Bidirectionalizing Tree Transformation Languages: A Case Study

Shin-Cheng Mu, Zhenjiang Hu, Masato Takeichi

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, with the updated source satisfying certain healthiness conditions. Several bidirectional transformation languages have been proposed. In this paper, on the other hand, we aim at making existing transformations bidirectional. As a case study we chose the Haskell combinator library, HaXML, and embed it into Inv, a language the authors previously developed to deal with bidirectional updating. With the embedding, existing HaXML transformations gain bidirectionality.

1 Introduction

XML [6] has emerged as the *de facto* standard for representation of structured data and information interchange. Many organisations use XML as an interchange format for data produced by applications like graph-plotters, spreadsheets, and relational databases. Transformation of XML documents from one format to another plays a significant role in data interchange. The XML address book in Figure 1, where each entry contains a name, an email address, and a telephone number may be transformed to an HTML document in Figure 2, with an index of names and a table enlisting the contact details. The transformation may be written in a domain-specific language, such as XSLT. We may use this transformation in an XML editor where the source XML document is displayed to the user as HTML, or a homepage builder where a webpage is generated from an XML database.

However, it is not specified how the XML document shall be updated if the HTML view is altered. Yet this reverse transformation from the view to the source, although not yet well-studied, is also important [13]. In an XML editor or in a homepage builder, we may wish that when the user, for example, adds or deletes a person in the view in Figure 2, the original document in Figure 1 be updated correspondingly. Further more, the changes should also trigger an update of the index of names in Figure 2. We may even wish that when an additional name is added to the index, a fresh, empty entry be added to the bodies in both the source document and the view.

```
<addrbook>
  <person>
    <name>Shin-Cheng Mu</name>
    <email>scm@mist</email>
    <tel>+81-3-5841-7411</tel>
  </person>
  <person>
    <name>Zhenjiang Hu</name>
    <email>hu@mist</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 representing an address book.

木変換言語の双方向化に関する事例研究.

穆信成, 胡振江, 武市正人, 東京大学大学院情報理工学系研究科, Graduate School of Information Science and Technology, University of Tokyo.

コンピュータソフトウェア, Vol.23, No.2 (2006), pp.129–141.
[論文] 2005 年 8 月 22 日受付.

```
<html>
<body>
  <h1>IPL Address Book</h1>
  <ul><li>Shin-Cheng Mu</li>
      <li>Zhenjiang Hu</li>
      <li>Masato Takeichi</li>
  </ul>
  <table>
   <tr><th>Name</th>
       <th>Email</th>
       <th>Tel</th></tr>
   <tr><td>Shin-Cheng Mu</td>
       <td>scm@mist</td>
       <td>+81-3-5841-7411</td></tr>
   <tr><td>Zhenjiang Hu</td>
       <td>hu@mist</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. 2 A view of the address book in HTML.

This so-called *bidirectional updating* problem (coined by, to the best of the authors' knowledge, [10]) is attracting lots of interests recently, as people identified various situations where one wants to transform some data structure into a different form and wishes 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, known as *view updating* in the database community [3] [7] [9] [20] [1]. One may want to synchronise bookmarks of several different web browsers [10], allowing bookmarks to be edited and reorganized and later combine changes. One may want to have a programmable editor [13] 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.

Several domain-specific languages [10] [17] [19] [13] have been proposed to define bidirectional transformations. In the forward direction, these transformations map a *source* tree to a *view*; in the backward direction, they map a modified view, together with the original source, to a correspondingly modified source. One would like to know, however, whether legacy transforms written in existing languages, such as XSLT or CDuce [5], can be made bidirectional. As far as we are aware, there is little work on this.

As a case study, we show in this paper how to *bidirectionalise* an existing XML processing language, HaXML. HaXML [24] is a collection of utilities for parsing, filtering, transforming, and generating XML documents using Haskell. It provides, among other tools, a combinator library which can be seen as a domain-specific language embedded in the general-purpose functional language Haskell. For the rest of the paper when we talk about HaXML, we will be referring to its combinator library.

If we think of the forward transformation as a function from the source to the view, bidirectional updating, at the first glance, is the problem of looking for a suitable source among the inverse image of the given view. The situation is made a bit more difficult when the transformation involves duplication and structural constraints. If we delete a name in the index part in Figure 2, for example, the edited view is not in the range of the transform. Yet we still need to produce a reasonable updated source.

The language Inv was developed to study the bidirectional updating behaviour. In [18], Inv was designed to be a programming language defining injective functions only, due to the authors' belief that a study of bidirectional updating can be made more elegant if we first concentrate on injective functions. In [19], the language was given an extended semantics, where every Inv expression of type $A \to B$ induces a binary relation between A and B, mapping the edited values in A, which might not be in the range of the injective function, to a reasonable choice of source in B. The sometimes biased choice can be inferred by algebraic rules.

The main work of this paper is to add another layer by developing an embedding of HaXML to Inv, such that existing HaXML transformations gains bidirectionality. The embedding of each HaXML construct is based on its forward semantics. When we switch to the extended semantics, however, we get a default backward transformation.

For the rest of the paper, we briefly review the core of HaXML in Section 2. Then, we highlight the bidirectional updating problem in Section 3. After explaining the basic concepts of bidirectionality and the language Inv in Section 4, we show that any transformation specified by HaXML can be embedded into a bidirectional transformation in Inv in Section 5. Related works are discussed in Section 6, and conclusions are made in Section 7. A prototype implementation of the HaXML embedding is now available on the authors' homepage.

2 Tree Documents and Tree Transformations

In this section we will briefly review the core of the combinator library of HaXML. For presentation of this paper, we will use a representation of XML simpler than that used by HaXML. We define a range of values as below:

$$\begin{aligned} V &::= \mathit{String} \mid [V] \mid (V, V) \mid T \\ T &::= \langle \mathit{String} \rangle [T] \mid \mathsf{L}\ \mathit{String} \\ [a] &::= [\,] \mid a : [a] \end{aligned}$$

In this paper, the string is the only atomic type. We use typewriter font to denote a string literal. We can construct pairs (V, V), lists $[a]$ (where the variable a can be substituted for other types), and trees. A tree is either a leaf, or a node with a label and a list of subtrees†1. When it is clear from the context we omit the L constructor to save space. An XML tree is represented by T. This rather simplified view omits some features, such as attributes, that are trivial to add, and some features such as IDRefs, which will be our future work. The values will be extended in Section 4 to record user editing. Figure 3 rewrites Figure 1 in this representation.

```
addrbook = ⟨addrbook⟩
  [⟨person⟩[⟨Name⟩[Shin-Cheng Mu],
            ⟨email⟩[scm@mist],
            ⟨tel⟩[+81-3-5841-7411]],
   ⟨person⟩[⟨name⟩[Zhenjiang Hu],
            ⟨email⟩[hu@mist],
            ⟨tel⟩[+81-3-5841-7411]],
   ⟨person⟩[⟨name⟩[Masato Takeichi],
            ⟨email⟩[takeichi@acm.org],
            ⟨tel⟩[+81-3-5841-7411]]
  ]
```

Fig. 3 An example of simplified representation of tree documents.

