t2g\ (\mathsf{Node}\ n\ ch) & = & \mathsf{N}\ (label2string\ n)\ (t2g'\ ch) \\
t2g'\ (t_1, t_2, \ldots, t_n) & = & [t2g\ t_1, t2g\ t_2, \ldots, t2g\ t_n] \\
t2g'\ [] & = & [] \\
t2g'\ (t : ts) & = & t2g\ t : t2g'\ ts \\
t2g'\ s & = & \mathsf{N}\ s\ []
\end{array}
$$

where *label2string* is a function to turn a label name to a string, and the backward mapping can be realized by a type-directed transformation.

$$
\begin{array}{lcl}
g2t & :: & G \rightarrow TypeT \rightarrow T \\
g2t\ (\mathsf{N}\ n\ [])\ \mathsf{String} & = & n \\
g2t\ (\mathsf{N}\ n\ ch)\ (\mathsf{Node}\ n'\ t) & = & \mathsf{Node}\ n'\ (g2t\ ch\ t),\ \textbf{if}\ n = tag2name\ n' \\
g2t\ (c : cs)\ (T_1, T_2) & = & (g2t\ c\ T_1, g2t\ cs\ T_2) \\
g2t\ (c : cs)\ [T] & = & g2t\ c\ T\ :\ g2t\ cs\ [T]
\end{array}
$$

The second argument to *g2t* is the type definition of the document type in T; in the address book above, it refers to the definition of *ADDR*.

3. A Bidirectional Transformation Language

Our editor is view-oriented in that it allows users to develop their structured documents by editing the view. The editor then produces the three components of a structured document automatically.

This view-oriented environment requires a mechanism to relate the three components with the view. The technique we need is related to *view-updating* [2, 5, 9, 16, 1] intensively studied in the database community: given a database and a query which produces a view from the database, to reflect view modification upon the database. Though the idea is very similar, there are two major difficulties applying the existing technique for our view-oriented editor.

1. Our view may contain local data dependency as seen in Figure 2 where the same name appears twice in the view. This requires synchronization both between the view and the source document and between mutually dependent parts in the view.

2. Our view modification should be reflected not only on the source document, but also on the transformation. In other words, the transformation (query), which is assumed to be fixed in the existing view-updating technique, should be modifiable in our framework.

Our solution is to design a programming language, which we call X, to specify the transformation from the source document to the view. The language X is embedded in another programming language called Inv, designed to deal with duplication and alignments in bi-directional editing. It enables us to automatically derive view-to-source updating from the source-to-view transformation. We then use X in the

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

Figure 3. The Language X for Specifying Bidirectional Transformations

specification of our editor, where the user is able to alter both the source and the transformation through editing the view.

In this section we give an introduction of X and Inv. The specification of our editor will be discussed in Section 4.

3.1. The Language $\boldsymbol{X}$

In X, the document designer describes the forward transformation as a function from the source to the view. The good thing is that, given an X program, our system automatically derives the backward updating — how the source shall be updated in respond to the user's editing on the view. In this section we will give an informal overview of X by going through their intended forward transform and giving some examples. How the backward updating is derived will be described in Section 3.2.

3.1.1. An Informal Introduction to X The language X is inspired by similar languages proposed in [13, 10]. Two requirements make X divert from previous works. We need a language construct, Dup, to describe data dependency inside the view. We also adopt a rather different way to describe the semantics of X in order to deal with structural changes — maintaining the consistency of data when the user delete or inserts an element in the view. 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 X constructs is given in Figure 4. The primitive construct GFun takes a pair of functions f and g that satisfy the property that g is the inverse of f *in the range of* f, and vice versa:

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

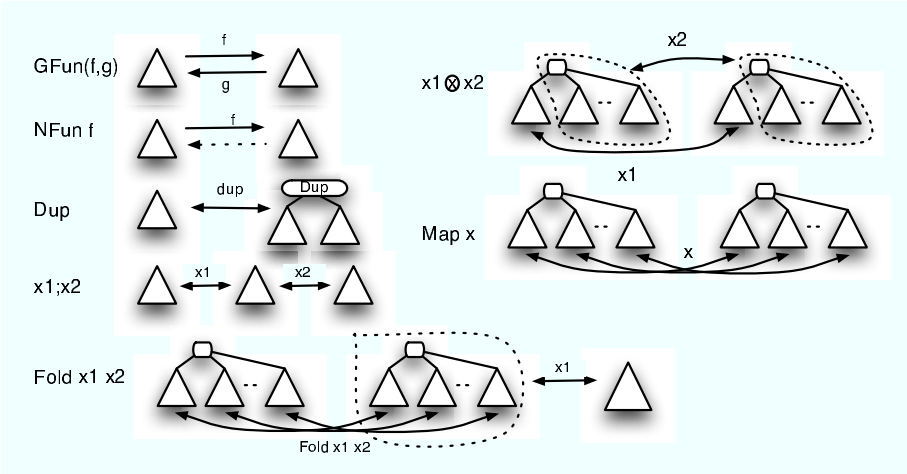

Figure 4. Intuitive Explanation of X Constructs

Here the semicolon denotes function composition, defined by $(f; g)\ a = g\ (f\ a)$. The two properties are satisfied by all Galois-connected pairs of functions, thus the name GFun. The function f is used for the forward transformation, and g for backward updating. The construct NFun, on the other hand, takes any function as the forward tranformation. The resulting view, however, is supposed to be not editable — there is no backward transformation, and any editing action on the view is to be ignored.

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

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

More X primitives defined using GFun are given in Figure 5, where the two functions in the pairs are in fact inverses of each other. Some of them take an extra index as an argument. In such cases they can be considered as macros defining a collection of functions, since those index cannot be results of other computation. Another interesting transformation is given by

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

where the function $sortT$ sorts the subtrees of the root according to the labels of each subtree. It is clear that $sortT$ is not invertible, but $sortT$ and $sortT$ do satisfy the properties INV1 and INV2.

The transformation constX v maps any source to a constant view. The transformation numberX computes the number of children of the source tree. They are

defined respectively by:

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

where *length* computes the length of a list and *show* prints a number into a string. Note that one can turn any function f into a transformation $\mathsf{NFun}\ f$, although its ability to update the source by editing the view will be hindered.

