A Programmable Editor for Developing Structured Documents based on Bidirectional Transformations

Zhenjiang Hu
University of Tokyo
7-3-1 Hongo, Bunkyo
Tokyo 113-8656, Japan
hu@mist.i.u-tokyo.ac.jp

Shin-Cheng Mu
University of Tokyo
7-3-1 Hongo, Bunkyo
Tokyo 113-8656, Japan
scm@ipl.t.u-tokyo.ac.jp

Masato Takeichi
University of Tokyo
7-3-1 Hongo, Bunkyo
Tokyo 113-8656, Japan
takeichi@mist.i.u-tokyo.ac.jp

ABSTRACT

This paper presents 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 the 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 cannot only describe the relationship between the document source and its view, but also data dependency in the view.

Categories and Subject Descriptors

D.1.1 [**Programming Techniques**]: Applicative (Functional) Programming; D.1.2 [**Programming Techniques**]: Automatic Programming—*program transformation, program synthesis*; H.4.1 [**Information Systems Applications**]: Office Automation—*spreadsheets, word processing*

General Terms

Documentation, Languages

Keywords

View updating, Bidirectional transformation, Functional programming, Document Engineering, Editor

1. INTRODUCTION

XML [5] 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

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

Figure 1: A DTD for the Address Book

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

Figure 2: An XML Document of the Address Book

[23] 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 with an example of construction of an address book. 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. We may start by defining an address book type (Figure 1), which allows an arbitrary number of people's addresses including a name, some email addresses if there are, and a telephone number.

```
<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>
```

Figure 3: A Transformation in XSLT

Then, we construct an XML document (Figure 2) of this type to store address information. And finally, we define a transformation (Figure 3) to display[1] the address book in a friendly way (Figure 4), say by sorting persons according to the last names and adding an index of names. Notice the difference between the two XML documents, the original XML document in Figure 2 and the view in Figure 4. Besides difference in their structures, the former has no redundant information, while the later does; e.g., the same names appear twice in the view. 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.

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:

- First, they treat the three components of structured documents 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 that in Figure 5. This refinement requires corresponding changes on the XML document and the transformation, which is difficult.

- Second, they expect the users to be XML experts 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.

[1]To simplify our presentation, we consider the view as another XML data. It should be very straightforward to present this XML data in another format with a suitable style-sheet description.

```
<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 4: A View of the Address Book in XML

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

Figure 5: Another DTD for the Address Book

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 obtained automatically.

One challenge in design and implementation of such editing system 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 4, we would wish that when the user, for example, adds or deletes a person, the original document in Figure 2 be updated correspondingly. Further more, the changes should also trigger an update of the index of names in Figure 4. 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 describing the relationship among the source data, the view, and the transformation 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, 7, 11, 19, 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 [16, 12], 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 an unified, presentation-oriented interface for developing the three components through editing operations on the view.
  - *Presentation-oriented*: we provide 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

In this section, we introduce the notations in which we will describe structured documents in this paper. As in the introduction, we may 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.

For conciseness, we choose a Haskell-like[4] notation to represent structured documents. Other alternatives include HaXML [26], XDuce [13] or CDuce [3], languages or libraries designed for specification of XML documents in a functional style.

For the sake of conciseness and simplicity, we talk about only a subset of XML. We omitted attributes, which should not be too difficult to cope with by some simple extension. We do not have node sharing, not allowing, for example, IDRef in XML for referring to other nodes through a unique identifier. These will be among our future work to do.

2.1 Document Types

We may use Haskell types to represent the types of documents. For instance, we may define the type of the address book in Figure 2 as follows.

```
data Addrbook = Addrbook [Person]
data Person = Person (Name, [Email], Tel)
data Name = Name String
data Email = Email String
data Tel = Tel String
```

Any tree so constructed is type-checked by the Haskell type system, which is a good thing for the final document. However, for the interactive refinement of the documents, we should allow inconsistency during document development. To this end, we make use of the following generic tree

```
data Tree = N String [Tree]
```

to represent contents of any XML document, independent of all DTDs.

2.2 Source Documents

The following gives an example of this representation of the the document source in Figure 2.