Predicates:

none	:: *Filter*	{ zero }
keep	:: *Filter*	{ identity }
elm	:: *Filter*	{ tagged element? }
txt	:: *Filter*	{ plain text? }
tag	:: *String* → *Filter*	{ named root }

Selection:

children	:: *Filter*	{ children of the root }

Construction:

literal	:: *String* → *Filter*	{ build plain text }
mkElem	:: *String* → [*Filter*] → *Filter*	{ build a tree using filters }
replaceTag	:: *String* → *Filter*	{ replace root's tag }

Fig. 4 Basic filters.

2.1 Tree Transformations

Combinators in HaXML are called *filters*. They have type $T \to [T]$, taking a tree and returning a possibly empty sequence of tree.

Basic Filters

A set of basic filters in HaXML is given in Figure 4. The simplest filters are none and keep; none fails on any input (returning an empty list), and keep takes any tree and returns just that tree.

The filter elm returns 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 tag t returns the input only if it is a tree whose root has the tag name t. The filter literal s always returns a leaf labelled s, while replaceTag s changes the label s if the input is a node, and returns empty list otherwise. The filters so far return either a singleton list or an empty list. In this paper we will call such filters *singleton* filters. Other filters do not have constraints on the length of the output. The filter children returns the immediate children of the tree, if any.

Filter Combinators

Figure 5 lists all combinators to compose filters out of simpler ones. The sequential composition $f \,⨾\, g$ applies f to the input, before applying g to

†1 This notation for trees is borrowed from CDuce [5]. However, unlike CDuce, we keep commas as delimiters in lists.

```
(⨾)         :: Filter → Filter → Filter
(|||)       :: Filter → Filter → Filter
cat         :: [Filter] → Filter
with        :: Filter → Filter → Filter
without     :: Filter → Filter → Filter
et          :: Filter → Filter → Filter
_?⟩ _ :⟩_   :: Filter → Filter → Filter → Filter
chip        :: Filter → Filter
```

Fig. 5 Basic filter combinators.

each of the output and concatenating the results. For example, tag `title` ⨾ children ⨾ txt returns all the plain-text children immediately enclosed by the input, provided that the input is labelled `title`[†2].

The combinator $f \mathbin{|||} g$ concatenates the results of filters f and g, while cat fs is its generalisation to a list of filters. The combinator f with g acts as a guard on the results of f, keeping only those that 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 the input if it is a leaf tree, and applies g otherwise. The expression $p ?\rangle\, f \;:\rangle g$ 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.

The filter 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.

Derived Combinators

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

$$f \,/\rangle\, g \;=\; f \,⨾\, \mathsf{children} \,⨾\, g$$
$$f \,\langle/\, g \;=\; f \;\mathsf{with}\; (\mathsf{children} \,⨾\, g)$$

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); $/\rangle$ can be seen as selecting a subtree given a path.

†2 In [24], composition is actually written backwards, as in $g \circ f$. In this paper we use forward composition to be consistent with the syntactical choice we made in Inv.

```
html
  [ body
    [ h1 [ literal IPL Address Book ],
      ul [(keep /⟩ tag person /⟩ tag name) ⨾ replaceTag li],
      table
        [ tr [ thLit Name, thLit Email, thLit Tel],
          (keep /⟩ tag person) ⨾ mkRow ] ] ]
 where
  mkRow  =  tr [(tag person/⟩tag name) ⨾ replaceTag td,
                (tag person/⟩tag email) ⨾ replaceTag td,
                (tag person/⟩tag tel) ⨾ replaceTag td]
html     =  mkElem html
body     =  mkElem body
h1       =  mkElem h1
ul       =  mkElem ul
li       =  mkElem li
table    =  mkElem table
tr       =  mkElem tr
thLit s  =  mkElem th [literal s]
td       =  mkElem td
```

Fig. 6 A transformation in HaXML.

As our major example, the transformation in Figure 6 maps the XML address book in Figure 1 to the HTML document in Figure 2.

3 The Bidirectional Updating Problem

Consider the filter

$$f = \mathsf{mkElem}\;\mathtt{m}\;[\mathsf{children} \,⨾\, \mathsf{tag}\;\mathsf{a}, \mathsf{children}]$$

upon receiving a source document $\langle \mathtt{r}\rangle[\langle \mathsf{a}\rangle[\,], \langle \mathsf{b}\rangle[\,]]$, producing the view $\langle \mathtt{m}\rangle[\langle \mathsf{a}\rangle[\,], \langle \mathsf{a}\rangle[\,], \langle \mathsf{b}\rangle[\,]]$. The first child, $\langle \mathsf{a}\rangle[\,]$, results from children ⨾ tag a, while the rest result from children.

It is conventional to call the source-to-view transform Get, and the view-to-source transform Put. Now assume the view is changed to $\langle \mathtt{m}\rangle[\langle \mathsf{a}\rangle[\,], \langle \mathsf{a}\rangle[\mathsf{c}], \langle \mathsf{b}\rangle[\,]]$. The altered view is not in the range of the function defined by f anymore. However, the system shall somehow know that the $\langle \mathsf{a}\rangle[\,]$ and $\langle \mathsf{a}\rangle[\mathsf{c}]$ came from the same subtree in the source, and Put it to the updated source $\langle \mathtt{r}\rangle[\langle \mathsf{a}\rangle[\mathsf{c}], \langle \mathsf{b}\rangle[\,]]$. If we perform Get again, we get a new view $\langle \mathtt{m}\rangle[\langle \mathsf{a}\rangle[\mathsf{c}], \langle \mathsf{a}\rangle[\mathsf{c}], \langle \mathsf{b}\rangle[\,]]$, which is now in the range of f. The source and the view are thus synchronised.

In general, the edited view may not be in the range of the transform. We may want to, in reasonable cases, have it be Put to some source. It may be the case that the editing shall not be allowed and the edited view is not mapped to any source. Or there may be more than one possible source, and the system has to make a choice.

That raises the question: what is a legal source? A more ambitious formulation of bidirectional updating may, for example, attempt to choose a source based on some external criteria (for example the *minimal change* principle in [17]). At present, however, we enforce only a conservative constraint, one that making sure that we do not need repeated Get and Put. For every transformation x, we assume the existence of two functions: $get_x :: S \rightarrow V$ defines the transformation from the source S to the view V, while $put_x :: (S \times V) \rightarrow S$ takes the original source and an edited view, and returns an updated source.

Definition 1 (Bidirectionality) A pair of functions $get_x :: S \rightarrow V$ and $put_x :: (S \times V) \rightarrow S$ is called *bidirectional* if they satisfy the following two properties.

