Bidirectionalizing Tree Transformations

Zhenjiang Hu[1,2], Kento Emoto[1], Shin-Cheng Mu[1], and Masato Takeichi[1]

[1] Graduate School of Information Science and Technology,
The University of Tokyo
{hu,emoto,scm,takeichi}@mist.i.u-tokyo.ac.jp

[2] PRESTO 21, Japan Science and Technology Agency

Abstract. A transformation from the source data to a target view is said to be *bidirectional* if, when the target is altered, the transformation somehow induces a way to reflect the changes back to the source. Several domain-specific bidirectional transformation languages have been proposed. In this paper, we intend to show that most existing unidirectional tree transformations can be made bidirectional. Therefore it may be not really necessary to design new domain-specific languages for bidirectional transformation. As a case study, we consider the combinator library HaXML, which has been widely used in the Haskell community for generating, editing, and transforming XML documents. We show that any transformation in HaXML can be fully compiled into a bidirectional transformation.

1 Introduction

XML [1] has been attracting a tremendous surge of interest as a universal, queryable representation for structured documents, which has in part been stimulated by the growth of the Web and e-commerce. It has emerged as the *de facto* standard for representation of structured data and information interchange, and many organizations use XML as an interchange format for data produced by applications like graph-plotters, spreadsheets, and relational databases. An XML document is essentially a tree, which contains two basic types of content: tagged elements and plain text. A tagged element is written as a start tag and an end tag, possibly enclosing a sequence of contents (elements or text fragments). Figure 1 shows an XML document representing an address book, where each entry contains a name, an email address, and a telephone number.

Transformation of XML documents from one format (structure) to another plays a significant role in data interchange. They are often described using a transformation language like XSLT. Figure 2 depicts an example of the tree transformation mapping the XML document in Figure 1 to a HTML document (Figure 3) which has an index of names and displays the contact details in a table. This transformation is *unidirectional*, in the sense that it describes the transformation from the XML document to the HTML view, but not the other

```
<addrbook>
  <person>
    <name> Zhenjiang Hu </name>
    <email> hu@mist.i.u-tokyo.ac.jp </email>
    <tel> +81-3-5841-7411 </tel>
  </person>
  <person>
    <name> Kento Emoto </name>
    <email> emoto@ipl.t.u-tokyo.ac.jp </email>
    <tel> +81-3-5841-7412 </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>
```

Fig. 1. An XML Document of the Address Book

way round. The transformation itself does not give much clue how the XML document shall be updated if the HTML view is altered.

There are many situations where one wants to transform some data structure into a different form and wish that changes made to the new form be reflected back to the source data. One may want modification on the view to be reflected back to the original database, which is known as *view updating* in the database commuinity [2–6]. One may want to synchronize the bookmark files of several different web browsers (on different machines) [7], allowing bookmarks and bookmark folders to be added, deleted, edited, and reorganized in any browser and later combining the changes performed in different browsers. One may want to have a programmable editor [8] supporting interactive refinement in the development of structured documents, where one performs a sequence of editing operations on the document view, and the editor automatically derives an efficient and reliable source document and a transformation that produces the document view.

These situations call for *bidirectional transformations* on tree-structured data. In one direction, these transformations map a *concrete* tree into an *abstract* one; in the other, they map a modified abstract tree, together with the original concrete tree, to a correspondingly modified concrete tree. Several domain-specific languages [7, 9, 10, 8] have been proposed to define bidirectional transformations. One would like to know, however, whether an existing transform written in tree-

```
<xsl:template match="/">
  <html>
  <body>
    <h1>IPL Address Book</h1>
    <ul>
      <xsl:for-each select="addrbook/person">
      <li><xsl:value-of select="name"/></li>
      </xsl:for-each>
    </ul>
    <table>
    <tr>
      <th>Name</th>
      <th>Email</th>
      <th>Tel</th>
    </tr>
    <xsl:for-each select="addrbook/person">
    <tr>
      <td><xsl:value-of select="name"/></td>
      <td><xsl:value-of select="email"/></td>
      <td><xsl:value-of select="tel"/></td>
    </tr>
    </xsl:for-each>
    </table>
  </body>
  </html>
</xsl:template>
```