```
addrbook = 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 "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" []]]]
```

The generic representation does not distinguish tag names from texts, since both of them are represented by strings. As a matter of fact, we can think of labels attached to inner nodes as tag names, and labels to leaves as text.

2.3 Transformation and Views

We will use the language in Section 3 to specify transformations mapping source documents to views. It is worth noting again that views allow local data dependency while source documents do not allow.

To give the whole expression of our formulation, we show an example of the transformation description below, which implement the same transformation in Figure 3.

```
sortX ;
applyX [] Dup ;
applyX [1] (modifyRootX "Index" ; Map keepX) ;
copyX [1] [2, 1] ;
deleteX ;
hoistX "Dup"
```

We will return to explain this transformation after explaining our transformation language.

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. We borrow the view-updating technique [2, 7, 11, 19, 1], which has been intensively studied in the database community. Given a database and a query which produces a view from the database, the view-updating technique is to reflect view modification upon the database. Though the idea is very similar, there are two major difficulties in using this technique in our view-oriented editor.

- Our view may contain local data dependency as seen in Figure 4 where the same name appears twice in the view. This requires synchronization both between the view and the source document and between data and its dependence inside the view.
- Our view modification should be reflected not only on the source document, but also on the transformation. In other words, the transformation (query) part, which is assumed to be fixed in the existing view-updating technique, should be modifiable in our framework.

In this section, we present a language in which the document designer specifies the relationship between the source data and the view. The language is an extension to similar languages in [16, 12]. It plays an important role in the design of our editor (see Section 4).

3.1 Bidirectionality

Before explaining our language, we clarify what we mean by being *bidirectional*. Following the convention in [12], we call the type of source documents C (concrete view) and that of target documents A (abstract view). They are both embedded in Tree but we nevertheless distinguish them for clarity. A transformation x defined in X is associated with two functions. The function $\phi_x :: C \to A$ maps the concrete view to an abstract view, which is displayed and edited by the user. The function $\lhd_x :: C \times A \to C$ takes the original concrete view and the edited abstract view, and returns an updated concrete view. In [12] they are called *get* and *put* respectively.

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

$$\begin{array}{lll} \text{GET-PUT-GET}: & \phi_x\ (c \lhd_x a) = a & \text{where } a = \phi_x\ c \\ \text{PUT-GET-PUT}: & c' \lhd_x (\phi_x\ c') = c' & \text{where } c' = c \lhd_x a \end{array}$$

The PUT-GET-PUT property says that if c' is a recently updated concrete view, mapping it to its abstract view and immediately performing the backward update does not change its value. Note that this property only needs to hold for those c' in the range of $\lhd_x$. For an arbitrary c we impose the GET-PUT-GET requirement instead. Let a be the abstract view of c. Updating c with a and taking the abstract view, we get a again.

In [16, 12], on the other hand, the GET-PUT and PUT-GET properties are required to hold for arbitrary a and c':

$$\begin{array}{ll} \phi_x\ (c \lhd_x a) = a & \text{for any abstract view } a \\ c' \lhd_x (\phi_x\ c') = c' & \text{for any concrete view } c' \end{array}$$

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

Figure 6: The Language X for Specifying Bidirectional Transformations

However, as shown in [12], this would imply that ϕ is injective, which in turn imply that we would have difficulty dealing with duplication. A reasonable definition of ϕ_x where x duplicates data would not be injective, since modifying either of the duplicated data should yield the same source.

Once we introduce duplication into our language, however, an editing action at one location of the abstract view may cause corresponding changes at other locations. Therefore we need an extra ϕ_x to perform the change in the abstract view. The two bidirectional properties above guarantees that no further updating is necessary.