Get-Put-Get:

$$get_x\ (put_x\ s\ v) = v \qquad \text{for } v = get_x\ s$$

Put-Get-Put:

$$put_x\ s'\ (get_x\ s') = s' \text{ for } s' = put_x\ s\ v$$

The Get-Put-Get property says that updating s with v and taking its view, we get v again, provided that v was indeed resulted from s — for general v this property may not hold. The Put-Get-Put property says that if s' is a recently updated source, mapping it to its view and immediately performing the backward update does not change its value. This property only needs to hold for those s' in the range of put_x . The two properties together ensures that when the user alters the view, we need to perform only one *put* followed by one *get*. No further updating is necessary. In Section 5. 4 we will show that our HaXML embedding is indeed bidirectional.

Remark: The following Get-Put and Put-Get properties are required in [17] [10] to hold for arbitrary v and s':

Get-Put: $put_x\ s\ (get_x\ s) = s$ for any source s

Put-Get: $get_x\ (put_x\ s\ v) = v$ for any view v

For our application, the Put-Get property does not hold for general v, as seen in the example above. The Get-Put property (which actually holds for our embedding if we restrict s to *untagged* values) implies our Put-Get-Put property, but we specify only the weaker constraint in the definition of bidirectionality. (**End of remark**)

4 The Language Inv

In this section we give a brief introduction to Inv. The reader is referred to [19] for a more complete account.

4. 1 The Language Inv

Shown in Figure 7 is a subset of Inv we need for this article. By Inv_V we denote the union of Inv expressions with the set of variable names. We denote by $[\![\_]\!]$ the semantics function. The full semantics of Inv is discussed in [19]. For the purpose of this paper, it suffices to think of each construct as defining a relation which, when its domain and range are restricted to the types defined in Section 2, reduces to an injective partial function. When the input is a *tagged* value, to be defined in Section 4. 2, the relation maps the input to an updated result induced

$$\begin{aligned}
\mathsf{Inv} ::= {} & \mathsf{Inv}^{\smile} \mid nil \mid cons \mid node \mid isStr \mid neq \\
& \mid \delta \mid dupNil \mid dupStr\ String \\
& \mid \mathsf{Inv}; \mathsf{Inv} \mid id \mid \mathsf{Inv} \cup \mathsf{Inv} \\
& \mid \mathsf{Inv} \times \mathsf{Inv} \mid assocr \mid assocl \mid swap \\
& \mid \mu(V \colon \mathsf{Inv}_V)
\end{aligned}$$

$$\begin{aligned}
[\![nil]\!]\ () &= [\,] \\
[\![cons]\!]\ (a, x) &= a \colon x \\
[\![node]\!]\ (a, x) &= \langle a \rangle x \\
[\![id]\!]\ a &= a \\
[\![p?]\!]\ a &= a \text{ if } p\ a \\
[\![dupNil]\!]\ a &= (a, [\,]) \\
[\![dupStr]\!]\ s\ a &= (a, s) \\
[\![\delta]\!]\ a &= (a, a) \\
[\![f; g]\!]\ x &= [\![g]\!]\ ([\![f]\!]\ x) \\
[\![f \times g]\!]\ (a, b) &= ([\![f]\!]\ a, [\![g]\!]\ b) \\
[\![isStr]\!]\ s &= s \text{ if } s \text{ is a string} \\
[\![neq]\!]\ (a, b) &= (a, b) \text{ if } a \neq b \\
[\![f^{\smile}]\!] &= [\![f]\!]^{\circ} \\
[\![\mu F]\!] &= [\![F\ \mu F]\!] \\
[\![swap]\!]\ (a, b) &= (b, a) \\
[\![assocr]\!]\ ((a, b), c) &= (a, (b, c)) \\
[\![assocl]\!]\ (a, (b, c)) &= ((a, b), c) \\
[\![f \cup g]\!] &= [\![f]\!] \cup [\![g]\!],
\end{aligned}$$

$$\text{if } dom\, f \cap dom\, g = ran\, f \cap ran\, g = \emptyset$$

Fig. 7 The language Inv and its semantics when restricted to values without editing tags.

by the algebraic rules in [19].

The language Inv is a point-free language in the "squiggle" style, with constructors for datatypes (*nil* and *cons*, and *node*, where the domain of *nil* is restricted to unit type), identity function *id*, composition (;), product $(f \times g)$, and union $(f \cup g$, with f and g having disjoint domains and ranges). One can also produce an empty list or a string using *dupNil* or *dupStr*. Note that $(f; g \times h)$ should be bracketed as $((f; g) \times h)$. The functions *swap*, *assocl* and *assocr* distributes the components of the input pair. We also define:

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

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

The *converse* of a relation R is defined by

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

In the injective semantics, the *reverse* operator $(\_)^{\smile}$ corresponds to converses of relations. The reverse of *cons*, for example, decomposes a non-empty list into its head and tail. The reverse operator distributes into composite constructs by the following rules, all implied by the semantics definition $[\![f^{\smile}]\!] = [\![f]\!]^{\circ}$:

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

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

A number of list processing functions can be defined using the fixed-point operator, such as:

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

In Inv there is no higher-order functions. However, *foldr* and *map* can be seen as macros.

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

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

Note that to duplicate a list we shall duplicate each element and unzip the resulting list of pairs. In the discussion later we will omit the type subscript.

Concatenating two lists $x +\!\!+ y$ is not injective. Nor is the standard function $concat :: [[A]] \rightarrow [A]$ flattening a list of lists. However, the function $catx(x,y) = (x, x +\!\!+ y)$ is injective:

$$\begin{aligned} catx &= \mu(X : swap; dupNil^{\smile}; dupNil; swap \cup \\ &\qquad (cons^{\smile} \times id); assocr; \\ &\qquad (dup \times X); trans; (cons \times cons)) \end{aligned}$$

With *catx* we can define the following injective variant of *concat*:

$$\begin{aligned} concatx &= foldr\,cx\,(nil; dup) \\ \textbf{where}\;\; cx &= subr; (id \times catx); \\ &\qquad assocl; (swap; cons \times id) \end{aligned}$$

which is informally specified by $concatx\,[x,y,\ldots,z] = ([x,y,\ldots z], x +\!\!+ y +\!\!+ \ldots +\!\!+ z)$. This function turns out to be crucial in our HaXML embedding.

4.2 Duplication, Alignment, and Concatenation

One of the motivation behind the development of Inv was to study the handling of duplication and structural alignment. To do so we have to extend the domain of values we deal with:

$$\begin{aligned} V &::= A \mid V^{+} \mid V^{-} \mid [V] \mid (V,V) \mid T \\ T &::= \langle A \rangle [T] \mid \mathsf{L}\,A \\ [a] &::= [\,] \mid a : [a] \\ A &::= String \mid *String \mid \top \end{aligned}$$

The *editing tags* $(\_)^{+}$, $(\_)^{-}$, $*(\_)$, and $\top$ records the action performed by the user. When the user changes the value of a string the editor marks it with the $*(\_)$ tag. The $(\_)^{+}$ tag indicates that the tagged element is newly inserted by the user. When the user deletes an element it is wrapped by a $(\_)^{-}$, keeping note that it ought to be deleted but we temporary leave it there for further processing. The symbol $\top$ denotes an unconstrained value, to be further refined[†3]. Values containing any of the tags

†3 In the semantics in [19], there is no $\top$. Instead, a relation may non-deterministically map the input to many outputs, and refined when composed

are called *tagged*, otherwise *untagged*. The injective semantics of Inv deals with untagged values only. In this section, we will informally talk about how Inv programs behave, in the extended semantics, given tagged input. A more formal account is given in [19].