Fig. 2. A Transformation in XSLT

```
<html>
<body>
  <h1>IPL Address Book</h1>
  <ul>
    <li> Zhenjiang Hu </li>
    <li> Kento Emoto </li>
    <li> Shin-Cheng Mu </li>
    <li> Masato Takeichi </li>
  </ul>
  <table>
    <tr>
      <th>Name</th>
      <th>Email</th>
      <th>Tel</th>
    </tr>
  </index>
  <tr>
    <td> Zhenjiang Hu </td>
    <td> hu@mist.i.u-tokyo.ac.jp </td>
    <td> +81-3-5841-7411 </td>
  </tr>
  <tr>
    <td> Kento Emoto </td>
    <td> emoto@ipl.t.u-tokyo.ac.jp </td>
    <td> +81-3-5841-7412 </td>
  </tr>
  <tr>
    <td> Shin-Cheng Mu </td>
    <td> scm@mist.i.u-tokyo.ac.jp </td>
    <td> +81-3-5841-7411 </td>
  </tr>
  <tr>
    <td> Masato Takeichi </td>
    <td> takeichi@acm.org </td>
    <td> +81-3-5841-7430 </td>
  </tr>
  </table>
<body>
<html>
```

Fig. 3. A View of the Address Book in HTML

transformation languages, like XSLT, can be made bidirectional. As far as we are aware, there is little work on this.

In this paper, we argue that most tree transformations can be made bidirectional. As a case study, we consider the combinator library HaXML [11], which has been widely used in the Haskell community for generating, editing, and transforming XML documents. The library HaXML can be seen as a domain-specific language embedded in the general-purpose functional language Haskell. XML documents are represented using native Haskell data type, and HaXML provids a set of powerful higher order functions to process them. A transform coded in HaXML is usually more compact than its equivalent in DOM, SAX, or XSLT.

In this paper we will present a technique we call *bidirectionalization*, which compiles any HaXML transformation into a bidirectional language X [8], thereby making it bidirectional. Our main result can be summarized as follows:

> *Any tree transformation in HaXML can be made bidirectional, according to which any part of the data before transformation can be updated via editing operations on the transformed data if the transformation is specified in HaXML.*

The result is surprising because HaXML is a unidirectional tree transformation language designed without bidirectionality in mind. With bidirectionalization, we obtain bidirectional transformation for free.

The rest of the paper is organized as follows. We start by briefly reviewing the core of HaXML [11], a general-purpose unidirectional transformation language, in Section 2. After explaining the basic concepts of bidirectionality and a bidirectional transformation language X [8] in Section 3, we show that any transformation specified by HaXML can be translated to a bidirectional transformation in X in Section 4. Conclusions are made in Section 5.

2 Tree Documents and Tree Transformations

Before explaining how to bidirectionize tree transformations in Section 4, we begin with a brief overview of the core concept of HaXML [11], a combinator library in Haskell [12], showing how tree documents are generated, edited, and transformed with HaXML. We will use a Haskell-like notation for the rest of this paper.

2.1 Documents as Trees

There are two basic types of XML contents: elements and text fragments. An element consists of a tag and a sequence of contents. In HaXML, XML documents are represented by native Haskell data structure:

```
data Content = CElem Element
```

```
addrbook = N "Addrbook"
  [ N "Person"
      [ N "Name" [N "Zhenjiang Hu" []],
        N "Email" [N "hu@mist.i.u-tokyo.ac.jp" []]
        N "Tel" [N "+81-3-5841-7411" []]],
    N "Person"
      [ N "Name" [N "Kento Emoto" []],
        N "Email" [N "emoto@ipl.t.u-tokyo.ac.jp" []],
        N "Tel" [N "+81-3-5841-7412" []]],
    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" []]],
    N "Person"
      [ N "Name" [N "Masato Takeichi" []],
        N "Email" [N "takeichi@acm.org" []],
        N "Tel" [N "+81-3-5841-7430" []]]
  ]
```

Fig. 4. An Example of Simplified Representation of Tree Documents

```
              | CText String
data Element = Elem Name [Attribute] [Content]
```

For the sake of conciseness and simplicity, we shall omit attributes in documents and represent tag labels by strings. Therefore, an XML document is essentially an internally labelled rose tree with text fragments as its leaves. To further simplify the matter, we will represent documents by the following tree:

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

This representation does not distinguish labels from texts, since both of them are represented by strings. However, we may think of labels attached to inner nodes as tag names, and labels to leaves as text. For an element with no content such as `<br/>`, we may represent it by `N "br" [N "" []]`.