3.2 The Language X

The syntax of the language X for specifying bidirectional transformation is given in Figure 6. Primitive transformations are denoted by non-terminal B. They can be composed to form more complicated transformations by one of the combinators defined in X. The language looks very similar to the bidirectional languages proposed in [16, 12]. The most important difference lies in the new language construct Dup, which enables description of data dependency inside the view.

In this section, we will focus on how to use the language to describe transformation of our interest. An important property of the language X is the following theorem, whose proof is omitted due to space limitation.

THEOREM 1 (BIDIRECTIONALITY OF X).
Any transformation described in X is bidirectional. □

We omit the proof of the theorem, but we can see its correctness from the explanation of the language below.

3.2.1 Primitive Transformations

Rather than giving a fixed set of primitive transformations as in [12], we adopt a general way to define two classes of primitive transformations — the *bidirectional* primitives (GFun) and the *unidirectional* ones (NFun). Together with the special primitive Dup, they are described below.

Duplication

In the forward direction, the function ϕ_{Dup} generates two copies of its input.

$$\phi_{\mathsf{Dup}}\ c = \mathtt{N}\ \text{“Dup”}\ [c, c]$$

In the backward direction, $\lhd_{\mathsf{Dup}}$ checks which of the two copies was touched by the user by comparing them with the original view c, and keeps only the changed one.

$$\begin{array}{lcll} c \lhd_{\mathsf{Dup}} (\texttt{N}\ \text{``}\texttt{Dup}\text{''}\ [a_1, a_2]) & = & a_2 & \text{if } a_1 = c \\ & = & a_1 & \text{if } a_2 = c \\ & = & a_1 & \text{otherwise} \end{array}$$

Here we assume that the user performs only one editing action before an updating event is triggered. Therefore, if none of a_1 and a_2 equals c, it must be the case that $a_1 = a_2$, because they result from the same editing action.

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. This is achieved by a backward update $\lhd_{\mathsf{Dup}}$ followed by a forward transform ϕ_{Dup}. The backward phase updates the touched value. The forward phase then overwrites the copies in the abstract view with new values.

The presence of Dup makes ϕ_x, where x uses Dup, a non-total and non-injective transformation. The function $\lhd_{\mathsf{Dup}}$ does not satisfy the PUT-GET rule for views not in the range of ϕ_{Dup}, and is therefore not a well-behaved transform in [12].

Bidirectional Primitive Transformations

A bidirectional primitive GFun (f, g) consists of two functions f and g satisfying:

$$\begin{array}{ll} \text{INV1}: & f \circ g \circ f = f \\ \text{INV2}: & g \circ f \circ g = g \end{array}$$

That is, g is the inverse of f *in the range of* f. The property is satisfied by all Galois-connected pairs of functions, thus the name GFun. The bidirectional semantics of GFun (f, g) is given by

$$\begin{array}{lcl} \phi_{\mathsf{GFun}\ (f,g)} c & = & f\ c \\ c \lhd_{\mathsf{GFun}\ (f,g)}\ a & = & g\ a \end{array}$$

In words, the abstract view is obtained by applying f to the concrete view, while the concrete view can be obtained by applying g to the abstract view, ignoring the original concrete view. That $\phi_{\mathsf{GFun}\ (f,g)}$ and $\lhd_{\mathsf{GFun}\ (f,g)}$ satisfy the bidirectional property is a direct consequence of INV1 and INV2.

Let us see some useful primitive transformations defined in this way. The simplest transformation is the identity transformation:

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

which relates two identical data, and is defined by a pair of two identity functions. In this example, the pair of functions are inverse of each other.

Another interesting transformation is defined by

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

which relates the concrete data with the abstract data such that the children of the root in the abstract view are sorted. The function *sortT* sorts the subtrees of the root, according to the first child value of each subtree. It is clear that *sortT* is not invertible, but *sortT* and *sortT* do satisfy the properties of INV1 and INV2.

Similarly, we may define other primitive transformations that are useful for manipulating tree locally.

- swapX $i\ j$ swaps the ith and jth subtrees of the root.