The construct Dup takes a tree and produces two copies of the input, connected by a special node labelled Dup. The Dup operator is the only means in X to specify 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 $x_1 \mathbin{\hat{;}} x_2$ informally means "do x_1, then do x_2". The product construct $x_1 \otimes x_2$ behaves similar to products in ordinary functional languages, apart from that we are working on trees rather than pairs. The input tree is sliced into two parts: the left-most child, and the root plus the other children. The transform x_1 is applied to the left-most child, while x_2 is applied to the rest. The result is then combined together.

As an example, consider the following transformation.

$$\mathsf{Dup} \mathbin{\hat{;}} (\mathsf{numberX} \otimes \mathsf{idX})$$

It maps the input tree to a new tree consisting the number of children of the input tree together with and a copy of the input tree. Although the number shown in the new tree is not editable because the transformation numberX is defined in terms of NFun, its value can be automatically changed if we remove or add a child of the copied tree in the view.

The combinator $\mathsf{If}\ p\ x_1\ x_2$ applies the transform x_1 to the input if the input satisfies the predicate p. Otherwise x_2 is applied. For instance, we may write

$$\mathsf{If}\ (\lambda c.\, \mathit{sumtree}\ c > 10)\ \mathsf{Dup}\ \mathsf{idX}$$

to duplicate the source tree if the sum of all the node values is greater than 10, and keep it unchanged otherwise.

The combinators Map and Fold processes the input tree recursively. The forward transform of $\mathsf{Map}\ x$ applies the transformation x to all subtrees of the given tree, leaving the root label unchanged. The transform $\mathsf{Fold}\ x_1\ x_2$ is defined like a fold on rose trees. The transform x_2 is applied to leaves, x_1 to internal nodes. Its forward transform is given by

$$\begin{array}{lcl} \mathsf{Fold}\ x_1\ x_2\ (\mathtt{N}\ c\ [\,]) & = & x_2\ (\mathtt{N}\ c\ [\,]) \\ \mathsf{Fold}\ x_1\ x_2\ (\mathtt{N}\ c\ cs) & = & ((\mathsf{Map}\ (\mathsf{Fold}\ x_1\ x_2)) \mathbin{\hat{;}} x_1)\ (\mathtt{N}\ c\ cs) \end{array}$$

In the base case, we simply apply x_2 to the leaf. In the recursive case, $\mathsf{Fold}\ x_1\ x_2$ is applied to all subtrees of the input tree, before x_1 is applied to the result, thus the use of sequencing.

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

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

- $\mathsf{sinkPivotX}\ i$ moves the leftmost subtree one leve down, such that it becomes the first child of the ith subtree in the result. $\mathsf{liftPivotX}\ i$ does the inverse.

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

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

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

Note that f^{-1} is a partial function. If the input is not in its domain, the error would be caught by the implementation and an error message is displayed.

- $\mathsf{exchangeX}$ exchanges the root with the node of the leftmost child tree that has no child.

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

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

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

- $\mathsf{replaceHoleX}\ t$ 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

3.1.2. Examples The following X programs define some useful transformations on trees. The transformation insertX v inserts some document v as the leftmost child of the root, deleteX deletes the leftmost child using, and modifyRootX n overwrites the label of the root to n.

$$\begin{array}{lcl}
\mathsf{insertX}\ v & = & \mathsf{insertHoleX} \mathbin{\hat{;}} (\mathsf{replaceHoleX}\ v \otimes \mathsf{idX}) \\
\mathsf{deleteX} & = & (\mathsf{constX}\ \Omega \otimes \mathsf{idX}) \mathbin{\hat{;}} \mathsf{deleteHoleX} \\
\mathsf{modifyRootX}\ n & = & \mathsf{insertX}\ (\mathtt{N}\ n\ [\,]) \mathbin{\hat{;}} \mathsf{exchangeX} \mathbin{\hat{;}} \mathsf{deleteX}
\end{array}$$

A path is a sequence of non-negative integers $[a_1, a_2, \ldots, a_n]$, denoting the subtree obtained by going into the a_1-th child of the root, then into the a_2-th child, and so on. For example, $[\,]$ denotes the root node (or the entire tree), and $[0]$ denotes the first child of the root. We define a collection of functions indexed by paths as below:

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

The transformation applyX p x applies a 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. The transformation fromPivotXP p moves the pivot — the first child of the root, to the path p, while toPivotXP moves the subtree at path p to the pivot position. Although they are defined like higher order transformations, they can be considered as macros, since their arguments are always of finite sizes and cannot be computed from other transformations. With them the transformation $moveX\ p_1\ p_2$, moving the subtree at p_1 such that it ends up at the path p_2, can be defined by:

$$\mathsf{moveX}\ p_1\ p_2 = \mathsf{toPivotXP}\ p_1 \mathbin{\hat{;}} \mathsf{fromPivotXP}\ p_2$$

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.

$$\begin{array}{l}
\mathsf{sortX} \mathbin{\hat{;}} \mathsf{Dup} \mathbin{\hat{;}} \\
\mathsf{applyX}\ [0]\ (\mathsf{modifyRootX}\ \mathtt{Index} \mathbin{\hat{;}} \mathsf{Map}\ \mathsf{keepX}) \mathbin{\hat{;}} \\
\mathsf{moveX}\ [0]\ [0, 0] \mathbin{\hat{;}} \mathsf{hoistX}\ \mathtt{Dup}
\end{array}$$

We sort the address book according to person's names by sortX, duplicate the address book, keep only the name (first child) for each person in the duplicated address book using Map keepX and change the root name to be `Index` using modifyRootX `Index`. The index is lifted the list of names to beginning, before we remove remove the label `Dup` using hoistX `Dup`.

3.2. Semantics

In the previous section we gave an informal introduction of the language X, focusing on its forward transformation. We have not yet discussed how the backward transformation — generating a new source when the view is edited, is specified. In this section we will present a more formal definition of the constructs in X.

In [15], the authors defined a programming language, Inv, in which the programmers can define injective functions only. In [14], the language was given an extended semantics, where every Inv expression of type $A \to B$ induces a binary relation between A and B, mapping the edited values in A to a reasonable choice of source in B.