Figure 4 gives an example of this representation of the the document source in Figure 1.

2.2 Tree Transformations

Combinators in HaXML are also called *filters*, with the type:

```
type Filter = Tree -> [Tree]
```

A filter takes a tree and returns a sequence of tree. The result might be empty, a singleton list, or a collection of trees.

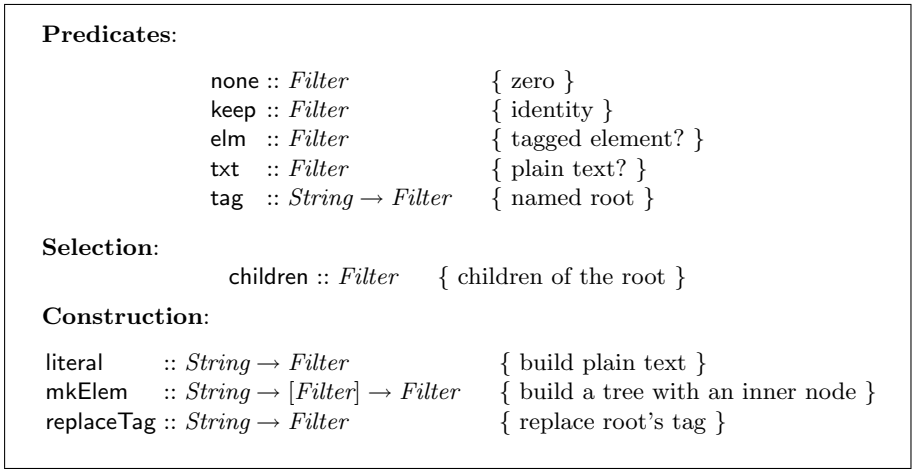

Fig. 5. Basic Filters

Basic Filters

A set of basic filters in HaXML is given in Figure 5. The simplest possible filter, none, fails on any input (returning an empty list); keep takes any tree and returns just that tree. The filter elm is a predicate, returning just this item if it is not a leaf, otherwise it fails. Conversely, txt returns this item only if the item is a leaf. The filter children returns the immediate children of the tree, if any. The filter tag t returns the input only if it is a tree whose root has the tag name t. The function literal s discards the input and returns a leaf labelled s. The function mkElem t fs builds a tree with the root label t; the argument fs is a list of filters, each of which is applied to the current item. The results are concatenated and become the children of the created element.

Filter Combinators

Figure 6 lists all combinators one can use to compose filters. The most important and useful among them is o, which plugs two filters together: the left filter is applied to the results of the right filter. So, for instance, the expression

txt `o` children `o` tag "`title`"

returns all the plain-text children immediately enclosed by the input, provided that the input is labelled `title`.

The combinator f ||| g concatenates the results of filters f and g, while cat fs is a generalisation of ||| to arbitrary numbers of filters. The combinator f `with` g acts as a guard on the results of f, keeping only those which are productive (yielding non-empty results) under g. Its dual, f `without` g, excludes those results of f that are productive under g. The filter f `et` g applies f to

o	:: *Filter* → *Filter* → *Filter*	{ Irish composition }
(|||)	:: *Filter* → *Filter* → *Filter*	{ append results }
cat	:: [*Filter*] → *Filter*	{ concatenate ressults }
with	:: *Filter* → *Filter* → *Filter*	{ guard }
without	:: *Filter* → *Filter* → *Filter*	{ negative guard }
et	:: (*String* → *Filter*) → *Filter* → *Filter*	{ disjoint union }
_ ?> _ :> _	:: *Filter* → *Filter* → *Filter* → *Filter*	{ condition }
chip	:: *Filter* → *Filter*	{ in-place children application }

Fig. 6. Basic Filter Combinators

the input if it the input is a leaf tree, and applies g to the input otherwise. The expression p ?> f :> g represents conditional branches; if the (predicate) filter p is productive given the input, the filter f is applied to the input, otherwise g is applied. The filter chip f applies f to the immediate children of the input. The results are concatenated as new children of the root.