```
swapX $i\ j$ = GFun $(f, f)$
  where
    $f\ (\texttt{N}\ n\ ts) = \texttt{N}\ n\ (\mathsf{take}\ (i-1)\ ts \mathbin{+\!\!+} ts!!j$
                      $\mathbin{+\!\!+} \mathsf{take}\ (j-i-1)\ (\mathsf{drop}\ i\ ts)$
                      $\mathbin{+\!\!+} ts!!i \mathbin{+\!\!+} \mathsf{drop}\ j\ ts)$
```

- hoistX n: If the root has label n and a single child t, then the result is t.

```
hoistX $n$ = GFun $(f, g)$
  where
    $f\ (\texttt{N}\ m\ [t]) = t,$ if $m = n$
    $g\ t = \texttt{N}\ n\ [t]$
```

  Note that f is a partial function. If the input is not in its domain, an error message appears.

- newRoot n makes the current tree the single child of a new root with label n.

```
newRootX $n$ = GFun $(f, g)$
  where
    $f\ t = \texttt{N}\ n\ [t]$
    $g\ (\texttt{N}\ m\ [t]) = t,$ if $m = n$
```

- exchangeX exchanges the root with the node of the leftmost child tree that has no child.

```
exchangeX = GFun $(f, f)$
  where
    $f\ (\texttt{N}\ n\ (\texttt{N}\ m\ [\,] : ts)) = \texttt{N}\ m\ (\texttt{N}\ n\ [\,] : ts)$
```

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

```
insertHoleX = GFun $(f, g)$
  where
    $f\ (\texttt{N}\ n\ ts) = \texttt{N}\ n\ (\Omega : ts)$
    $g\ (\texttt{N}\ n\ (\Omega : ts)) = \texttt{N}\ n\ ts$
```

- deleteHoleX deletes the hole appearing as the leftmost child of the root.

```
deleteHoleX = GFun $(f, g)$
  where
    $f\ (\texttt{N}\ n\ (\Omega : ts)) = \texttt{N}\ n\ ts$
    $g\ (\texttt{N}\ n\ ts) = \texttt{N}\ n\ (\Omega : ts)$
```

- replaceHoleX t replaces the hole with tree t.