We call the type of source documents S and that of views V. They are both embedded in a generic tree (i.e., G in Section 2), but we nevertheless distinguish them for clarity. The trick is that every X construct is embedded as an Inv expression denoting, in the injective semantics, a function of type $S \to (S \times V)$ that takes a source and produces a pair consisting of a copy of the given source together with the view. The function is apparently injective because the source is kept in the output. Its inverse, of type $(S \times V) \to S$, maps the original source and its corresponding view back to the source. In the extended semantics, however, when given the original source and an *edited* view, the Inv expression magically produces an updated source consistent with the transform.

In Section 3.2.1 we give a brief introduction to Inv, and in Section 3.2.2 we will discuss how it handles duplication and structural alignment. The discussion here is nevertheless not complete, and the reader is referred to [14] for a more complete account. The embedding is presented in Section 3.2.3. For a transformation to be used in an editor, it has to satisfy a set of constraints called *bidirectionality*. This is given in Section 3.2.4, where we also show that all X transformations are bidirectional.

3.2.1. The Language Inv This language can deal with a range of values defined by the syntax below.

$$
\begin{array}{lcl}
V & = & A \mid V^{+} \mid V^{-} \mid [V] \mid (V \times V) \mid \mathsf{Tree} \\
\mathsf{Tree} & = & \mathtt{N}\ A\ [\mathsf{Tree}] \\
[a] & = & [\ ] \mid a : [a] \\
A & = & \mathit{String} \mid *\mathit{String} \mid \top
\end{array}
$$

For the purpose of this paper, the string is the only atomic type. We can construct pairs, lists, and trees. The *editing tags* $(_)^{+}$, $(_)^{-}$, $*(_)$, and $\top$ are used to record the action performed by the user of the editor. The $*(_)$ tags applies to atomic values (strings). 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 $(_)^{-}$, keeping note that it ought to be deleted but we temporary leave it there for further processing. Values containing any of the tags are called *tagged*, otherwise they are *untagged*.

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

$$
\begin{array}{lcl}
[\![\mathit{nil}]\!]\ () & = & [\,] \\
[\![\mathit{cons}]\!]\ (a, x) & = & a : x \\
[\![\mathit{node}]\!]\ (a, x) & = & \mathsf{N}\ a\ x \\
[\![\mathit{id}]\!]\ a & = & a \\
[\![p?]\!]\ a & = & a \ \text{ if } p\ a
\end{array}
$$

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

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

$$
\begin{array}{lcl}
[\![\delta]\!]\ a & = & (a, a)
\end{array}
$$

$$
\begin{array}{lcl}
[\![f; g]\!]\ x & = & [\![g]\!]\ ([\![f]\!]\ x) \\
[\![f \times g]\!]\ (a, b) & = & ([\![f]\!]\ a, [\![g]\!]\ b) \\
[\![f \cup g]\!] & = & [\![f]\!] \cup [\![g]\!], \\
 & & \text{if } \mathit{dom}\ f \cap \mathit{dom}\ g = \mathit{ran}\ f \cap \mathit{ran}\ g = \emptyset
\end{array}
$$

$$
\begin{array}{lcl}
[\![\mathit{prim}(f, g)]\!] & = & f \\
[\![\mathit{prim}(f, g)^{\smile}]\!] & = & g \\
[\![f^{\smile}]\!] & = & [\![f]\!]^{\circ} \quad f \text{ not } \mathit{prim} \\
[\![\mu F]\!] & = & [\![F\ \mu F]\!]
\end{array}
$$

Figure 6. The Language Inv and its Semantics when Restricted to Untagged Values.

Figure 6 shows a subset of Inv we need for this article. The non-terminal P denotes predicates, and by Inv_V we denote the union of Inv expressions with the set of variable names. The full semantics of Inv is discussed in [14]. For the purpose of this paper, it suffices to think of each construct as defining a relation which, when its domain and range are restricted to untagged types, reduces to an injective partial function. When the input is tagged, on the other hand, through the trick described in [14], the relation maps the input to a reasonable updated result.

The language Inv deals with a wider range of datatypes, including unit, pairs, lists, and trees. A list is built by constructors *nil* and *cons*, where the input of *nil* is restricted to unit type. The constructor *node* produces a tree from a pair consisting of a label and a list of subtrees. One can also produce a fresh empty list or a string using *dupNil* or *dupStr*.

The function *id* is the identity function, the unit of composition. Given a predicate p, we denote by $p?$ a subset of *id* returning only those input satisfying p. Function composition is denoted by semicolon. Union of relations, modelling conditional branches, is simply defined as set union, with a restriction that the two relations have disjoint domains and ranges. The product $(f \times g)$ takes a pair and applies f and g to the two components respectively. The functions *swap*, *assocl* and *assocr* distributes the components of the input pair. 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:

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

In the injective semantics $subr\ (a,(b,c)) = (b,(a,c))$ and $trans\ ((a,b),(c,d)) = ((a,c),(b,d))$.

The *converse* of a relation R is defined by

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

In the injective semantics, the *reverse* operator $(_)^{\smile}$ corresponds to converses on relations. 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 operator distributes into composition, products and union by the following rules, all implied by the semantics definition $[\![f^{\smile}]\!] = [\![f]\!]^{\circ}$:

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

In particular, the reverse of two functions composed is their reverses composed backwards.

The δ operator is worth our attention. It generates a copy of its argument. We restrict the use of δ to atomic strings only. Its reverse is a partial function accepting only pairs of identical elements. Therefore, the inverse of duplication is equality test.

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 $unzip :: [(A \times B)] \to ([A] \times [B])$ transposes a list of pair to a pair of lists. Their definitions in Invare exactly the point-free counterpart of their usual definitions:

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

Finally, the *prim* construct is a backdoor kept for X only. Both the forward and backward transform are given explicitly as arguments to *prim*.

3.2.2. Duplication and Alignment One of the motivation behind the development of Inv was to study the handling of duplication and structural alignment.

As described in the previous section, $\delta^{\smile}$ is a partial function performing equality test. The two components in the pair are compared, and one of the is returned only when they are identical. When the user edits an atomic value, the action is

recorded by a $*(_)$ tag. In the extended semantics, we generalise δ such that it recognises the tag:

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

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.