As described in the previous section, $\delta^{\smile}$ is a partial function performing equality test. In the extended semantics, we generalise δ such that it recognises the tag:

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

When the two values are not the same but one of them was edited by the user, the edited one gets precedence.

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

One of the main achievement of [19] is that, using the same definition of *unzip* above with a small amount of annotations, *zip* in the extended semantics knows how to zip together two lists when they contain inserted or deleted elements. For example, $unzip\,[(1, \mathtt{a}), (2, \mathtt{b}), (3, \mathtt{c})]$ yields $([1, 2, 3], [\mathtt{a}, \mathtt{b}, \mathtt{c}])$. If we label one element with a delete tag, $zip\,([1, 2^{-}, 3],\ [\mathtt{a}, \mathtt{b}, \mathtt{c}])$ yields $[(1, \mathtt{a}), (2, \mathtt{b})^{-}, (3, \mathtt{c})]$ — the corresponding element is marked for deletion as well. If we insert an element, say $([1, 2, 3], [\mathtt{a}, \mathtt{b}, \mathtt{d}^{+}, \mathtt{c}])$, zipping them together yields $[(1, \mathtt{a}), (2, \mathtt{b}), (\top, \mathtt{d})^{+}, (3, \mathtt{c})]$. An unconstrained value is invented and paired with the newly inserted $\mathtt{d}$, and might later be further constrained by δ or other structural constraints.

Recall that $dup_{[a]} = map\ dup_a; unzip$. The use of *unzip* synchronises the shape of the two lists in the backward updating. Similarly with dup_{Tree} where we use *unzip* to synchronise the list of subtrees.

The reverse of *concatx* maps $([x_1, x_2, \ldots, x_n], x)$ to $[x_1, x_2, \ldots, x_n]$ if $x_1 +\!\!+\ x_2 \ldots +\!\!+\ x_n = x$. Rather than simply returning the first component of the pair, every element in x is checked against elements in x_1, $x_2 \ldots x_n$. There are a number of ways to partition a list x into segments, but, in the injective semantics, only one of them is consistent with the original input. In the extended semantics, however, we have to make a biased choice when new items are inserted to the list. According to the semantics in [19], this particular definition of *concatx* tend to glue the new element at the edge of lists to the back. For example, $([[1, 2], [3, 4]], [1, 2, 5^{+}, 3, 4])$ is mapped to $[[1, 2], [5^{+}, 3, 4]]$.

with other relations. However, $\top$ was indeed used in the implementation.

$$\begin{array}{l}
concatw\ [\,] \qquad\quad\ = ([\,], [\,]) \\
concatw\ ([\,]{:}xs) \ \ = (([\,]{:}) \times id)\ (concatw\ xs) \\
concatw\ ([a]{:}xs) = (([a]{:}) \times (a{:}))\ (concatw\ xs) \\
concatw\ ((a{:}b{:}x){:}\ xs) = \\
\quad ((\lambda((b{:}x){:}xs) \rightarrow (a{:}b{:}x){:}xs) \times (a{:})) \\
\qquad (concatw\ ((b{:}x){:}xs)) \\
concatw = \mu(X : \\
\quad nil^{\smile}; nil; dup \cup \\
\quad cons^{\smile}; (\\
\qquad swap; dupNil^{\smile}; X; (dupNil; swap; cons \times id) \cup \\
\qquad (wrap^{\smile} \times id); (dup; (wrap \times id) \times X); \\
\qquad\quad trans; (cons \times cons) \cup \\
\qquad (cons^{\smile} \times id); assocr; \\
\qquad\quad (dup \times cons; X; (cons^{\smile} \times id)); \\
\qquad\quad trans; (assocl; swap \times id); assocr; \\
\qquad\quad (id \times (cons \times cons)); assocl; \\
\qquad\quad (swap; cons \times id))) \\
wrap = dupNil; cons
\end{array}$$

Fig. 8 *concatw* **and its** Inv **translation.**

In some occasions, however we need to make the new element a stand alone singleton list. The Haskell function *concatw* in Figure 8 deals with singleton list separately. Its Inv translation coincides with *concatx* in the injective semantics. In the extended semantics, $concatw^{\smile}$ breaks the list after newly inserted elements. For example, $([[1, 2], [3, 4]], [1, 2, 5^{+}, 3, 4])$ is mapped to $[[1, 2], [5]^{+}, [3, 4]]$. The behaviour of *concatx* and *concatw* are not arbitrary, but as results of algebraic rules in [19]. We need both functions in the embedding.

5 Bidirectionalisation Embedding

We are now ready to show how HaXML filters can be embedded into Inv. We call the source documents S and that of views V. The trick is that every HaXML construct is embedded as an Inv expression denoting, in the injective semantics, a function

$S \to (S \times [V])$ that takes a source and produces a pair consisting of a copy of the given source together with the view.

The function is apparently injective because the source is kept in the output. Its inverse, of type $(S \times [V]) \to S$, maps the original source and its corresponding views back to the source. In the extended semantics, when given the original source and an *edited* view, the Inv expression produces an updated source consistent with the transform.

5.1 Embedding Basic Filters

The embedding from HaXML constructs to Inv is denoted by $\lceil \_ \rceil$. The filter none always pair the input with an empty list. It is therefore simply embedded as *dupNil*. The filter keep, on the other hand, always produces a singleton list of the input:

$$\begin{aligned} \lceil \mathsf{none} \rceil &= \mathit{dupNil} \\ \lceil \mathsf{keep} \rceil &= \mathit{dup}; (\mathit{id} \times \mathit{wrap}) \end{aligned}$$

where $\mathit{wrap} = \mathit{dupNil}; \mathit{cons}$, wrapping an item into a singleton list. Other "singleton" filters are defined similarly:

$$\begin{aligned} \lceil \mathsf{elm} \rceil &= \mathit{isNode}; \mathit{dup}; (\mathit{id} \times \mathit{wrap}) \cup \\ &\quad\ \mathit{isStr}; \mathit{dupNil} \\ \lceil \mathsf{txt} \rceil &= \mathit{isNode}; \mathit{dupNil} \cup \\ &\quad\ \mathit{isStr}; \mathit{dup}; (\mathit{id} \times \mathit{wrap}) \end{aligned}$$

where $\mathit{isNode} = \mathit{node}^{\smile}; \mathit{node}$.