Examples

A number of useful tree transformations can be defined as HaXML filters. For instance, we may define the following two path selection combinators /> and </.

$$f \mathbin{/>} g = g \text{ `o` } \mathsf{children} \text{ `o` } f$$
$$f \mathbin{</} g = f \text{ `with` } (g \text{ `o` } \mathsf{children})$$

Both of them apply f to the input and prune away those subtrees of the result that does not make g productive (i.e., g does not fail): /> is an 'interior' selector, returning the inner structure; </ is an 'exterior' selector, returning the outer structure.

Another class of useful filter combinators allows one to process trees recurively. The combinator deep f

$$\mathsf{deep}\ f = f \text{ ?> } f \text{ :> } (\mathsf{deep}\ f \text{ `o` } \mathsf{children})$$

potentially pushes the action of filter f deep inside the document sub-tree. It first tries the given filter on the current node: if the filter is productive then it stops, otherwise it moves to the children recursively. Another powerful recursion combinator is foldXml: the expression foldXml f applies the filter f to every level of the tree, from the leaves upwards to the root.

$$\mathsf{foldXml}\ f = f \text{ `o` } (\mathsf{chip}\ (\mathsf{foldXml}\ f))$$

```
html
  [ body
    [ h1 [ literal "IPLAddressBook" ],
      ul [replaceTag "li" `o` (keep /> tag "person" /> tag "name") ]
      table
        [ tr [ th [literal "Name", th [literal "Email"], th [literal "Tel"]],
          foldXML tabling `o` (keep /> tag "person") ] ] ]
  where
    tabling = tag "person" ?> replaceTag "tr" :>
              (tag "name" ?> repalceTag "td" :>
              (tag "email" ?> repalceTag "td" :>
              (tag "tel" ?> repalceTag "td" :>
               keep)))
```

Fig. 7. A Transformation in HaXML

Recall the transformation described by XSLT in Figure 2. By defining HTML constructors by

```
html  = mkElem "html"
body  = mkElem "body"
h1    = mkElem "h1"
ul    = mkElem "ul"
li    = mkElem "li"
table = mkElem "table"
tr    = mkElem "tr"
th    = mkElem "th"
td    = mkElem "td"
```

we can define it in HaXML as in Figure 7.

3 A Bidirectional Transformation Language

In this section, we present the bidirectional transformation language X, which is essentially the same as that in [8]. It will serve as the basis of our bidirectionalization transformation.

3.1 Bidirectionality

Before explaining our language, we clarify what we mean by being *bidirectional*. Following the convention in [7], 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

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

Fig. 8. The Language X for Specifying Bidirectional Transformations

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 [7] they are called *get* and *put* respectively.

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

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

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. Here, by "updating", we mean one of the following four editing operations: insertion of a hole to the view, deletion of a hole from the view, replacement of a subtree by a hole, and change of a node name.

3.2 The Language *X*

The syntax of the language X for specifying bidirectional transformation is given in Figure 8. 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 [9, 7]. The most important difference lies in the new language construct Dup, which enables description of data dependency inside the view. An important property of the language X is the following theorem.

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

Primitive Bidirectional Transformations

Bidirectional Primitive Transformations A bidirectional primitive $\mathsf{BFun}\ (f, g)$ consists of two functions f and g satisfying the GET-PUT-GET and PUT-GET-PUT properties. The bidirectional semantics of $\mathsf{BFun}\ (f, g)$ is given by

$$\begin{aligned} \phi_{\mathsf{BFun}\ (f,g)}\, c &= f\ c \\ c \lhd_{\mathsf{BFun}\ (f,g)}\, a &= g\ c\ a \end{aligned}$$

For a special case where g does not use its second argument and is the inverse of f, we write it as $\mathsf{GFun}\ (f, g)$:

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

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.