Still, the δ operator handles atomic values only. To unify structural data, we have to synchronise their shapes as well. Let $zip = unzip^{\smile}$. This partial function of type $([A] \times [B]) \to [(A \times B)]$ zips together two lists only if they have the same length. In general zipping functions are useful as constraints on shape. In an editor, however, the user may add or delete elements in one of the list. The edited lists may not have the same lengths, and we have to somehow zip them and still align the paired elements together.

One of the main achievement of [14] is that, using the same definition of *unzip* above with a small amount of annotations, *zip* in the extended semantics knows how to zip together two lists when they contain inserted or deleted elements. For example, $unzip\ [(1, \mathtt{a}), (2, \mathtt{b}), (3, \mathtt{c})]$ yields $([1, 2, 3], [\mathtt{a}, \mathtt{b}, \mathtt{c}])$. If we label one element with a delete tag, $zip\ ([1, 2^{-}, 3], [\mathtt{a}, \mathtt{b}, \mathtt{c}])$ yields $[(1, \mathtt{a}), (2, \mathtt{b})^{-}, (3, \mathtt{c})]$ — the corresponding element is deleted as well. If we insert an element, say $([1, 2, 3], [\mathtt{a}, \mathtt{b}, \mathtt{d}^{+}, \mathtt{c}])$, zipping them together yields $[(1, \mathtt{a}), (2, \mathtt{b}), (\top, \mathtt{d})^{+}, (3, \mathtt{c})]$. An unconstrained value is invented and paired with the newly inserted $\mathtt{d}$, and might later be further constrained by δ or other structural constraints[1].

With *zip* we can define a generic duplication operator. Let dup_a be a type-indexed collection of functions, each having type $a \to (a \times a)$:

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

In particular, to duplicate a list we shall duplicate each element and unzip the resulting list of pairs. The use of *unzip* synchronises the shape of the two lists in the backward updating. In the discussion later we will omit the type subscript.

3.2.3. Embedding X in Inv As mentioned in the previous sections, each X transformation is embedded as an Inv expression having type $S \to (S \times V)$ — a program taking 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 their forward transform. The backward updating inferred from their Inv definition turn out to be exactly what we expect.

We denote by $\lceil _ \rceil$ the embedding of X into Inv. The primitives GFun and NFun are implemented using the backdoor *prim*. The input is first duplicated by *dup*. One of

the duplicated tree is then passed to $prim\ (f, g)$ and $prim\ (f, \lambda x.\top)$, respectively. In both cases the forward transform is given by f. For GFun, the backward transform is specified by the programmer. For NFun, we simply throw away the edited view by a constant function always producing an undefined value. The reverse of *dup* then unifies it with the cached source.

$$\begin{array}{lcl} \lceil \mathsf{GFun}(f,g) \rceil & = & dup; (id \times prim\ (f,g)) \\ \lceil \mathsf{NFun}\ f \rceil & = & dup; (id \times prim\ (f, \lambda x.\top)) \end{array}$$

The Dup construct is defined by:

$$\begin{array}{lcl} \lceil \mathsf{Dup} \rceil & = & dup; (id \times dup; futatsu; mkRoot) \\ \quad \textbf{where}\ \ futatsu & = & (id \times dupNil; cons); cons \\ \qquad\qquad mkRoot & = & dupStr\ \mathtt{Dup}; swap; node \end{array}$$

Here *dup* is called twice, the first time to produce the cached input, the second time for the actual duplication. The expression *futatsu*; *mkRoot* simply produces a tree with root labelled Dup from a pair of trees.

Assume that we have two embedded transformation $\lceil f \rceil :: A \to (A \times B)$ 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 get rid of the intermediate value of type B, but there is no information-losing constructs in Inv. The solution is to notice that the reverse of $\lceil f \rceil$ takes a pair $(A \times B)$ and reduces it to A. Composition of X programs is thus defined by:

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

The application of $\lceil f \rceil^{\smile}$ is only in the specification — in the implementation we can simply throw away the intermediate value. However, it is instructive to look at its effect 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 an updated source a', we will 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'. Finally we send (a, b') to $\lceil f \rceil^{\smile}$, computing the updated a'. This exactly how backward updating for composition is defined in [13, 10], which turns out to follow naturally from our definition.

Products in X is defined by:

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

The function *slice* is given by $slice\ (\mathtt{N}\ a\ (t : x)) = (t, \mathtt{N}\ a\ x)$. 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 on it to determine which branch was taken.

Let f be an X transformation. The Inv expressions *trList* f and *trSubtrees* f respectively applies $\lceil f \rceil$ to every element of a list and the list of children of a tree, while keeping a copy of the input.

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

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, $\mathit{unzip}^{\smile}$ takes care of the alignment when the user inserts to or deletes from the trasnformed list. With the two helper functions, Map can be defined by:

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

Finally, Fold is defined recursively as below:

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

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

3.2.4. Bidirectionality To model the behaviour of our editor, for every transform x in X we define a pair of functions:

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

where $\mathit{snd}\ (s, v) = v$, the function *norm* removes the tags in the tree and produces a normal form. The function $\phi_x :: S \to V$ defines the transformation from the source to the view. The function $\lhd_x :: (S \times V) \to S$ takes the original source and an edited view, and returns an updated source. In [10] they are called *get* and *put* respectively.

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. However, how can we be sure that we do not need to repeat the *put-get* cycle again?

We call a transformation x *bidirectional* if the following two properties hold:

$$\begin{array}{ll} \text{GET-PUT-GET}: & \phi_x(s \lhd_x v) = v \quad \text{where } v = \phi_x s \\ \text{PUT-GET-PUT}: & s' \lhd_x (\phi_x s') = s' \quad \text{where } s' = s \lhd_x v \end{array}$$

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 $\lhd_x$. 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. The two properties together ensures that when the user alters the view, we need to perform only one *put* followed by one *get*. No further updating is necessary.

Let relation composition be defined by $R; S = \{(a, c) \mid \exists b \cdot (a, b) \in R \wedge (b, c) \in S\}$, *notag* a partial function maps the input to itself if it does not contain tags, and $dom\ R = \{(a, a) \mid \exists b \cdot (a, b) \in R\}$, the operator taking the domain of a relation. An important result in [14] is that the following properties hold:

$$\begin{aligned} notag; [\![x]\!]; [\![x^{\smile}]\!]; norm &= notag; dom\ [\![x]\!] \\ [\![x^{\smile}]\!]; norm; [\![x]\!]; [\![x^{\smile}]\!]; norm &\subseteq [\![x^{\smile}]\!]; norm \end{aligned}$$

Further more, the inclusion in the second property becomes an equality for a certain class of Inv expressions. From the two properties above, the GET-PUT-GET and PUT-GET-PUT laws follow immediately.

