Bidirectionalising HaXML

Shin-Cheng Mu Zhenjiang Hu Masato Takeichi

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

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, 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.

Keywords XML, bidirectional updating, domain-specific language, reversible computation

1. Introduction

XML [5], partly stimulated by the growth of the Web and e-commerce, has emerged as the *de facto* standard for representation of structured data and information interchange. Many organizations 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 (structure) 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 [12]. 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 person will be added to the person bodies in both the source document and the view.

This so-called *bidirectional updating* problem (coined by, to the best of the authors' knowledge, [9]) is attracting lots of interests recently, as people identified various situations where one wants to transform some

ICFP '05 September 26-28, 2005, Tallinn, Estonia.
Copyright © 2005 ACM supplied by printer. . . $5.00.

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

Figure 1. An XML document representing an address book.

```
<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.i.u-tokyo.ac.jp </td>
      <td> +81-3-5841-7411 </td>
  </tr>
  <tr><td> Zhenjiang Hu </td>
      <td> hu@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>
```

Figure 2. A wiew of the address book in HTML.

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, which is known as *view updating* in the database commuinity [3, 6, 8, 19, 1]. One may want to synchronize the bookmark files of several different web browsers (on different machines) [9], 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 [12] 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 [9, 16, 18, 12] 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 an existing transform written in tree-transformation languages, like XSLT, 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 [23] 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. 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 the HaXML combinators is usually more compact than its equivalent in DOM, SAX, or XSLT. For the rest of the paper when we talk about HaXML, we will be refering 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, however, 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.

In [18], we developed a language Inv to deal with bidirectional updating, paying special attention to the handling of duplication and structural alignments. The development of Inv takes a layered approach. In the original semantics of Inv, the programmer is allowed to define injective functions only. In an extended semantics, the *reverse* of every Inv function maps an edited output, which might not be in the range of the function, to a reasonable input. In this paper, we add another layer by developing an embedding of HaXML to Inv. Therefore, when the programmer designs a forward transformation, we get a backward transformation for free.

The rest of the paper is organized as follows. We start by briefly reviewing the core of HaXML [23], a general-purpose unidirectional transformation language, 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.

2. Tree Documents and Tree Transformations

In this section we will briefly review the core components of the combinator library of HaXML [23]. An XML content is either an element or a text fragment. An element consists of a tag and a sequence of contents. HaXML represents XML documents by native Haskell data structure. For presentation of this paper, we will use a more simplified representation of XML trees. The languages defined in this paper will deal with a range of values defined by the syntax below:

$$\begin{array}{lcl} V & ::= & \mathit{String} \mid [V] \mid (V, V) \mid T \\ T & ::= & \mathsf{N}\,\mathit{String}\,[T] \mid \mathsf{L}\,\mathit{String} \\ [a] & ::= & [\,] \mid a : [a] \end{array}$$

```
addrbook = N Addrbook
  N Person
    [N Name [Shin-Cheng Mu],
     N Email [scm@ipl.i.u-tokyo.ac.jp],
     N Tel [+81-3-5841-7411]],
  [N Person
    [N Name [Zhenjiang Hu],
     N Email [hu@mist.i.u-tokyo.ac.jp],
     N Tel [+81-3-5841-7411]],
  N Person
    [N Name [Masato Takeichi],
     N Email [takechi@mist.i.u-tokyo.ac.jp],
     N Tel [+81-3-5841-7411]]
  ]
```

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

For the purpose of 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 $[a]$, and trees. A tree is either a leaf, or a node with a label and a list of subtrees. When it is clear from the context we would omit the L constructor to save space. This rather simplified view of XML omits some features, such as attributes, that are trivial to add, and some features such as IDRefs, which will be our future work. The range of values will be further extended in Section 4 to record user editing.

Figure 3 gives an example of this representation of the the document source in Figure 1.

2.1 Tree Transformations

Combinators in HaXML are called *filters*. They have type $T \to [T]$, taking a tree and returning a sequence of tree. The result might be empty, a singleton list, or a collection of trees.