```
replaceHoleX $t$ = GFun $(f, g)$
  where
    $f\ \Omega = t$
    $g\ t' = \Omega,$ if $t = t'$
```

Restrictive Primitive Transformations

Not all primitive transformations we wish to have satisfy the properties INV1 and INV2. One example is the constX transformation that does not care about the concrete view but only requires the abstract view to be a constant tree. Another example is the numberX transformation that relates the concrete view with the abstract view such that the abstract view shows the number of the children of the root in the concrete view.

We specify these transformations using a single function

$$\mathsf{NFun}\ f$$

only showing how to map the concrete view to the abstract view. The bidirectional semantics of this kind of transformation can be defined as follows.

$$\begin{array}{lcl} \phi_{\mathsf{NFun}\ f} c & = & f\ c \\ c \lhd_{\mathsf{NFun}\ f}\ a & = & c \end{array}$$

Notice that $c \lhd_{\mathsf{NFun}\ f}$ always returns the original concrete view c, and ignores any change on the abstract view a. Notice also that $\mathsf{NFun}\ f$ does satisfy both the GET-PUT-GET and the PUT-GET-PUT properties, as seen in the following calculations.

$$\begin{array}{ll} & \phi_{\mathsf{NFun}\ f}\ (c \lhd_{\mathsf{NFun}\ f}\ \ a) \\ = & \quad \{ \text{ Def. of } \lhd_{\mathsf{NFun}\ f} \} \\ & \phi_{\mathsf{NFun}\ f}\ c \\ = & \quad \{ \text{ by the condition of } a \ \} \\ & a \end{array}$$

$$\begin{array}{ll} & c' \lhd_x (\phi_x\ c') \\ = & \quad \{ \text{ Def. of } \lhd_{\mathsf{NFun}\ f} \} \\ & c' \end{array}$$

Below are the definitions of the two transformations mentioned above.

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

One can turn any function f to be a transformation $\mathsf{NFun}\ f$, though its ability to update the source by editing the view will be hindered. In a sense, transformations defined by $\mathsf{NFun}\ f$ are not really "bidirectional", since all changes on the abstract view are simply ignored. However, it is still very helpful when used together with Dup, which we will see in Section 3.3.2.

3.3 Transformation Combinators

The set of transformation combinators is useful to construct bigger transformations. An informal explanation of these combinators is given in Figure 7. Most of the combinators are essentially the same as those in [12]. There are three new combinators, namely duplication, condition, and fold. The duplication combinator is to introduce data dependency inside a document. Different from the reference structure for sharing data, the duplication transformation treats duplicated data and the original in the same way. The condition combinator is used to apply different transformations according to the context or information of the local tree, and the fold combinator is useful for specifying interactive processing of documents.

3.3.1 Sequencing

Given two bidirectional transformations x_1 and x_2, the transformation $x_1; x_2$ informally means "do x_1, then do x_2". Its bidirectional semantics is given by

$$\begin{array}{lcl} \phi_{x_1;x_2} & = & \phi_{x_2} \circ \phi_{x_1} \\ c \lhd_{x_1;x_2} a & = & c \lhd_{x_1} ((\phi_{x_1}\ c) \lhd_{x_2} a) \end{array}$$

The forward transform $\phi_{x_1;x_2}$ is simply the sequential composition of ϕ_{x_1} and ϕ_{x_2}. To update the concrete view c with a modified abstract view a, we need to know what the intermediate concrete view was. It is computed by $\phi_{x_1}\ c$. The expression $(\phi_{x_1}\ c) \lhd_{x_2} a$ then computes an intermediate abstract view, which is used to update c with $\lhd_{x_1}$.

3.3.2 Product

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 forward transformation is defined by

$$\phi_{x_1 \otimes x_2}\ (\mathtt{N}\ c\ (c_1 : cs)) = \mathtt{N}\ a\ (a_1 : as)$$

where

$$\begin{array}{lcl} a_1 & = & \phi_{x_1}\ c_1 \\ \mathtt{N}\ a\ as & = & \phi_{x_2}\ (\mathtt{N}\ c\ cs). \end{array}$$

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. The backward updating is defined by updating the two slices separately.

$$(\mathtt{N}\ c\ (c_1 : cs)) \lhd_{x_1 \otimes x_2} (\mathtt{N}\ a\ (a_1 : as)) = \mathtt{N}\ c'\ (c'_1 : cs')$$

where

$$\begin{array}{lcl} c'_1 & = & c_1 \lhd_{x_1}\ a_1 \\ \mathtt{N}\ c'\ cs' & = & (\mathtt{N}\ c\ cs) \lhd_{x_2} (\mathtt{N}\ a\ as). \end{array}$$

As an example, consider the following transformation.

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

It maps a tree to another such that the new tree consists of not only the original tree but also the number of children of the root in the original tree. Notice the use of Dup together with the primitive transformation numberX. Although the number shown in the new tree is not editable because the transformation numberX is defined in terms of Nfun before, its value can be automatically changed if we remove or add a child to the root of the original tree.