Theorem 1 (Bidirectionality of X)
Any transformation described in X is bidirectional. □

Remark: The following GET-PUT and PUT-GET properties are required in [13, 10] to hold for arbitrary v and s':

GET-PUT: $\phi_x(s \lhd_x v) = v$ for any view v
PUT-GET: $s' \lhd_x (\phi_x s') = s'$ for any source s'

However, as shown in [10], this would imply that ϕ_x is injective, which in turn would imply difficulties dealing with duplication. A reasonable definition of ϕ_x where x duplicates data shall not be injective, since modifying either of the duplicated data should yield the same source. (**End of remark**)

4. The Programmable Editor

Our editor serves as a presentation-oriented (view-oriented) environment supporting interactive development of structured documents. It allows users to develop structured documents in a WYSIWYG (what you see is what you get) manner, and automatically produces the three components of a structured document.

4.1. Editing Operations

We consider the following editing operations.

$$\begin{aligned} E \ ::= \ & \mathsf{InsertE}\ p\ v \\ \mid\ & \mathsf{DeleteE}\ p \\ \mid\ & \mathsf{CopyE}\ p_1\ p_2 \\ \mid\ & \mathsf{MoveE}\ p_1\ p_2 \\ \mid\ & \mathsf{FieldEditE}\ p\ l \\ \mid\ & \mathsf{DuplicateE}\ p \\ \mid\ & \mathsf{TransformE}\ p\ x \end{aligned}$$

They are standard except for the last two operators. InsertE p v, for instance, inserts a tree v as the first child of the node at path p, and FieldEditE p l modifies the label of the node at path p to l. The last two new editing operators, DuplicateE p and TransformE p x, are the special features in our editor: DuplicateE p duplicates the tree at path p and the two trees should be kept identical, and TransformE p x applies a bidirectional transformation x to the tree at path p.

The state of the editor is a triple

$$\mathcal{S} = (c, x, a)$$

where c and a denote the concrete source and the view respectively, and x denotes a bidirectional transformation. Each state $\mathcal{S} = (c, x, a)$ holds the following SYNC property.

$$\begin{array}{rcl} a & = & \phi_x\ c \\ c & = & c \lhd_x a \end{array}$$

This SYNC property expresses the relationship among the three elements in a state, and the bidirectionality of x ensures an automatic adjustment among the three elements in case some of them is modified. To be precise, let (c, x, a) be a given state.

- If c changes to c', the new state is $(c', x, \phi_x\ c)$;
- If x changes to x', the new state is $(c, x', \phi_{x'}\ c)$;
- If a changes to a', the new state is $(c \lhd_x a', x, \phi_x\ (c \lhd_x a'))$.

We define two functions for describing the above adjustments.

$$\begin{array}{rcl} \mathcal{A}_{cx}\ (c, x, a) & = & (c, x, \phi_x\ c) \\ \mathcal{A}_a\ (c, x, a) & = & \mathcal{A}_{cx}\ (c \lhd_x a, x, a) \end{array}$$

$\mathcal{A}_{cx}$ adjust the editor state when c or x changes, while $\mathcal{A}_a$ adjust the editor state when a changes.

The operational semantics of the editing operations is given in Figure 7. Each editing operation is a state transformer with two steps; transforming some components of the editor state and then adjusting the state to meet the SYNC property. Given the state (c, x, a), the operator InsertE p v is (1) to insert a tree v to the view a by a general tree insertion function insert and accordingly to change the path expressions in the transformation x so that the nodes at these paths refer to the same ones, and then (2) to adjust the state by $\mathcal{A}_a$. Here, insP x p is a function to "increase" some node number in some paths in x. Let $p = p_1 \mathbin{+\!\!+} [a]$, and p' be a path expression in x satisfying $p' = p_1 \mathbin{+\!\!+} [b] \mathbin{+\!\!+} p_2$ and $b > a$, then p' will be changed to $p_1 \mathbin{+\!\!+} [b+1] \mathbin{+\!\!+} p_2$. Other editing operations like deleteE, copyE, moveE, and fieldEditE are defined similarly. The duplicateE and transformE are two editing operations that change the transformation x. Thanks to the SYNC property of the editor state, their semantics is very clear.

$$
\begin{array}{lcl}
(\mathsf{InsertE}\ p\ v)\ (c, x, a) & \rightarrow & \mathcal{A}_a\ (c,\ \mathsf{incP}\ x\ p,\ \mathsf{insert}\ p\ v\ a) \\
(\mathsf{DeleteE}\ p)\ (c, x, a) & \rightarrow & \mathcal{A}_a\ (c,\ \mathsf{decP}\ x\ p,\ \mathsf{delete}\ p\ a) \\
(\mathsf{CopyE}\ p_1\ p_2)\ (c, x, a) & \rightarrow & \mathcal{A}_a\ (c,\ \mathsf{incP}\ x\ p_2,\ \mathsf{copy}\ p_1\ p_2\ a) \\
(\mathsf{MoveE}\ p_1\ p_2)\ (c, x, a) & \rightarrow & \mathcal{A}_a\ (c,\ \mathsf{incP}\ (\mathsf{decP}\ x\ p_1)\ p_2,\ \mathsf{move}\ p_1\ p_2\ a) \\
(\mathsf{FieldEditE}\ p\ l)\ (c, x, a) & \rightarrow & \mathcal{A}_a\ (c,\ x,\ \mathsf{fieldEdit}\ p\ l\ a) \\
(\mathsf{DuplicateE}\ p)\ (c, x, a) & \rightarrow & \mathcal{A}_{cx}\ (c,\ (x\ \hat{;}\ \mathsf{applyX}\ p\ \mathsf{Dup}),\ a) \\
(\mathsf{TransformE}\ p\ x')\ (c, x, a) & \rightarrow & \mathcal{A}_{cx}\ (c,\ (x\ \hat{;}\ \mathsf{applyX}\ p\ x')),\ a)
\end{array}
$$

Figure 7. The Operational Semantics of the Editing Operations

Note the difference between the two forms of editing operations in our editor: editing operations directly manipulating views and editing operations formalized as bidirectional transformations between views. Considering the insertion operator, we have two forms:

$\mathsf{InsertE}\ p\ v$
$\mathsf{TransformE}\ p\ (\mathsf{insertX}\ v)$

The former inserts a tree to the view and propagates this change to other places of the view, while the latter performs an *independent* insertion on the view, causing no changes elsewhere. Note also that not any editing sequence is valid in our system. For example, the view produced by a restrictive primitive transformation is not editable by InsertE. However, it can be modified by an independent editing operation.