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 if the input satisfies certain predicate, or empty list otherwise. In this paper we will call them *singleton* filters.

Other filters do not have fixed constraints on the length of the output list. 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 $f \mathbin{⨾} g$ applies f to the input, before applying g to 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`. In [23], 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.

The combinator $f \mathbin{|||} g$ concatenates the results of filters f and g, while cat fs is a generalisation of $|||$ 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 to the input otherwise. The expression

Predicates:

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

Selection:

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

Construction:

literal	::	$String \rightarrow Filter$	{ build plain text }
mkElem	::	$String \rightarrow [Filter] \rightarrow Filter$	{ build a tree with an inner node }
replaceTag	::	$String \rightarrow Filter$	{ replace root's tag }

Figure 4. Basic filters.

$\fatsemi$	::	$Filter \rightarrow Filter \rightarrow Filter$	{ sequential composition }
$(\vert\vert\vert)$	::	$Filter \rightarrow Filter \rightarrow Filter$	{ append results }
cat	::	$[Filter] \rightarrow Filter$	{ concatenate ressults }
with	::	$Filter \rightarrow Filter \rightarrow Filter$	{ guard }
without	::	$Filter \rightarrow Filter \rightarrow Filter$	{ negative guard }
et	::	$(String \rightarrow Filter) \rightarrow Filter \rightarrow Filter$	{ disjoint union }
$\_?\rangle\ \_\ :\rangle\_$	::	$Filter \rightarrow Filter \rightarrow Filter \rightarrow Filter$	{ condition }
chip	::	$Filter \rightarrow Filter$	{ in-place children application }

Figure 5. Basic filter combinators.

$p?\rangle\, f\ :\rangle g$ 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. For example, the filter mkElem m [children $\fatsemi$ tag a, children], applied to input N r [N a [ ], N b [ ]], produces N m [N a [ ], N a [ ], N b [ ]]. The first child, N a [ ], results from children $\fatsemi$ tag a, while the rest result from children.

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 $\langle/$.

$$
\begin{aligned}
f \mathbin{/\rangle} g &= f \fatsemi \mathsf{children} \fatsemi g \\
f \mathbin{\langle/} g &= f\ \mathsf{with}\ (\mathsf{children} \fatsemi g)
\end{aligned}
$$

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): $/\rangle$ is an 'interior' selector, returning the inner structure; $\langle/$ is an 'exterior' selector, returning the outer structure.

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

$$
\mathsf{deep}\, f = f ?\rangle\, f\ :\rangle (\mathsf{children} \fatsemi \mathsf{deep}\, f)
$$

```
<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>
```

Figure 6. A transformation in XSLT.

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.

$$\mathsf{foldXml}\, f \;=\; (\mathsf{chip}\,(\mathsf{foldXml}\, f)) \fatsemi f$$

Consider the transformation in XSLT in Figure 6, which can map the XML address book in Figure 1 to the HTML document in Figure 2. We can define it in HaXML as in Figure 7.

3. The Bidirectional Updating Problem

Consider again the filter

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

upon receiving a source document $\mathsf{N}\,\mathtt{r}\,[\mathsf{N}\,\mathtt{a}\,[\,], \mathsf{N}\,\mathtt{b}\,[\,]]$, producing the view $\mathsf{N}\,\mathtt{m}\,[\mathsf{N}\,\mathtt{a}\,[\,], \mathsf{N}\,\mathtt{a}\,[\,], \mathsf{N}\,\mathtt{b}\,[\,]]$.

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

If the user changes the view to $\mathsf{N}\,\mathtt{m}\,[\mathsf{N}\,\mathtt{a}\,[\,], \mathsf{N}\,\mathtt{c}\,[\,], \mathsf{N}\,\mathtt{b}\,[\,]]$, the system should PUT the view to $\mathsf{N}\,\mathtt{r}\,[\mathsf{N}\,\mathtt{c}\,[\,], \mathsf{N}\,\mathtt{b}\,[\,]]$. In the next GET the filter $\mathsf{tag}\,\mathtt{a}$ would produce a null result and the resulting view would be $\mathsf{N}\,\mathtt{m}\,[\mathsf{N}\,\mathtt{c}\,[\,], \mathsf{N}\,\mathtt{b}\,[\,]]$.