3.3.3 Conditional Branches

In the forward direction, 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.

$$\begin{array}{lcll} \phi_{\mathsf{If}\ p\ x_1\ x_2}\ c & = & \phi_{x_1}\ c & \text{if } p\ c \\ & = & \phi_{x_2}\ c & \text{otherwise} \end{array}$$

In the backward direction, we check the root label to determine whether to apply $\lhd_{x_1}$ or $\lhd_{x_2}$ to the modified view.

$$\begin{array}{lcll} c \lhd_{\mathsf{If}\ p\ x_1\ x_2}\ a & = & c \lhd_{x_1} a & \text{if } p\ c \\ & = & c \lhd_{x_2} a & \text{otherwise} \end{array}$$

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 the all the node values is greater than 10, and keep it unchanged otherwise.

3.3.4 Map

We define two (higher order) transformation combinators, Map and Fold to recursively transform trees.

The well-known function *map* on lists is defined by

$$\begin{array}{lcl} \mathit{map}\ f\ [\,] & = & [\,] \\ \mathit{map}\ f\ (a : x) & = & f\ a : \mathit{map}\ f\ x \end{array}$$

The forward transform of $\mathsf{Map}\ x$ simply applies the transformation x to all subtrees of the given tree, leaving the root label unchanged.

$$\phi_{\mathsf{Map}\ x}\ (\mathtt{N}\ c\ cs) = \mathtt{N}\ c\ (\mathit{map}\ \phi_x\ cs)$$

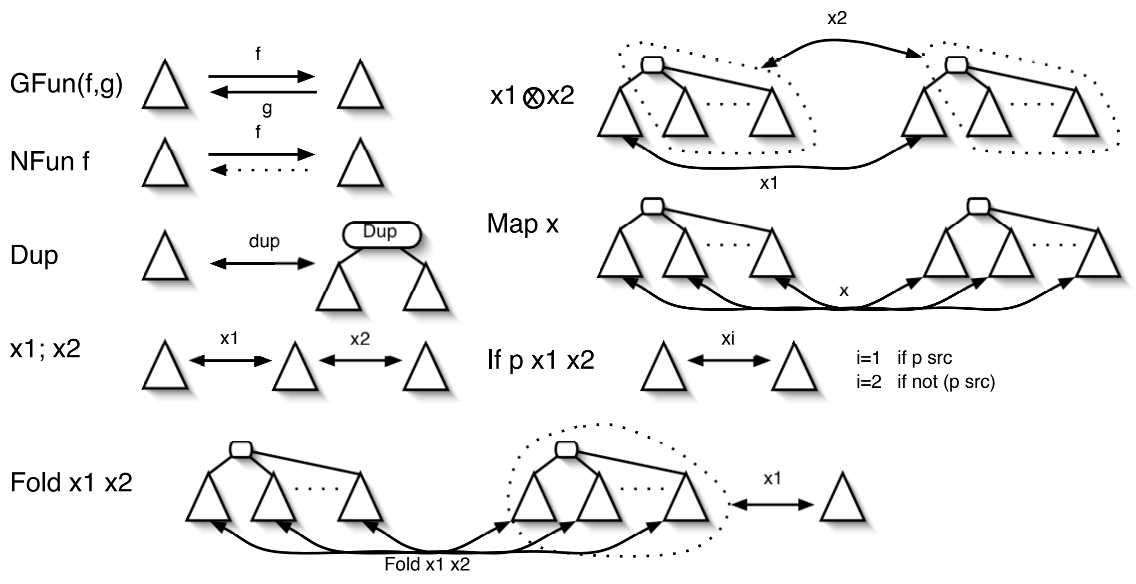

Figure 7: Intuitive Explanation of Transformation Combinators

The backward updating is defined by updating the subtrees separately,

$$(\mathtt{N}\ c\ cs) \lhd_{\mathsf{Map}\ x} (\mathtt{N}\ c\ as) \quad = \quad \mathtt{N}\ c\ (zip_{\lhd_x}\ cs\ as)$$

where the abstract and the concrete trees should have the same label, and function zip is defined as follows.

$$\begin{array}{lcl} zip_\oplus\ [\,]\ [\,] & = & [\,] \\ zip_\oplus\ (a:x)\ (b:y) & = & a \oplus b : zip_\oplus\ x\ y \end{array}$$

3.3.5 Fold

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 defined by

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

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.

In the backward direction, we use the cached copy of the concrete view to determine the depth of recursion to go into. Being able to reuse Map and sequencing significantly simplifies the definition.

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

If we expand the second clause of the definition, we get

$$(\mathtt{N}\ c\ cs) \lhd_{\mathsf{Fold}\ x_1\ x_2} a \quad = \quad \mathtt{N}\ c'\ cs'$$

where

$$\begin{array}{lcl} (\mathtt{N}\ c'\ as') & = & (\mathtt{N}\ c\ as) \lhd_{x_1} a \\ as & = & map\ \phi_{\mathsf{Fold}\ x_1\ x_2}\ cs \\ cs' & = & zip_{\lhd_{\mathsf{Fold}\ x_1\ x_2}}\ cs\ as'. \end{array}$$

Like in sequencing, we need an application of $map\ \phi_{\mathsf{Fold}\ x_1\ x_2}$ to create an intermediate value in order to perform $\lhd_{x_1}$. The subtrees are then updated using $zip_{\lhd_{\mathsf{Fold}\ x_1\ x_2}}$.

3.4 Programming in X

3.4.1 Editing as Bidirectional Transformation

With the language X, we are able to define the important editing operations as bidirectional transformations.

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

We may insert some document v as the leftmost child of the root using insertX v, or delete the leftmost child using deleteX, or modify the root node information with a new name n using modifyRootX n.

Other editing operations like moveX and copyX can be defined via a combination of insertX and deleteX.

3.4.2 Efficiency

Our formulation of the most important editing operations is very efficient. By efficiency we mean two things. First, we are able to define efficient editing operations. For example, another useful editing operation, keepX, which returns the leftmost subtree, could be naively realized by a sequence of deletion operations. Our language, however, allows the following more efficient definition:

$$\begin{array}{lcl} \mathsf{keepX} & = & \mathsf{idX} \otimes (\mathsf{constX}\ \Omega)\ ; \\ & & \mathsf{hoistX}\ (\mathsf{Root}_\Omega) \end{array}$$

where Root_Ω denotes the root node of the Ω tree. Note that the Ω tree is tree with just a single node acting as a hole to be filled.

Second, and more important in implementation of our presentation-oriented editors, we can produce as much editable data as possible in the view. There are many ways to define a source-to-view transformation; one may go extremely to define them just as basic transformations in terms of NFun with a non-invertible function. Consider the editing operation insertX defined in Section 3.4.1. One could have defined it as

$$\begin{array}{lcl} \mathsf{insertX'}\ v & = & \mathsf{NFun}\ f \\ & & \mathsf{where}\ f\ (\mathtt{N}\ n\ ts)\ =\ \mathtt{N}\ n\ (v : ts) \end{array}$$

which forbids any modification on the view. The definition of insertX in in Section 3.4.1, on the other hand, imposes no restriction at all on editing of the view.

3.4.3 An Example