The filter tag is slightly more complicated because we need to check the value of the tag:

$$\begin{aligned} \lceil \mathsf{tag}\, s \rceil = \ & \mathit{node}^{\smile}; (\mathit{sEq}\, s \times \mathit{id}); \mathit{node}; \mathit{dup}; \\ & \quad (\mathit{id} \times \mathit{wrap}) \cup \\ & \mathit{node}^{\smile}; (\mathit{sNEq}\, s \times \mathit{id}); \mathit{node}; \mathit{dupNil} \cup \\ & \mathit{isStr}; \mathit{dupNil} \end{aligned}$$

where $\mathit{sEq}\, s$ and $\mathit{sNEq}\, s$ check whether the given string equals or not equals s. They are defined by:

$$\begin{aligned} \mathit{sEq}\, s &= \mathit{dupStr}\, s; \delta^{\smile} \\ \mathit{sNEq}\, s &= \mathit{dupStr}\, s; \mathit{neq}; (\mathit{dupStr}\, s)^{\smile} \end{aligned}$$

The filter children uses *dup* to copy the list of children, after decomposing the input using $\mathit{node}^{\smile}$. The input is reconstructed using *node* again.

$$\begin{aligned} \lceil \mathsf{children} \rceil = \ & \mathit{node}^{\smile}; (\mathit{id} \times \mathit{dup}); \mathit{assocl}; \\ & \quad (\mathit{node} \times \mathit{id}) \cup \\ & \mathit{isStr}; \mathit{dupNil} \end{aligned}$$

We will defer the discussion about another important filter mkElem to the next section, after we talk about sequential composition.

5.2 Embedding Sequential Composition

Assume that we have two embedded filters $\lceil f \rceil :: A \to (A \times [B])$ and $\lceil g \rceil :: B \to (B \times [C])$. How should we produce their embedded composition of type $A \to (A \times [C])$? The Inv expression $\lceil f \rceil; (\mathit{id} \times \mathit{map} \lceil g \rceil)$ applies $\lceil f \rceil$ to the input and $\lceil g \rceil$ to every result of $\lceil f \rceil$, resulting in $(A \times [(B \times [C])])$. We now need to get rid of the intermediate values of type B, and concatenate the nested Cs into a single list. However, there is no information-losing constructs in Inv.

Let us first try to concatenate all the Cs together. The function $\mathit{pull} :: [(B \times [C])] \to ([(B \times [C])] \times [C])$ below, using *concatx*, collects all the Cs a single list, while keeping the input $[(B \times [C])]$.

$$\mathit{pull} = \mathit{unzip}; (\mathit{id} \times \mathit{concatx}); \mathit{assocl}; (\mathit{zip} \times \mathit{id})$$

The next step is to notice that $\lceil g \rceil^{\smile}$ has type $(B \times [C]) \to B$. If we apply $\mathit{map}\, \lceil g \rceil^{\smile}$ to the list $[(B \times [C])]$, we get a list of Bs. Finally, we can eliminate the Bs using $\lceil f \rceil^{\smile} :: (A \times [B]) \to A$. Composition of filters is therefore defined by:

$$\begin{aligned} \lceil f \mathbin{\hat{;}} g \rceil &= \lceil f \rceil \triangleright \lceil g \rceil \\ f \triangleright g &= f; (\mathit{id} \times \mathit{map}\, g; \mathit{pull}; (\mathit{map}\, g^{\smile} \times \mathit{id})); \\ &\quad\ \mathit{assocl}; (f^{\smile} \times \mathit{id}) \end{aligned}$$

We isolate the definition of $\triangleright$ because we will use it again later. The seemingly inefficient applications of $\lceil f \rceil^{\smile}$ and $\lceil g \rceil^{\smile}$ is only in the specification. This is essential the same trick used by [4] to embed Turing machines into reversible Turing machines, where the embedded Turing machine is ran backwards to eliminate the intermediate result. The situation is merely made more complicated by the fact that filters return a list of results.

What made the effort worth, however, is that the same Inv expression also specifies how to perform the backward updating. Consider children $\hat{;}$ children, given the input $t = \langle \mathtt{a} \rangle [\langle \mathtt{b} \rangle [\mathtt{c}, \mathtt{d}], \langle \mathtt{e} \rangle [\mathtt{f}, \mathtt{g}]]$. In the forward run, the output is the original input paired with the list of grandchildren: $(t, [\mathtt{c}, \mathtt{d}, \mathtt{f}, \mathtt{g}])$. Assume the user inserts a new item $[\mathtt{c}, \mathtt{d}, \mathtt{h}^{+}, \mathtt{f}, \mathtt{g}]$. Now let us trace $\lceil \mathsf{children} \mathbin{\hat{;}} \mathsf{children} \rceil^{\smile}$, which expands to (abbreviating $\lceil \mathsf{children} \rceil$ to *ch*):

$$\begin{aligned} & (\mathit{ch} \times \mathit{id}); \mathit{assocr}; (\mathit{id} \times (\mathit{map}\, \mathit{ch} \times \mathit{id})); \\ & \quad (\mathit{id} \times \mathit{pull}^{\smile}; \mathit{map}\, \mathit{ch}^{\smile}); \mathit{ch}^{\smile} \end{aligned}$$

We first apply $(\mathit{ch} \times \mathit{id}); \mathit{assocr}; (\mathit{id} \times (\mathit{map}\, \mathit{ch} \times \mathit{id}))$ to $(t, [\mathtt{c}, \mathtt{d}, \mathtt{h}^{+}, \mathtt{f}, \mathtt{g}])$, yielding:

$$\begin{aligned} & (t, ([(\langle \mathtt{b} \rangle [\mathtt{c}, \mathtt{d}], [\mathtt{c}, \mathtt{d}]), (\langle \mathtt{e} \rangle [\mathtt{f}, \mathtt{g}], [\mathtt{f}, \mathtt{g}])], \\ & \quad [\mathtt{c}, \mathtt{d}, \mathtt{h}^{+}, \mathtt{f}, \mathtt{g}])) \end{aligned}$$

It simply reproduces the intermediate values that were generated in the forward run. Then we apply $pull^{\smile}$ to the right of the pair, which compares $[[\mathtt{c},\mathtt{d}],[\mathtt{f},\mathtt{g}]]$ against against $[\mathtt{c},\mathtt{d},\mathtt{h}^{+},\mathtt{f},\mathtt{g}]$ using $concatx^{\smile}$, resulting in $[[\mathtt{c},\mathtt{d}],[\mathtt{h}^{+},\mathtt{f},\mathtt{g}]]$. Now we have

$$(t,[(\langle\mathtt{b}\rangle[\mathtt{c},\mathtt{d}],[\mathtt{c},\mathtt{d}]),(\langle\mathtt{e}\rangle[\mathtt{f},\mathtt{g}],[\mathtt{h}^{+},\mathtt{f},\mathtt{g}])])$$