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

Figure 7. A transformation in HaXML.

Had the user changed the view to N m [N c [ ], N a [ ], N b [ ]], however, the system may choose to declare that this is an invalid change and warn the user, since tag a could not have produced N c [ ].

Apart from editing labels, the user is allowed to insert new elements too. Assume a new element N b [ ] is inserted between the results of tag a and children (inserting into the result of the latter is relatively easier to deal with), resulting in N m [N a [ ], N b [ ], N a [ ], N b [ ]]. The reasonable choice is to assume that N b [ ] is the result of children and produces the source N r [N b [ ], N a [ ], N b [ ]]. If the inserted element is N a [ ], however, the system needs to make a biased choice.

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. The updated source, after a GET operation, results in an update view. 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 [16]). 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: $get_x :: S \to V$ defines the transformation from the source to the view, while $put_x :: (S \times V) \to S$ takes the original source and an edited view, and returns an updated source.

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

$$\begin{array}{lll} \text{GET-PUT-GET}: & get_x\,(put_x\,s\,v) = v & \text{where } v = get_x\,s \\ \text{PUT-GET-PUT}: & put_x\,s'\,(get_x\,s') = s' & \text{where } s' = put_x\,s\,v \end{array}$$

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'

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

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

$$
\begin{array}{lll}
[\![f;g]\!]\,x & = & [\![g]\!]\,([\![f]\!]\,x) \\
[\![f \times g]\!]\,(a,b) & = & ([\![f]\!]\,a, [\![g]\!]\,b) \\
[\![f \cup g]\!] & = & [\![f]\!] \cup [\![g]\!], \\
\multicolumn{3}{l}{\text{if } \mathit{dom}\,f \cap \mathit{dom}\,g = \mathit{ran}\,f \cap \mathit{ran}\,g = \emptyset}
\end{array}
$$

$$
\begin{array}{lll}
[\![\mathit{swap}]\!]\,(a,b) & = & (b,a) \\
[\![\mathit{assocr}]\!]\,((a,b),c) & = & (a,(b,c)) \\
[\![\mathit{assocl}]\!]\,(a,(b,c)) & = & ((a,b),c)
\end{array}
\qquad
\begin{array}{lll}
[\![\mathit{isStr}]\!]\,s & = & s \text{ if } s \text{ is a string} \\
[\![\mathit{neq}]\!]\,(a,b) & = & (a,b) \text{ if } a \neq b \\
[\![f^{\smile}]\!] & = & [\![f]\!]^{\circ} \\
[\![\mu F]\!] & = & [\![F\,\mu F]\!]
\end{array}
$$

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

Figure 8. The language Inv and its semantics when restricted to values without editing tags.

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 [16, 9] 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 our examples above. The GET-PUT property (which actually holds for our HaXML 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

The language Inv was developed to study the bidirectional updating behaviour. In [17], Inv was designed to be a programming language defining injective functions only, due to the our belief that a study of bidirectional updating can be made more elegant if we first concentrate on injective functions. In [18], the language was given an extended semantics, where every Inv expression of type $A \to B$ induces a binary relation between A and B, mapping the edited values in A to a reasonable choice of source in B. The sometimes biased choice can be inferred by algebraic rules.

In Section 4.1 we give a brief introduction to Inv, and in Section 4.2 we discuss how its extended semantics handles duplication and structural alignment. To be self-contained, we provide a brief, simplified summary about the extended semantics in Section 4.3. The reader is referred to [18] for a more complete account.

4.1 The Language Inv

Shown in Figure 8 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 [18]. 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 by the algebraic rules in to be mentioned in Section 4.3.