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

- $\mathsf{constX}\ t$ ignores the concrete view and gives a constant abstract view t.

$$\mathsf{constX}\ t = \mathsf{BFun}\ (\lambda x.\, t, \lambda c\, a.\, c)$$

- $\mathsf{hoistX}\ n$: If the root has label n and a single child t, then the result is t.

$$\begin{aligned} &\mathsf{hoistX}\ n = \mathsf{GFun}\ (f, g) \\ &\quad \mathsf{where} \\ &\qquad f\ (\mathtt{N}\ m\ [t]) = t, \text{ if } m = n \\ &\qquad g\ t = \mathtt{N}\ n\ [t] \end{aligned}$$

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

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

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

$$\begin{aligned} &\mathsf{insertHoleX} = \mathsf{GFun}\ (f, g) \\ &\quad \mathsf{where} \\ &\qquad f\ (\mathtt{N}\ n\ ts) = \mathtt{N}\ n\ (\Omega : ts) \\ &\qquad g\ (\mathtt{N}\ n\ (\Omega : ts)) = \mathtt{N}\ n\ ts \end{aligned}$$

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

$$\begin{array}{l}\mathsf{deleteHoleX} = \mathsf{GFun}\ (f, g)\\ \quad \mathsf{where}\\ \qquad f\ (\mathtt{N}\ n\ (\Omega : ts)) = \mathtt{N}\ n\ ts\\ \qquad g\ (\mathtt{N}\ n\ ts) = \mathtt{N}\ n\ (\Omega : ts)\end{array}$$

– replaceHoleX t replaces the hole with tree t.

$$\begin{array}{l}\mathsf{replaceHoleX}\ t = \mathsf{GFun}\ (f, g)\\ \quad \mathsf{where}\\ \qquad f\ \Omega = t\\ \qquad g\ t' = \Omega,\ \text{if } t = t'\end{array}$$

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

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

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.

$$\begin{array}{rl}c \lhd_{\mathsf{Dup}} (\mathtt{N}\ \text{“}\mathtt{Dup}\text{”}\ [a_1, a_2]) &= a_2 \text{ if } a_1 = c\\ &= a_1 \text{ if } a_2 = c\\ &= a_1 \text{ otherwise}\end{array}$$

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.

Bidirectional Transformation Combinators

The set of transformation combinators is useful to construct bigger transformations.

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

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

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 $\lhd_{x_1}$.

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

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

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

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

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.

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

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.

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

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

$$\begin{aligned} map\ f\ [\,] \quad &= [\,] \\ map\ f\ (a : x) &= f\ a : map\ f\ x \end{aligned}$$

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

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

The backward updating is defined by updating the subtrees separately,

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

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

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

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

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

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.

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

Examples: Bidirectional Editing Operations

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

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

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

4 Bidirectionalization Translation

We are now ready to show how tree transformations in HaXML can be compiled into the bidirectional transformation language X.

4.1 Tree Represenation of Lists

Our idea is to translate each filter in HaXML to a bidirectional transformation in X. There is, however, a technical problem due to their type difference; a filter is a map from Tree to [Tree], whereas a transformation in X is a map from Tree to Tree. We solve this problem by coding lists as a special tree whose root has a special label "`List`". For example, a list of trees $[t_1, t_2, \ldots, t_n]$ is represented by `N` "`List`" $[t_1, t_2, \ldots, t_n]$.

Below are some functions on the tree representation of lists.

$$\begin{array}{ll}
\mathsf{emptyList} & = \texttt{N "List" } [] \\
\mathsf{nullList}\ (\texttt{N "List" } []) & = \mathsf{True} \\
\mathsf{nullList}\ (\texttt{N _ _}) & = \mathsf{False} \\
\mathsf{singletonList}\ (\texttt{N "List" } [x]) & = \mathsf{True} \\
\mathsf{singletonList}\ (\texttt{N _ _}) & = \mathsf{False}
\end{array}$$

We are defining a set of primitive bidirectional transformations for manipulating tree representation of lists.

Transformation listizeX wraps a tree as a list that contains the tree as its single element.

$$\begin{array}{l}
\mathsf{listizeX} = \mathsf{GFun}\ (f, g) \\
\quad \textbf{where} \\
\quad f\ t = \texttt{N "List" } [t] \\
\quad g\ (\texttt{N "List" } [t]) = t
\end{array}$$

Transformation appendX flattens a list that has two list elements to a new list whose elements are a concatenation of the elements of the two element lists.