Next, we apply $(id \times map\ ch^{\smile})$, through which $ch^{\smile}$ is applied to $(\langle\mathtt{e}\rangle[\mathtt{f},\mathtt{g}],[\mathtt{h}^{+},\mathtt{f},\mathtt{g}])$. The two lists of children $[\mathtt{f},\mathtt{g}]$ and $[\mathtt{h}^{+},\mathtt{f},\mathtt{g}]$, are unified by by *dup* into $[\mathtt{h}^{+},\mathtt{f},\mathtt{g}]$, since *dup* makes use of *zip*, which recognises insertion tags. We thus have

$$(t,[\langle\mathtt{b}\rangle[\mathtt{c},\mathtt{d}],\langle\mathtt{e}\rangle[\mathtt{h}^{+},\mathtt{f},\mathtt{g}]])$$

Finally, $ch^{\smile}$ is applied. The updated source is

$$\langle\mathtt{a}\rangle[\langle\mathtt{b}\rangle[\mathtt{c},\mathtt{d}],\langle\mathtt{e}\rangle[\mathtt{h}^{+},\mathtt{f},\mathtt{g}]]$$

Through the example, two points are worth noticing. Firstly, to update through sequentially composed filters, we (at least in the specification level) regenerate the original intermediate values, and use them to generate update intermediate values. Backward updating for composition is defined similarly in [17] [10] by hand. The updating behaviour in our embedding, on the other hand, follow naturally from the definition of composition and its extended semantics.

Secondly, $concatx^{\smile}$ made a biased choice of joining the newly added item to the right. However, it is not always the preferred choice. Consider $children\,\hat{;}\,txt\,\mathsf{a}$ and input $t' = \langle\mathsf{r}\rangle[\mathsf{a},\mathsf{a}]$, output $(t',[\mathsf{a},\mathsf{a}])$, and edited output $(t',[\mathsf{a},\mathsf{a}^{+},\mathsf{a}])$. In the backward run, $concatx^{\smile}$ would produce the intermediate result $[[\mathsf{a}],[\mathsf{a}^{+},\mathsf{a}]]$ and attempt to match it with the old result of $\mathsf{txt\,a}$. However, the singleton filter $\mathsf{txt\,a}$ never returns a list with two elements.

When the second component in the sequential composition is a singleton filter, we shall switch to *concatw* which, in the situation above, would produce $[[\mathsf{a}],[\mathsf{a}]^{+},[\mathsf{a}]]$. In the implementation we can perform a static analysis distinguishing between singleton and non-singleton filters, and choose the preferred version of concatenation.

5.3 Embedding Other Filter Combinators

The embedding of $\mathsf{mkElem}\,s\,fs$ makes use of *concatx* and reverse application in a way similar to sequential composition. The auxiliary function *appF* applies all the filters in turn, before *concatx* concatenate their results. We then use $(appF\ fs)^{\smile}$ to consume the un-concatenated list of lists.

$$\begin{aligned}
\lceil \mathsf{mkElem}\,s\,fs\rceil &= appF\ fs; (id \times concatx);\\
&\quad assocl; ((appF\ fs)^{\smile} \times dupStr\ s); swap;\\
&\quad node; wrap\\
appF\ [\,] &= dupNil\\
appF\ (f{:}fs) &= dup; (\lceil f\rceil \times appF\ fs); trans;\\
&\quad (dup^{\smile} \times cons)
\end{aligned}$$

The filter *chip* can be defined in a number of ways. The following definition makes use of $\rhd$: we apply f to every result of node destructor $node^{\smile}$, and use the auxiliary function *cap* to place the result under the original root.

$$\begin{aligned}
\lceil \mathsf{chip}\,f\rceil &= (node^{\smile} \rhd \lceil f\rceil); cap \cup\\
&\quad isStr; dup; (id \times wrap)\\
cap &= (node^{\smile} \times id); assocr; (\delta \times id);\\
&\quad trans; (node \times node; wrap)
\end{aligned}$$

The embedding of $f\ |||\ g$ uses *catx* to concatenate the results of f and g. We also make use of $\lceil f\rceil^{\smile}$ to consume the garbage output of *catx*. The filter cat, on the other hand, is

$$\begin{aligned}
\lceil f\ |||\ g\rceil &= \lceil f\rceil; (\lceil g\rceil \times id); assocr;\\
&\quad (id \times swap; catx); assocl; (\lceil f\rceil^{\smile} \times id)\\
\lceil \mathsf{cat}\,[f]\rceil &= \lceil f\rceil\\
\lceil \mathsf{cat}\,(f{:}fs)\rceil &= \lceil f\ |||\ cat\,fs\rceil
\end{aligned}$$

The with and $\mathsf{without}$ combinators are defined using the $\rhd$ operator used in the definition of sequential composition. The auxiliary definitions *dom* and *notdom* checks whether the input is in the domain of g.

$$\begin{aligned}
\lceil f\ \mathsf{with}\ g\rceil &= \lceil f\rceil \rhd dom\ \lceil g\rceil\\
dom\ g &= g; (dupNil^{\smile}; dupNil \cup\\
&\quad (id \times cons^{\smile}; cons); dupfst;\\
&\quad (g^{\smile} \times wrap))
\end{aligned}$$

$$\begin{aligned}
\lceil f\ \mathsf{without}\ g\rceil &= \lceil f\rceil \rhd notdom\ \lceil g\rceil\\
notdom\ g &= g; ((id \times cons^{\smile}; cons); dupNil \cup\\
&\quad dupNil^{\smile}; dupNil; dupfst; (id \times wrap))
\end{aligned}$$

where *dupfst* duplicates the first component of the input and is defined by $dupfst = (dup \times id); assocr; (id \times swap); assocl$.

Finally, $\_?\rangle\ \_:\rangle\_$ is defined using union.

$$\begin{aligned}
\lceil p?\rangle\ f :\rangle g\rceil &= dom\ \lceil p\rceil; \lceil f\rceil \cup notdom\ \lceil p\rceil; \lceil g\rceil\\
\textbf{where}\ dom\,p &= p; (id \times cons^{\smile}; cons); p^{\smile}\\
notdom\,p &= p; dupNil^{\smile}; dupNil; p^{\smile}
\end{aligned}$$

5.4 Get and Put

For a transformation x, we define the GET and PUT functions to be:

$$\begin{array}{lcl}
\mathit{norm}\,[\,] & = & [\,] \\
\mathit{norm}\,(a^+ : x) & = & \mathit{norm}\;a : \mathit{norm}\;x \\
\mathit{norm}\,(a^- : x) & = & \mathit{norm}\;x \\
\mathit{norm}\,(a : x) & = & \mathit{norm}\;a : \mathit{norm}\;x \\
\mathit{norm}\,(a, b) & = & (\mathit{norm}\;a, \mathit{norm}\;b) \\
\mathit{norm}\,(\langle a\rangle x) & = & \langle \mathit{norm}\;a\rangle(\mathit{norm}\;x) \\
\mathit{norm}\,(a^+) & = & \mathit{norm}\;a \\
\mathit{norm}\,(a^-) & = & \mathit{norm}\;a \\
\mathit{norm}\,(*a) & = & a \\
\mathit{norm}\;a & = & a
\end{array}$$

Fig. 9 Definition of *norm*.

$$\begin{array}{lcl}
\mathit{get}_x\;s & = & \mathit{snd}\,([\![\lceil x\rceil]\!]\;s) \\
\mathit{put}_x\;s\;v' & = & \mathit{norm}\,([\![\lceil x\rceil^{\smile}]\!]\,(s, v'))
\end{array}$$

where $\mathit{snd}\,(s, v) = v$, and the function *norm* removes the tags in the tree and produces a normal form, defined in the obvious way in Figure 9. We start with a source document and use get_x to produce an initial view. After each editing action, put_x is called (with a cached copy of the source) to update the source, before get_x to produce a new view.