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

The function *id* is the identity function, the unit of composition. The function *isString* is a subset of *id* returning strings only, while *neq* returns a pair unchanged only of the two components are not the same. Function composition is denoted by semicolon. Union of relations, modelling conditional branches, is simply defined as set union, with a restriction that the two relations have disjoint domains and ranges. The product $(f \times g)$ takes a pair and applies f and g to the two components respectively. Note that composition binds tighter than product. Therefore $(f; g \times h)$ should be bracketed as $((f; g) \times h)$.

The functions *swap*, *assocl* and *assocr* distributes the components of the input pair. All functions that move around the components in a pair can be defined in terms of products, *assocr*, *assocl*, and *swap*. We find the following functions useful:

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

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

The *converse* of a relation R is defined by

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

In the injective semantics, the *reverse* operator $(\_)^{\smallsmile}$ corresponds to converses on relations. The reverse of *cons*, for example, decomposes a non-empty list into the head and the tail. The reverse of *nil* matches only the empty list and maps it to the unit value. The reverse of *swap* is itself, and *assocr* and *assocl* are reverses of each other. The reverse operator distributes into composition, products and union by the following rules, all implied by the semantics definition $[\![f^{\smallsmile}]\!] = [\![f]\!]^{\circ}$:

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

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

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

A number of list processing functions can be defined using the fixed-point operator. The standard functions *foldr*, *map*, and *unzip* (transposing a list of pair to a pair of lists) can be defined exactly as the point-free counterpart of their usual definitions:

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

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 $a \rightarrow (a \times a)$:

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

In particular, 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:

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

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

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

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:

$$\begin{array}{lcl} V & ::= & A \mid V^{+} \mid V^{-} \mid [V] \mid (V,V) \mid T \\ T & ::= & \mathsf{N}\, A\, [T] \\ [a] & ::= & [\,] \mid a : [a] \\ A & ::= & String \mid\; *String \mid \top \end{array}$$

The *editing tags* $(\_)^{+}$, $(\_)^{-}$, $*(\_)$, and $\top$ are used to record the action performed by the user of the editor. The $*(\_)$ tags applies to atomic values (strings). When the user changes the value of a string the editor marks the string with the $*(\_)$ tag. The $(\_)^{+}$ tag indicates that the tagged element is newly inserted by the user. When the user deletes an element it is wrapped by a $(\_)^{-}$, keeping note that it ought to be deleted but we temporary leave it there for further processing. The symbol $\top$ denotes an unconstrained value, to be further refined[1]. Values containing any of the tags are called *tagged*, otherwise they are *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. In the next section we will give it a more formal account.

As described in the previous section, $\delta^{\smile}$ is a partial function performing equality test. The two components in the pair are compared, and one of them is returned only when they are identical. When the user edits an atomic value, the action is recorded by a $*(\_)$ tag. In the extended semantics, we generalise δ such that it recognises the tag:

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

When the two values are not the same but one of them was edited by the user, the edited one gets precedence and goes through. Therefore $(*n, m)$ is mapped to $*n$. If both values are edited, however, they still have to be the same.

[1] In the semantics in [18], there is no $\top$. Instead, a relation may non-deterministically map the input to many outputs, and refined when composed with other relations. However, $\top$ was indeed used in the implementation.

```
$concatw = \mu(X :$
 $nil\breve{}; nil; dup \cup$
 $cons\breve{}; ($
   $swap; dupNil\breve{}; X; (dupNil; swap; cons \times id) \cup$
   $(wrap\breve{} \times id); (dup; (wrap \times id) \times X);$
     $trans; (cons \times cons) \cup$
   $(cons\breve{} \times id); assocr;$
     $(dup \times cons; X; (revcons \times id));$
     $trans; (assocl; swap \times id); assocr;$
     $(id \times (cons \times cons)); assocl;$
     $(swap; cons \times id)))$
```

Figure 9. Definition of *concatw*.