We consider the specification of the transformation mapping the source document in Figure 2 to the view in Figure 4. The major difference between the view and the source document is that the entries in the view are sorted, and the view has an additional index of names. The transformation in XSLT has been given in Figure 3.

First, we consider specification of the transformation which applies transformation x to the subtree at path p but leave other parts of the tree unchanged. A path is a sequence of positive 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 $[1]$ denotes the first child of the root.

$$\begin{array}{lcl} \mathsf{applyX}\ [\,]\ x & = & x \\ \mathsf{applyX}\ (i : p)\ x & = & \mathsf{swapX}\ 1\ i\ ; \\ & & \mathsf{applyX}\ p\ x \otimes \mathsf{idX}\ ; \\ & & \mathsf{swapX}\ 1\ i \end{array}$$

Note that the applyX behaves as a higher order transformation; it accepts a transformation and returns a new transformation as the result.

Now we can code our transformation in X as follows.

$$\begin{array}{l} \mathsf{sortX}\ ; \\ \mathsf{applyX}\ [\,]\ \mathsf{Dup}\ ; \\ \mathsf{applyX}\ [1]\ (\mathsf{modifyRootX}\ \text{“}\mathtt{Index}\text{”}\ ;\ \mathsf{Map\ keepX})\ ; \\ \mathsf{copyX}\ [1]\ [2,1]\ ; \\ \mathsf{deleteX}\ ; \\ \mathsf{hoistX}\ \text{“}\mathtt{Dup}\text{”} \end{array}$$

We sort the address book according to person's names, duplicate the address book, keep only the name (first child) for each person in the duplicated address book and change the root name to be "Index", copy the list of names to the sorted address book and then delete it, and remove the label of "Dup" due to the duplication operation.

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.

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

They are standard except for the last two operators. For instance, InsertE $p\ v$ 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 internal data and the view respectively, and x denotes a bidirectional transformation. Each state $\mathcal{S} = (c, x, a)$ holds the following SYNC property.

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

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 the following functions for the above adjustments.

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

$\mathcal{A}_{cx}$ 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 8. 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 $p = p_1 +\!\!+ [a]$, and p' be a path expression in x satisfying $p' = p_1 +\!\!+ [b] +\!\!+ p_2$ and $b > a$, then p' will be changed to $p_1 +\!\!+ [b+1] +\!\!+ p_2$. 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

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

Figure 8: The Operational Semantics of the Editing Operations

to the SYNC property of the editor state, their semantics is very clear.

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:

$$
\begin{array}{l}
\mathsf{InsertE}\ p\ v \\
\mathsf{TransformE}\ p\ (\mathsf{insertX}\ v)
\end{array}
$$

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 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 a transformation accepting typed document sources. The difficult lies in finding the document type. One possible solution is to use the automatic extraction techniques [6, 9] to extract the document type information from c, however this approach is effective only when there is large amount of sample documents, which is not really suitable in our situation.

We adopt another approach. We ask the users to provide a type for the view (see our example in Section 5), and we infer types for the document source and the transformation. To do so, we borrow the idea from [20], where given a DTD for the XML source data and a query, an inference system derives a tight DTD for the view. Since our transformations are built up upon primitive transformations in terms of GFun (f, g) and NFun f, we can utilize the inference algorithms in [20], provided the types for functions used in the primitive transformations are given. We hope to design a language to define the functions used in primitive transformations and derive their types automatically in the future.

4.3 Infinite Undo

Another 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 operation to recover the state.

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

Here $s|p$ denotes the subtree in the view s at the path p, and 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 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: the system waits for the user to input an editing command, and updates the view upon accepting a command.

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 Figures 1, 2, and 3, such that the resulting view looks like that in Figure 4.

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.

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, we tell the system that the type of the view should be the following