$$\begin{array}{l}
\mathsf{appendX} = \mathsf{BFun}\ (f, g) \\
\quad \textbf{where} \\
\quad f\ (\texttt{N}\ c\ [\texttt{N}\ a\ ta, \texttt{N}\ b\ tb]) \ = \ \texttt{N}\ c\ [\texttt{N}\ b\ (ta +\!\!+ tb)] \\
\quad g\ (\texttt{N}\ c\ [\texttt{N}\ a\ ta, \texttt{N}\ b\ tb])\ (\texttt{N}\ c'\ [\texttt{N}\ b'\ ts]) \ = \\
\qquad\qquad \texttt{N}\ c'\ [\texttt{N}\ a\ (\mathsf{take}\ n\ ts), \texttt{N}\ b'\ (\mathsf{drop}\ n\ ts)] \\
\quad\ \ \textbf{where}\ n \ = \ \mathsf{length}\ ta
\end{array}$$

Transformation concatX is a generalization of appendX, applying to a list that contains arbitrary number of list elements.

$$\begin{array}{l}
\mathsf{concatX} = \mathsf{BFun}\ (f, g) \\
\quad \textbf{where} \\
\quad f\ (\texttt{N}\ c\ ts) \ = \ \texttt{N "List" } (\mathsf{foldr}\ (+\!\!+)\ []\ (\mathsf{map}\ \mathsf{getContents}\ ts)) \\
\quad g\ (\texttt{N}\ c\ ts)\ (\texttt{N "List" } ts') \ = \ \texttt{N}\ c\ (\mathsf{recons}\ ts\ ts') \\
\quad \mathsf{getContents}\ (\texttt{N _}\ cs) \ = \ cs \\
\quad \mathsf{recons}\ [\texttt{N}\ r\ cs]\ cs' \ = \ [\texttt{N}\ r\ cs'] \\
\quad \mathsf{recons}\ ((\texttt{N}\ r\ cs) : rs)\ cs' \ = \\
\qquad\qquad (\texttt{N}\ r\ (\mathsf{take}\ n\ cs')) : (\mathsf{recons}\ rs\ (\mathsf{drop}\ n\ cs')) \\
\quad\ \ \textbf{where}\ n \ = \ \mathsf{length}\ cs
\end{array}$$

Finally, transformation filterX keeps those list elements that satisfy condition p.

$$
\begin{array}{l}
\mathsf{filterX}\ p = \mathsf{BFun}\ (f, g) \\
\qquad \textbf{where} \\
\qquad f\ (\mathtt{N}\ \text{“}\mathtt{List}\text{”}\ ts)\ =\ \mathtt{N}\ \text{“}\mathtt{List}\text{”}\ (\mathsf{filter}\ p\ ts) \\
\qquad g\ (\mathtt{N}\ \text{“}\mathtt{List}\text{”}\ ts)\ (\mathtt{N}\ \text{“}\mathtt{List}\text{”}\ ts') = \mathtt{N}\ \text{“}\mathtt{List}\text{”}\ (\mathsf{recons}\ ts\ ts') \\
\qquad \mathsf{recons}\ cs\ []\ =\ cs \\
\qquad \mathsf{recons}\ (c : cs)\ (c' : cs')\ =\ \textbf{if}\ p\ c\ \textbf{then}\ c' : (\mathsf{recons}\ cs\ cs') \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \textbf{else}\ c : (\mathsf{recons}\ cs\ (c' : cs'))
\end{array}
$$

4.2 Translation of Basic Filters

Under the tree representation of lists, filters in HaXML have the same type as that of bidirectional transformations in X, which enables us to express filters directly in terms of bidirectional transformation functions and combinators in X to achieve bidirectionality.

The following is the definitions of the predicate filters in terms of X.

$$
\begin{array}{l}
\mathsf{none}\ = \mathsf{constX}\ \mathsf{emptyList} \\
\mathsf{keep}\ = \mathsf{listizeX} \\
\mathsf{elm}\ = \mathsf{If}\ (\mathsf{not} \circ \mathsf{leafNode})\ \mathsf{keep}\ \mathsf{none} \\
\mathsf{txt}\ = \mathsf{If}\ \mathsf{leafNode}\ \mathsf{keep}\ \mathsf{none} \\
\mathsf{tag}\ n = \mathsf{If}\ ((==n) \circ \mathsf{root})\ \mathsf{keep}\ \mathsf{none}
\end{array}
$$

Function leafNode returns True if the input is not a leaf tree, and False otherwise.