4.2. Deriving Structured Documents

This section explains how to produce the three components for a structured document after a sequence of editing operations. Recall that in Section 2 the three components of a structured document are the document type, the document source, and the transformation.

The first two elements of the final editor state (c, x, a) almost give the source document and the transformation we want to have. What is remained to do is to find a suitable document type to structure c and to make x be a transformation accepting the typed document source. We ask the users to provide a type for the view (see an example in Section 5), and we infer the type for the document source from the transformation.

Our derivation algorithm is depicted in Figure 8, which accepts as input a stable editor state (c, x, a) and a type A (in T) that are expected for the view a, and returns a typed document source c' together with its type C. The algorithm consists of three steps: (1) deriving a typed view from the generic view a with an expected type A; (2) inferring a type C for the document source from x and A, and (3)

Input:
(c, x, a): a stable editor state where c and a are of type G
A: a specific type in T, which is expected for a
Output:
C: a specific type satisfying $x : C \leftharpoonup A$
c': a document in C satisfying $c = t2g\ c'$
Algorithm:
(1) $a' := g2t\ a\ A$
(2) infer C such that $x : C \leftharpoonup A$
(3) $c' := g2t\ c\ C$

Figure 8. Algorithm for Deriving Structured Document Components from an Editor State

deriving a typed document source c' from c and C. It should be noted that the first step is only for validation, and a' is not used for producing c' and C.

The second step is quite simple due to simplicity of the language X, which does not have local variables. Since our transformations are built up upon primitive transformations in terms of $\mathsf{GFun}\ (f, g)$ and $\mathsf{NFun}\ f$, we can derive the type of the source from that of the view, provided the types for functions used in the primitive transformations are given. We consider deriving types in the simple form of T in Section 2 for both document sources and views. Figure 9 summarizes the inference rules for inferring the type for c from the type of a in the stable state (c, x, a). We write $x : C \leftharpoonup A$ to denote the judgment that the type C of the source document can be derived from the type A of the view, according to the transformation x. Rules (NFun) and (GFun) are two base cases showing that the type C of the source document can be derived from the types of the functions used in defining primitive transformations. Rule (Dup) and Rule (Seq) are straightforward. Rules (Prod1), (Prod2) and (Prod3) give three cases for production. (Prod1) and (Prod2) imply higher priority on obtaining list structures than tuples. Rules (Cond) and (Map) are obvious. In Rule (Fold), we define a recursive type C for the document source.

4.3. Infinite Undo

An additional advantage of bidirectional transformations in our editor is the ability to implement infinite numbers of operations of undo. The following set of equations indicate that for any editing operations, there always exists another editing

$$\frac{f : C \rightarrow A}{\mathsf{NFun}\ f : C \leftarrow A} \qquad (\textsc{NFun})$$

$$\frac{f : C \rightarrow A \quad g : A \rightarrow C}{\mathsf{GFun}\ (f,\ g) : C \leftarrow A} \qquad (\textsc{GFun})$$

$$\mathsf{Dup} : A \leftarrow \mathsf{Node}\ \mathit{Dup}\ (A, A) \qquad (\textsc{Dup})$$

$$\frac{x_1 : C \leftarrow B \quad x_2 : B \leftarrow A}{x_1 \mathbin{\hat{;}} x_2 : C \leftarrow A} \qquad (\textsc{Seq})$$

$$\frac{\begin{array}{c} x_1 : C_1 \leftarrow A_1 \\ x_2 : \mathsf{Node}\ n\ C_2 \leftarrow \mathsf{Node}\ n\ A_2 \\ C_2 = [C_1] \end{array}}{x_1 \otimes x_2 : \mathsf{Node}\ n\ [C_1] \leftarrow \mathsf{Node}\ n\ (A_1, A_2)} \qquad (\textsc{Prod1})$$

$$\frac{\begin{array}{c} x_1 : C_1 \leftarrow A_1 \\ x_2 : \mathsf{Node}\ n\ C_2 \leftarrow \mathsf{Node}\ n\ A_2 \\ C_2 \neq [C_1] \end{array}}{x_1 \otimes x_2 : \mathsf{Node}\ n\ (C_1, C_2) \leftarrow \mathsf{Node}\ n\ (A_1, A_2)} \qquad (\textsc{Prod2})$$

$$\frac{\begin{array}{c} x_1 : C \leftarrow A \\ x_2 : \mathsf{Node}\ n\ [C] \leftarrow \mathsf{Node}\ n\ [A] \end{array}}{x_1 \otimes x_2 : \mathsf{Node}\ n\ [C] \leftarrow \mathsf{Node}\ n\ [A]} \qquad (\textsc{Prod3})$$

$$\frac{x_1 : C \leftarrow A \quad x_2 : C \leftarrow A \quad p : C \rightarrow \mathit{Bool}}{\mathsf{If}\ p\ x_1\ x_2 : c : C \leftarrow a : A} \qquad (\textsc{Cond})$$

$$\frac{x : C \leftarrow A}{\mathsf{Map}\ x : \mathsf{Node}\ n\ [C] \leftarrow \mathsf{Node}\ n[A]} \qquad (\textsc{Map})$$

$$\frac{\begin{array}{c} x_2 : \mathsf{String} \leftarrow \mathsf{Node}\ n\ A \\ x_1 : \mathsf{Node}\ m\ [\mathsf{Node}\ n\ A] \leftarrow \mathsf{Node}\ n\ A \\ C = \mathsf{String} \mid \mathsf{Node}\ m\ [C] \end{array}}{\mathsf{Fold}\ x_1\ x_2 : C \leftarrow \mathsf{Node}\ n\ A} \qquad (\textsc{Fold})$$