Let relation composition be defined by $R; S = \{(a, c) \mid \exists b \cdot (a, b) \in R \wedge (b, c) \in S\}$, *untagged* a partial function maps the input to itself if it does not contain tags, and $\mathit{dom}\;R = \{(a, a) \mid \exists b \cdot (a, b) \in R\}$, the operator taking the domain of a relation. An important result in [19] is that the following properties hold:

$$\mathit{untagged}; [\![x]\!]; [\![x^{\smile}]\!]; \mathit{norm} = \mathit{untagged}; \mathit{dom}\,[\![x]\!]$$
$$[\![x^{\smile}]\!]; \mathit{norm}; [\![x]\!]; [\![x^{\smile}]\!]; \mathit{norm} \subseteq [\![x^{\smile}]\!]; \mathit{norm}$$

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

5.5 Examples

Back to the example

$$f = \mathsf{mkElem}\;\mathtt{m}\;[\mathsf{children}\,\hat{;}\,\mathsf{tag}\;\mathsf{a}, \mathsf{children}]$$

For brevity, let $a = \langle \mathsf{a}\rangle[\,]$, $b = \langle \mathsf{b}\rangle[\,]$. Also let $a_1 = \langle \mathsf{a}\rangle[\mathsf{c}]$, $b_1 = \langle \mathsf{b}\rangle[\mathsf{c}]$ to be distinguished from a and b. Let $t = \langle \mathtt{r}\rangle[b, a]$ be the input. Calling $\mathit{get}_f\;t$ yields the pair $(t, [\langle \mathtt{m}\rangle[a, b, a]])$.

Now assume that the user deletes b in the view $\langle \mathtt{m}\rangle[a, b, a]$ and we perform a put_f. Applying $\lceil f\rceil^{\smile}$ to $(t, [\langle \mathtt{m}\rangle[a, b^-, a]])$ results in $\langle \mathtt{r}\rangle[b^-, a]$ — it is correctly inferred that b in the original tree shall be deleted. The *norm* function then actually removes the tagged b, and the updated source is $\langle \mathtt{r}\rangle[a]$.

Applying $\lceil f\rceil^{\smile}$ to $(t, [\langle \mathtt{m}\rangle[a, {b_1}^+, b, a]])$ yields $\langle \mathtt{r}\rangle[{b_1}^+, b, a]$. The inserted b_1, by the biased choice of *concatx*, is assumed to be a result of children.

If we insert a_1 instead, $\lceil f\rceil^{\smile}\,(t, [\langle \mathtt{m}\rangle[a, {a_1}^+, b, a]])$ yields $\langle \mathtt{r}\rangle[{a_1}^+, b, a]$. The next round of get_f results in $\langle \mathtt{m}\rangle[a_1, a, a_1, b, a]$ as the new view. If the user insert a_1 to the head of the list, on the other hand, the newly inserted a_1 has to be the result of children $\hat{;}$ tag a. Indeed, $\lceil f\rceil^{\smile}\,(t, [\langle \mathtt{m}\rangle[{a_1}^+, a, b, a]])$ yields $\langle \mathtt{r}\rangle[b, {a_1}^+, a]$ because a_1 shall be inserted in front of a. Elements inserted before the first a in the view must have an a label too, otherwise it could not have been the result of tag a.

The above cases shows that the default behaviour of our embedding mostly coincides with what we expect. If the programmer want some non-default behaviour, we need to use a different embedding of tag a whose injective semantics is the same but differs in the extended semantics. The actual implementation of Inv allows us to add more primitives. Since compound filter combinators preserve bidirectionality, one just need to be sure that the new primitives satisfy the healthiness conditions. This can be seen as adding extra annotations to the transformation to alter the default behaviour.

The bigger example in Figure 6 also work as we would expect. The use of tr = mkElem `tr` in mkRow specifies that an entire row must be added at once to the table, because a newly added element under `table` must be a result of mkElem `tr` with three sub-filters. As we expect, if we add a new `tr` entry to the constructed table, the *put* direction adds the new information to the original list. The index is updated in the next *get*. If we add a new name to the `ul` list, an entry with undefined `email` and `tel` fields is added to the source document.

However, one has to program the transform with care. There are more than one way to build the same table in the forward direction. For example, we could have simply scan through the address book and replace every `person` with `tr`, every `name`, `email`, and `tel` with `td`. However, such a transformation does not enforce enough structure on the input, and from it we cannot infer how the source is supposed to look like. Its reverse, therefore, does not yield meaningful results. It would help if the system knows the type of the source document.

6 Related Work

View-updating: to correctly reflect the modification on the view back to the database [3] [7] [9] [20] [1], is an old problem in the database community. In recent years, however, the need to synchronise data related by some transform starts to be recognised by researchers from different fields. In tools for aspect-oriented programming it is helpful to have multiple views of the same program [15]. In editors such as [23] [22] the user edits a view computed from the source by a transformation. Recent research on code clone [11] argues that a certain proportion of code in a software resembles each other, and it may help to develop software maintenance tools that keep the resembling pieces of code updated when one of them is altered. We are also developing file browsers using similar technique. It is argued in [16] that such *coupled transformation* problems are widespread and diverse.

In the context of data synchronisation, similar challenge was identified by [10] and coined the "bidirectional updating" problem. In [10] [8], a semantic foundation and a programming language (the "lenses") for bidirectional transformations are given. They form the core of the data synchronisation system Harmony [21]. Another very much related language was given by Meertens [17] to specify constraints in the design of user-interfaces. Due to their intended applications, less efforts were put on describing either element-wise or structural dependency inside the view.

The original motivation of our work was to build a theoretical foundation for presentation-oriented editors supporting interactive development of XML documents [2] [23] [22]. Proxima [22] is a presentation-oriented generic editor, to which one can "plug-in" their own editors for different types of documents and representations. However, it requires explicit specification of both forward and backward updating. Our goal is to specify only the forward transform and derive the backward updating automatically. We choose to based our formalisation of bidirectional updating on injective mapping. The extension to deal with duplication and structural changes are thus easier to cope with.