Similarly, we can redefine the filters for selection and for tree construction.

$$
\begin{array}{ll}
\mathsf{children} & = \mathsf{modifyRootX}\ \text{“}\mathtt{List}\text{”} \\
\mathsf{literal}\ n & = \mathsf{constX}\ (\mathtt{N}\ n\ []\ ;\ \mathsf{listizeX} \\
\mathsf{mkElem}\ n\ [x] & = x\ ;\ \mathsf{replaceTag}\ n \\
\mathsf{mkElem}\ n\ (x : xs) & = \mathsf{Dup}\ ;\ (x \otimes (\mathsf{hoistX}\ \text{“}\mathtt{Dup}\text{”}\ ;\ \mathsf{mkElem}\ n\ xs))\ ;\ \mathsf{appendX} \\
\mathsf{replaceTag}\ n & = \mathsf{modifyRootX}\ n\ ;\ \mathsf{listizeX}
\end{array}
$$

4.3 Translation of Filter Combinators

The bidirectionalization translation for the combinator filters is given below.

$$
\begin{array}{ll}
f\ \text{`o`}\ g & = g\ ;\ \mathsf{Map}\ f\ ;\ \mathsf{concatX} \\
f\ |||\ g & = \mathsf{Dup}\ ;\ (f \otimes \mathsf{Map}\ g)\ ;\ \mathsf{concatX} \\
\mathsf{cat}\ [f] & = f \\
\mathsf{cat}\ (f : fs) & = \mathsf{Dup}\ ;\ (f \otimes \mathsf{Map}\ (\mathsf{cat}\ fs))\ ;\ \mathsf{concatX} \\
f\ \text{`with`}\ g & = f\ ;\ \mathsf{filterX}\ (\mathsf{not} \circ \mathsf{nullList} \circ (\phi_g)) \\
f\ \text{`without`}\ g & = f\ ;\ \mathsf{filterX}\ (\mathsf{nullList} \circ (\phi_g)) \\
f\ \text{`et`}\ g & = (f\ \text{`oo`}\ \mathsf{tagged\ elm})\ |>|\ (g\ \text{`o`}\ \mathsf{txt}) \\
p\ ?>\ f\ :>\ g & = \mathsf{If}\ (\mathsf{not} \circ \mathsf{nullList} \circ (\phi_p))\ f\ g \\
\mathsf{chip}\ f & = \mathsf{Map}\ (f\ \ ;\ \ \mathsf{hoistX}\ \text{“}\mathtt{List}\text{”})\ \ ;\ \ \mathsf{listizeX}
\end{array}
$$

Two derived filter combinators |>| and oo are used, whose definitions are

$$f \mid\!>\!\mid g \;=\; f \;?\!>\; f \;:\!>\; g$$
$$f \;\text{`oo`}\; g = g \;;\; \mathsf{Map}\;(\mathsf{curryX}\;f)\;;\;\mathsf{concatX}$$

in which the bidirectional transformation curryX is defined as follows.

$$\begin{aligned}\mathsf{curryX}\; h = \mathsf{BFun}\;(f, g)\\ &\textbf{where}\\ &f\;(\mathtt{N}\;\text{“}\mathtt{Pair}\text{”}\;[a,x]) \;=\; \phi_{(h\,(\mathsf{root}\,a))}\;x\\ &g\;(\mathtt{N}\;\text{“}\mathtt{Pair}\text{”}\;[a,x])\;y \;=\; \mathtt{N}\;\text{“}\mathtt{Pair}\text{”}\;[a, x \lhd_{(h\,(\mathsf{root}\,a))}\;y]\end{aligned}$$

We omit detailed explanation of these definitions. It would not be difficult to understand them if comparing them with those in [11].

4.4 An Application: Bidirectional Editing

Recall the example in the introduction, where we transform the data in Figure 1 to that in Figure 3. This data transformation can be specified in HaXML, as seen in Section 2. According to the result in this section, we know that this transformation is actually bidirectional. Therefore, we cannot only edit the data in Figure 1 and ask a system to automatically update that in Figure 3, which is usual, but also edit the data in Figure 3 and ask a system to automatically update that in Figure 1, which benefits much from our bidirectionalization transformation.