Figure 9. An Inference System for Deriving Source Document Type

operation to recover the state.

$$(\mathsf{DeleteE}\ p)\ ((\mathsf{Insert}\ p\ v)\ s) = s$$
$$(\mathsf{InsertE}\ p\ (s|p))\ ((\mathsf{DeleteE}\ p)\ s) = s$$
$$(\mathsf{DeleteE}\ p_2)\ ((\mathsf{CopyE}\ p_1\ p_2)\ s = s$$
$$(\mathsf{InsertE}\ p_1\ (s|p_1))\ ((\mathsf{DeleteE}\ p_2)\ ((\mathsf{MoveE}\ p_1\ p_2)\ s)) = s$$
$$(\mathsf{FieldEditE}\ p\ (\mathsf{root}(s|p)))\ ((\mathsf{FieldEditE}\ p\ n)\ s) = s$$
$$(\mathsf{undoX})\ ((\mathsf{DuplicateE}\ p)\ s) = s$$
$$(\mathsf{undoX})\ ((\mathsf{TransformE}\ p)\ s) = s$$

Here $s|p$ denotes the subtree in the view s at the path p, and $\mathsf{root}\ v$ returns the label of the root node of the tree v. undoX is a new editing operation for undoing the last transformation. Its semantics can be defined by

$$(\mathsf{undoX})\ (c, x, a) = \mathcal{A}_{cx}(c, \mathsf{deleLast}\ x, a)$$

where $\mathsf{deleLast}$ is to delete the last added transformation.

These equations enable us to implement a sequence of undo operations by remembering a sequence of editing operations (for recovering the editor states) rather than a sequence of editor states. This saves much space, making it possible to implement infinite numbers of undo operations.

5. Editing = Developing

We view the development of structured documents as the process of constructing a triple (T, D, X) meeting the requirements the designer had in mind. We have implemented in Haskell a prototype editing system for supporting this development. The main purpose of this prototype system is for testing the idea, and the editor has a simple user interface as seen in Figure 10: the left is the source, the middle part is the view on which editing operations can be applied, the right 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 are not really necessary to be exposed to users in the interface; we do it just for easy testing.

We demonstrate how our editor works by going through the development of the address book in the introduction. From scratch, we start with an empty view with only one node labeled `"Root"`:

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

In the demonstration to follow, we will construct, via interaction with the editor, the triple (T, D, X) like those (but in different notations) in Figure 1 such that the resulting view looks like that in Figure 2.

The node or subtree in focus, on which the user performs editing operations, is selected by a cursor. Here, for simplicity, we use a path to denote the subtree we select.

The complete list of operations the user can perform on the focused subtrees has been given in Section 4. We will show how all these editing operations are used for developing our address book.

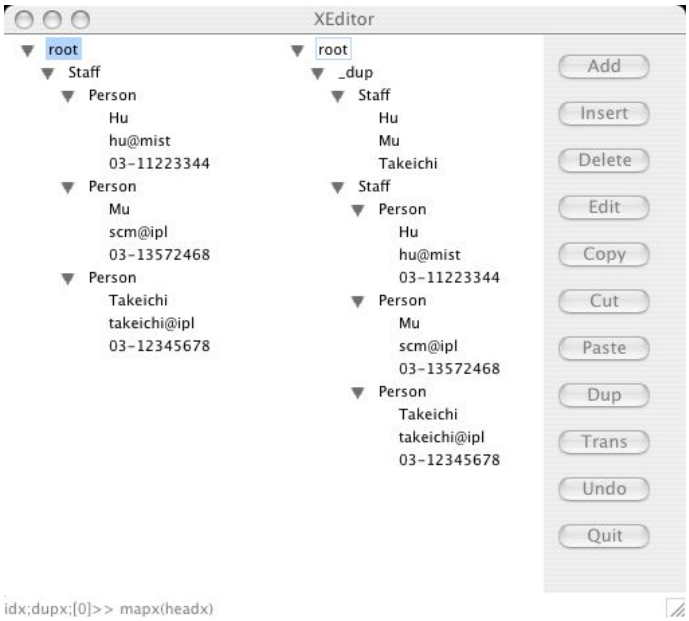

Figure 10. A Snapshot of the Prototyped Editor

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

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

and, by the InsertE operation, we insert a name and some contacts 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 "Masato Takeichi" []],
    N "Email" [N "takeichi@acm.org" []],
    N "Tel" [N "+81-3-5841-7430" []]]]
```

We may continue to add another person's contacts by copying the subtree rooted at the path [1] using the CopyE operation. The copied tree becomes a sibling of the original:

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

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

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

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

It should be noted that we are editing both the source document and the view, though we are not quite aware of this fact so far. The transformation X, is currently simply the identity transformation idX. Now suppose we want to sort persons according to their names, by selecting all the persons and apply the sortX transformation on it via the editing operation TransformE. The result looks like

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

What is sorted is the view. The source remains the same, while the transformation sortX now looks like the function that performs the sorting.

Next, we want to make an index of names of people in the address book. To do so, we first make a copy of the address book by the DuplicateE operation:

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

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

and then apply the transformation keepX via 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 "Zhenjiang Hu" []],
    N "Name" [N "Masato Takeichi" []]]
 N "Addrbook"
   [N "Person"
     [N "Name" [N "Zhenjiang Hu" []],
      ... ],
    N "Person"
     [N "Name" [N "Masato Takeichi" []],
      ...]]]
```

It should be remarked again that the duplication is one of the most important features of our system. It is different from the copy operation, which we performed just now to add a new person in the address book. Copied data are independent from each other. On the other hand, the duplicate operation indicates that the subtree and its duplicate should be synchronized. In this example, deletion, insertion, or modification of a person's information at one side causes corresponding change on the other side, unless we explicitly inform the editor to perform the editing operations independently.