We have also developed a domain-specific XML processing language, called X. The language, basically a point-free functional language closely related to the languages in [17] and [10], is currently used in our XML editor [13] as the language to describe transformations with. In [13], the semantics of X was given without the use of Inv. In a preliminary work [12] in a non-refereed workshop, we drafted an implementation of bidirectional HaXML. In both cases, however, the treatments with duplication and alignment were not satisfactory. In order to resolve the problem, we attempt to embed both HaXML and X into Inv. The embedding of HaXML is recorded in this paper, while that for X is described in a paper in preparation [14].

7 Conclusions and Future Work

We have presented an embedding of HaXML into Inv. With the embedding, existing HaXML transformations gain bidirectionality — the forward transform induces a backward transform which maps an edited view to an updated source. This makes HaXML be a more powerful transformation language than it was first designed for. As far as we are aware, this is the first attempt towards systematically bidirectionalising unidirectional languages.

At present, we have an Inv interpreter implemented in Haskell, in which both X [13] [14] and HaXML is embedded. The prototype is available from the authors' homepage[†4]. The HaXML embedding is in a relatively preliminary stage. The main difficulty of the HaXML embedding is that filters return a list of results, and the length of the list is fixed for singleton filters.

Could we have skipped the Inv layer and implement the backward updating in Haskell directly? Yes, but we prefer the current approach for two reasons. Firstly, rather then specifying in each reversed HaXML construct how to deal with insertion, deletion, and label editing, all the details are hidden in the Inv layer. Each embedding is designed mainly based on its forward semantics. We believe that it is a more elegant approach. Secondly, it integrates the bidirectionalisation of HaXML into a more general framework, which is the motivation of our research in the first place.

A question is: what can we say about the up-

†4 `http://www.ipl.t.u-tokyo.ac.jp/~ scm/`

dated source? The backward transformation does not in general always yield a result — some editing actions may be considered illegal. The Put-Get-Put property merely guarantees that if the backward transformation yields any source at all, it is well-behaved in the sense that an additional *get* followed by a *put* results in the same source, therefore no repeated updating is necessary. Exactly which source is returned is determined by the algebraic rules of the Inv primitives.

Apart from that, we assume no external criteria on the updated source. In [17], Meertens proposed the principle of *minimal change* — that a source shall be chosen such that minimal change is made to the view. The main difficulty is that the minimal change principle, in general, is not preserved by compound transformations. In [22] it was also shown that a minimal change is not always what the user wants. It will be an interesting challenge to develop a formalisation of bidirectional updating that maintains some external measurement on the chosen source.

Ideally, given a forward transformation, we wish to get the backward transformation for free by the embedding. As the examples show, however, transformations written without concern of backward updating in mind tend to lack necessary information. The experience gained from this case study, however, may help us in the development of a language designed for bidirectional updating. Currently we are studying what language design/features are suitable for bidirectionalisation, and investigating how type information helps in the inference of backward transformation.

Acknowledgments

The authors would like to thank the members of the PSD project — Yasushi Hayashi, Dongxi Liu, Keisuke Nakano, and Shingo Nishioka, for stimulating discussions. Also thanks to Kento Emoto for testing and debugging the HaXML embedding, and Kazutaka Matsuda for reviewing an earlier draft of this paper. Together with Akimasa Morihata they worked on the language *X* and efficiently implemented a prototype XML editor showing that the whole idea is feasible.

References

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

[2] Altova, Co.: XMLSpy. `http://www.xmlspy.com/products_ide.html`.

[3] Bancilhon, F. and Spyratos, N.: Update semantics of relational Views, *ACM Transactions on Database Systems*, Vol. 6, No. 4(1981), pp. 557–575.

[4] Bennett, C. H.: Logical reversibility of computation, *IBM Journal of Research and Development*, Vol. 17, No. 6(1973), pp. 525–532.

[5] Benzaken, V., Castagn, G. and Frisch, A.: CDuce: an XML-centric general-purpose language, in *Proceedings of the 2003 ACM SIGPLAN International Conference on Functional Programming*, ACM Press, 2003.

[6] Bray, T., Paoli, J., Sperberg-Macqueen, C. M. and Maler, E.: Extensible Markup Language (XML) 1.0 (Second Edition), October 2000. `http://www.w3.org/TR/REC-xml`.

[7] Dayal, U. and Bernstein, P. A.: On the correct translation of update operations on relational views, *ACM Transactions on Database Systems*, Vol. 7, No. 3(1982), pp. 381–416.

[8] Foster, J. N., Greenwald, M. B., Moore, J. T., Pierce, B. C. and Schmitt, A.: Combinators for bi-Directional tree transformations: a linguistic approach to the view update problem, in *The 32nd ACM SIGPLAN–SIGACT Symposium on Principles of Programming Languages* (*POPL 2005*), Long Beach, California, ACM Press, 2005, pp. 233–246.

[9] Gottlob, G., Paolini, P. and Zicari, R.: Properties and update semantics of consistent views, *ACM Transactions on Database Systems*, Vol. 13, No. 4(1988), pp. 486–524.

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

[11] Higo, Y., Kamiya, T., Kusumoto, S. and Inoue, K.: ARIES: refactoring support environment based on code clone analysis, in *The 8th IASTED International Conference on Software Engineering and Applications*(*SEA 2004*), Cambridge, USA, ACTA Press, November 9-11, 2004, pp. 222–229.

[12] Hu, Z., Emoto, K., Mu, S.-C. and Takeichi, M.: Bidirectionalizing Tree Tranformations, in *Workshop on New Approaches to Software Construction* (*WNASC 2004*), Komaba, Tokyo, Japan, September 13–14, 2004.

[13] Hu, Z., Mu, S.-C. and 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*, Verona, Italy, ACM Press, August 2004.

[14] Hu, Z., Mu, S.-C. and Takeichi, M.: A programmable editor for developing structured documents based on bidirectional transformations, 2006. Submitted to *Higher-Order and Symbolic Computation*.

[15] Janzen, D. and de Volder, K.: Programming with crosscutting effective views, in *ECOOP 2004 - Object-Oriented Programming, 18th European Conference*, Lecture Notes in Computer Science, No. 3086, Springer-Verlag, June 14-18, 2004, pp. 195–218.

[16] Lämmel, R.: Coupled software transformations (extended abstract), in *First International Workshop on Software Evolution Transformations*, 2004.

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

[18] Mu, S.-C., Hu, Z. and Takeichi, M.: An Injective Language for Reversible Computation, in *Seventh International Conference on Mathematics of Program Construction*, Lecture Notes in Computer Science, No. 3125, Springer-Verlag, July 2004.

[19] Mu, S.-C., Hu, Z. and Takeichi, M.: An algebraic approach to bi-directional updating, in *The Second Asian Symposium on Programming Language and Systems*(Chin, W.-N.(ed.)), Lecture Notes in Computer Science, No. 3302, Springer-Verlag, November 4-6, 2004, pp. 2–20.

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

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

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

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

[24] Wallace, M. and Runciman, C.: Haskell and XML: generic combinators or type-based translation?, in *Proceedings of the 1999 ACM SIGPLAN International Conference on Functional Programming*, ACM Press, September 1999, pp. 148–159.