We give three simple examples of bidirecetional editing below, and refer readers to the paper [8] for details about use of bidirectional transformations in construction of a programmable editor.

Example 1: If we *change* the first occurrence of `Zhenjiang Hu` to `ZHENJIANG HU` in Figure 3, our bidirectional transformation will reflect this change back to the data in Figure 1, by changing the name `Zhenjiang Hu` to `ZHENJIANG HU`. Furthermore, a forward transformation from the updated data in Figure 1 will change the second occurrence of `Zhenjiang Hu` to `ZHENJIANG HU` in Figure 3, keeping data consistency.

Example 2: If we *delete* the table entry

```
<tr><td>Zhenjiang Hu</td> ... </tr>
```

in Figure 3 (by first replacing this part by a special hole Ω^- and then deleting the hole), a backward transformation will delete the subtree

```
<person><name>Zhenjiang Hu</name> ... </person>
```

in Figure 1. Then a forward transformation will delete `<li> Zhenjiang Hu </li>`, keeping data consistency.

Example 3: If we *modify* the tag name `body` to `newbody`, an error message will be reported, because this requires to change the transformation rather than the data in Figure 1. Although not all data in Figure 3 are editable, any data in Figure 1 can be modified by editing some part of data in Figure 3.

5 Conclusions

In this paper, we propose a new general transformation, called *bidirectionalization*, for making tree transformations bidirectional. As far as we are aware, this is the first attempt. Our result is really encouraging: any tree transformation in HaXML is bidirectional. This make HaXML be a more powerful transformation language for data exchange than it was first designed. A rapid prototype system has been implemented. Recently, we realized that both X and HaXML could be redefined over a more fundamental but powerful injective language Inv [10, 13], This would enable us to bidirectionalize more complicated tree transformation, and we are now investigating how to compile XSLT into efficient bidirectional codes.

References

1. Bray, T., Paoli, J., Sperberg-McQueen, C.: Extensible markup language (xml) 1.0. (1998)
2. Bancilhon, F., Spyratos, N.: Updating semantics of relational views. ACM Transactions on Database Systems **6** (1981) 557–575
3. Dayal, U., Bernstein, P.A.: On the correct translation of update operations on relational views. ACM TODS **7** (1982) 381–416
4. Gottlob, G., Paolini, P., Zicari, R.: Properties and update semantics of consistent views. ACM Transactions on Database Systems **13** (1988) 486–524
5. Ohori, A., Tajima, K.: A polymorphic calculus for views and object sharing. In: ACM PODS'94. (1994) 255–266
6. Abiteboul, S.: On views and XML. In: Proceedings of the 18th ACM SIGPLAN-SIGACT symposium on Principles of Database Systems, ACM Press (1999) 1–9
7. Greenwald, M.B., Moore, J.T., Pierc, B.C., Schmitt, A.: A language for bidirectional tree transformations. Technical Report Technical Report MS-CIS-03-08, Department of Computer and Information Science University of Pennsylvania (2003)
8. Hu, Z., Mu, S.C., Takeichi, M.: A programmable editor for developing structured documents based on bidirectional transformations. In: Proceedings of ACM SIGPLAN 2004 Symposium on Partial Evaluation and Program Manipulation, ACM Press (2004) 178–189
9. Meertens, L.: Designing constraint maintainers for user interaction. `http://www.cwi.nl/~lambert` (1998)
10. Mu, S., Hu, Z., Takeichi, M.: An injective language for reversible computation. In: Seventh International Conference on Mathematics of Program Construction (MPC 2004), Stirling, Scotland, Springer Verlag, LNCS 3215 (2004) 289–313
11. Wallace, M., Runciman, C.: Haskell and XML: Generic combinators or type-based translation? In: ACM SIGPLAN International Conference on Functional Programming, Paris, ACM Press (1999) 148–159
12. Bird, R.: Introduction to Functional Programming using Haskell. Prentice Hall (1998)
13. Mu, S., Hu, Z., Takeichi, M.: An algebraic approach to bi-directional updating. In: Second ASIAN Symposium on Programming Languages and Systems(APLAS 2004), Taipei, Taiwan, Springer Verlag, LNCS 3302 (2004) 2–18