```
data Addrbook = Addrbook (Index, [Person])
data Index = Index [Name]
data Person = Person (Name, [Email], Tel)
data Name = Name String
data Email = Email String
data Tel = Tel String
```

and our system automatically returns the triple (T, D, X) (written in our notation) similar to those in Figures 1, 2 and 3.

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 [23], which have been designed and implemented for supporting development of structured documents in XML. Most of them, such as XMLSpy [15], 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 [8, 25] 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 [22, 14], 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 [24], 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, 7, 11, 19, 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 [21] does not happen in our case.

During the design of the bidirectional transformation language X, much was learnt from the lenses combinators in [12], 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 [16], 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 [10], 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 [18, 17].

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.

This work is still in an early stage, and there is much work to do. Particularly, rather than designing a new bidirectional language, we are interested to look into the possibility of making the existing transformation languages like XSLT to be efficiently bidirectional.

8. ACKNOWLEDGMENTS

We wish to thank Atsushi Ohori for introducing us the work on the view updating technique, which actually motivated this work. We thank Tetsuo Yokoyama for his remarks on an earlier draft of this paper, to the PSD project members in University of Tokyo for stimulating discussions on the design and implementation of this editor, and to Kento Emoto, Kazutaka Matsuda, and Akimasa Morihata for helping us to implement the prototype system. We are grateful to the referees, who provided detailed and helpful comments.

This work was partially supported by *Comprehensive Development of e-Society Foundation Software* Program of the Ministry of Education, Culture, Sports, Science and Technology, Japan.

9. 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] V. Benzaken, G. Castagna, and A. Frisch. CDuce: An XML-centric general-purpose language. In *Proceedings of 2003 ACM SIGPLAN International Conference on Functional Programming*. ACM Press, 2003.

[4] R.S. Bird. *Introduction to Functional Programming using Haskell*. Prentice Hall, 1998.

[5] Tim Bray, Jean Paoli, and C.M. Sperberg-McQueen. Extensible markup language (XML) 1.0. 1998.

[6] 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.

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

[8] R. Furuta, V. Quint, and J. 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] G. Gottlob, P. Paolini, and R. 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. 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.

[13] Haruo Hosoya, Jerome Vouilon, and Benjamin Pierce. Regular expression types for XML. In *Proceedings of 2000 ACM SIGPLAN International Conference on Functional Programming*, pages 11–22. ACM Press, 2000.

[14] Johan Jeuring. Implementing a generic editor. In *2nd Workshop on Programmable Structured Documents*, February 2004.

[15] Larry Kim. *The Official XMLSPY Handbook*. John Wiley & Sons, 2002.

[16] Lambert Meertens. Designing constraint maintainers for user interaction. `http://www.cwi.nl/~lambert`, June 1998.

[17] S.C. Mu, Z. Hu, and M. Takeichi. An algebraic approach to bi-directional updating. submitted for publication, June 2004.

[18] 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)*, Stirling, Scotland, July 2004. Springer Verlag, LNCS.

[19] Atsushi Ohori and Keishi Tajima. A polymorphic calculus for views and object sharing. In *ACM PODS'94*, pages 255–266, 1994.

[20] Yannis Papakonstantinou and Victor Vianu. DTD inference for views of XML data. In *Proceedings of the nineteenth ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems*, pages 35–46. ACM Press, 2000.

[21] 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.

[22] Martijn M. Schrage and Johan Jeuring. Xprez: A declarative presentation language for XML. See `http://www.cs.uu.nl/research/projects/proxima/`, 2003.

[23] XML Software. A list of XML editors. See http://www.xmlsoftware.com/editors.html, 2004.

[24] 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.

[25] 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.

[26] Malcolm Wallace and Colin Runciman. Haskell and XML: Generic combinators or type-based translation? In *ACM SIGPLAN International Conference on Functional Programming*, pages 148–159, Paris, 1999. ACM Press.