The keepX transformation used in the TransformE operation in the above, for example, is such an independent transformation. When it was applied to the subtree at [1] to extract the names, the main address book at [2] remains unchanged. On the other hand, if we insert the following entry (by the InsertE operation)

```
N "Person"
  [N "Name" [N "Shin-Cheng Mu" []],
   N "Email" [N "scm@mist.i.u-tokyo.ac.jp" []],
   N "Tel" [N "+81-3-5841-7411" []]]
```

to the "Addrbook" subtree at the path [2] as its last child, the name "Shin-Cheng Mu" will automatically appear in the index of names, resulting in:

```
N "Dup"
[N "Index"
   [N "Name" [N "Zhenjiang Hu" []],
    N "Name" [N "Shin-Cheng Mu" []]],
    N "Name" [N "Masato Takeichi" []]
 N "Addrbook"
   [N "Person"
```

```
  [N "Name" [N "Zhenjiang Hu" []],
   ... ],
N "Person"
  [N "Name" [N "Shin-Cheng Mu" []],
   ...]],
N "Person"
  [N "Name" [N "Masato Takeichi" []],
   ...]]]
```

Note also 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 the `"Addrbook"` and the names under the `"Index"` sorted.

Finally, after renaming the root name to `Addrbook`, we tell the system that the type of the view has the type of *ADDRVIEW*, which is defined by

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

and our system automatically returns the triple (T, D, X) (written in our notation) similar to those in Figure 1.

We summarize the important features of our programmable editor as follows.

- Our editor is presentation-oriented (view-oriented), with which the developer can directly edit the view, the exact display of the document. This WYSIWYG style is more friendly than existing editors. Those with little knowledge about XML will feel easy to use this system to develop their structured documents.
- Our editor allows simple description of data dependency in the view by the DuplicateE operation, and provides an efficient solution to keep consistency of the data in the view. As far as we are aware, this is the first structured document editor with local data synchronization.
- Our editor integrates the three components of a structured document in the view displayed to the user. The source data and the transformation are gradually built while the user edits the view, before the user finally imposes a type on the view.

6. Related Work

There are plenty of XML editors [19], which have been designed and implemented for supporting development of structured documents in XML. Most of them, such as XMLSpy [12], develop structured documents in the order of DTD, document content, and presentation. These kinds of tools cannot effectively support interactive document development, as strongly argued by researchers [7, 21] in the field

of document engineering. Moreover, these tools require developers to have much knowledge about DTD, XML and XSLT. In contrary, our editor provides a single integrated WYSIWYG interface, and requires less knowledge about XML.

The most related system to ours is Proxima [18, 11], a single presentation-oriented generic editor designed for all kinds of XML-documents and presentations. It is very similar to our system; it is also presentation orient and allows description of transformation and computation over view 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 the TreeCalc system [20], a simple tree version of the spreadsheet system, but it does not support structure modification on the view.

Our representation of the editor state by a triple (the document source, and transformation, and the view) is inspired by the work on view-updating [2, 5, 9, 16, 1] 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 that editing operations can modify not only the view but also the query, which is not exploited before. Since our transformation language does not have the JOIN operator, the problem of the costive propagation of deletion and annotation through views [17] does not happen in our case.

During the design of the bidirectional transformation language X, much was learnt from the lenses combinators in [10, 6], where a semantic foundation and a core programming language for bidirectional transformations on tree-structured data are given. The current lens combinators can clearly specify dependency between a source data and a view, but cannot describe dependency *inside a view*. This is not the problem in the context of data synchronization, but 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 makes dependency clearly described. Another very much related language is that given by Meertens [13], which is designed for specification of 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 in [8], where duplication is considered as the inverse of equality check and vice versa. In inverse computation, an inverse function computes an input merely from an output, but in bidirectional transformation, a 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 involved equality check mechanism. It should be interesting to see if inverse transformation with duplication can implement the view updating, and to compare these two approaches. Some attempt has been made in [15, 14].

7. Conclusions

In this paper, we proposed a presentation-oriented editor suitable for interactive development of structured documents. A 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 a key role in the design of our editor. The prototyped system with automatic view updating and infinite undos shows the promise of this approach.

We highlight several research directions in the future. First, we have not addressed in details the issues of optimization of bidirectional transformations in X, which would play an important role in practical and big applications. Second, it should be useful if a good type inference system could be constructed to make it clear which nodes are updatable (modifiable, insertable, and removable) in the editor view. Finally, it would be attractive to look into the possibility of making the existing transformation languages like XSLT to be efficiently bidirectional, instead of designing a new bidirectional language.

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.

Notes

1. In the semantics in [14], there is no $\top$. Instead, a relation may non-deterministically map the input to many outputs, and refined when composed with other relations. However, $\top$ was indeed used in the implementation.

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. F. Bancilhon and N. Spyratos. Updating semantics of relational views. *ACM Transactions on Database Systems*, 6(4):557–575, 1981.
3. R.S. 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. U. Dayal and P. A. Bernstein. On the correct translation of update operations on relational views. *ACM TODS*, 7(3):381–416, 1982.
6. 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.

7. R. Furuta, V. Quint, and J. André. Interactively editing structured documents. *Electronic Publishing Origination, Dissemination, and Design*, 1(1):19–44, 1988.
8. 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.
9. G. Gottlob, P. Paolini, and R. Zicari. Properties and update semantics of consistent views. *ACM Transactions on Database Systems*, 13(4):486–524, 1988.
10. Michael B. Greenwald, Jonathan T. Moore, Benjamin C. Pierc, 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.
11. Johan Jeuring. Implementing a generic editor. In *2nd Workshop on Programmable Structured Documents*, February 2004.
12. Larry Kim. *The Official XMLSPY Handbook*. John Wiley & Sons, 2002.
13. Lambert Meertens. Designing constraint maintainers for user interaction. `http://www.cwi.nl/~lambert`, June 1998.
14. S.C. Mu, Z. Hu, and M. 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.
15. S.C. Mu, Z. Hu, and M. 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.
16. Atsushi Ohori and Keishi Tajima. A polymorphic calculus for views and object sharing. In *ACM PODS'94*, pages 255–266, 1994.
17. 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.
18. Martijn M. Schrage and Johan Jeuring. Xprez: A declarative presentation language for XML. available at `http://www.cs.uu.nl/research/projects/proxima/`, 2003.
19. XML Software. A list of xml editors. See http://www.xmlsoftware.com/editors.html, 2004.
20. 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.
21. L. Villard, C. Roisin, and N. Layada. A XML-based multimedia document processing model for content adaptation. In *8th International Conference on Digital Documents and Electronic Publishing, LNCS*, September 2000.