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

One of the main achievement of [18] is that, using the same definition of *unzip* above with a small amount of annotations, *zip* in the extended semantics knows how to zip together two lists when they contain inserted or deleted elements. For example, $unzip\,[(1,\mathtt{a}),(2,\mathtt{b}),(3,\mathtt{c})]$ yields $([1,2,3],[\mathtt{a},\mathtt{b},\mathtt{c}])$. If we label one element with a delete tag, $zip\,([1,2^-,3],[\mathtt{a},\mathtt{b},\mathtt{c}])$ yields $[(1,\mathtt{a}),(2,\mathtt{b})^-,(3,\mathtt{c})]$ — the corresponding element is deleted as well. If we insert an element, say $([1,2,3],[\mathtt{a},\mathtt{b},\mathtt{d}^+,\mathtt{c}])$, zipping them together yields $[(1,\mathtt{a}),(2,\mathtt{b}),(\top,\mathtt{d})^+,(3,\mathtt{c})]$. An unconstrained value is invented and paired with the newly inserted $\mathtt{d}$, and might later be further constrained by δ or other structural constraints.

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 where we have to deal with alteration of x, however, we have to make a biased choice when new items are inserted to the list. According to the semantics in [18], 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 $[[1,2],[5^+,3,4]]$.

However, such a choice is not always preferred. In some occasions we need to make the new element a stand alone singleton list. The following Haskell function *concatw* deals with singleton list separately:

$$
\begin{array}{lcl}
concatw\,[\,] & = & ([\,],[\,]) \\
concatw\,([\,]:xs) & = & (([\,]:) \times id)\,(concatw\,xs) \\
concatw\,([a]:xs) & = & (([a]:) \times (a:))\,(concatw\,xs) \\
concatw\,((a:b:x):xs) & = & ((\lambda((b:x):xs) \to (a:b:x):xs) \times (a:))\,(concatw\,((b:x):xs))
\end{array}
$$

Its Inv translation, given in Figure 9, coincides with *concatx* in the injective semantics. In the extended semantics, $concatw\breve{}$ breaks the list after newly inserted elements. For example, $([[1,2],[3,4]],[1,2,5^+,3,4])$ is mapped to $[[1,2],[5]^+,[3,4]]$. We need both *concatx* and *concatw* in the embedding.

4.3 The Extended Semantics

To be self-contained, in this section we give a brief, simplified summary of the results in [18], explaining how the extended semantics deals with bidirectional updating. The readers can safely skip this section at first reading.

The main instructive example will be *unzip*, defined in Section 4.1. Its reverse, $zip = unzip^{\smallsmile}$, according to the distributivity of $^{\smallsmile}$, is given by:

$$\begin{aligned} zip \;=\; & \mu(X\colon (nil^{\smallsmile} \times nil^{\smallsmile}); \delta^{\smallsmile}; nil \cup \\ & \quad (cons^{\smallsmile} \times cons^{\smallsmile}); trans; (id \times X); cons) \end{aligned}$$

The puzzle is: how to make it work correctly in the presence of $(\_)^{+}$ and $(\_)^{-}$ tags?

We introduce several additional operators:

- *del* and *ins*, parameterised by a value. The function $del\,a$ takes a list x to $a^{-} : x$, while $ins\,a$ takes x to $a^{+} : x$;
- fst_b and snd_a, defined by:

  $$\begin{aligned} fst_b\,(a,b) &= a \\ snd_a\,(a,b) &= b \end{aligned}$$

  That is, fst_b eliminates the second component of a positive pair only if it equals b. Otherwise it fails. Similarly, snd_a eliminates the first component of an ordinary pair only of it equals a. When interacting with existing operators, they should satisfy the algebraic rules in Figure 10, which are obviously true given their semantics.
- Also, we restrict the domain of $cons^{\smallsmile}$ to lists whose head is *not* tagged by either $(\_)^{+}$ or $(\_)^{-}$.

An extended *zip* capable of dealing with deletion can be extended, from the original *zip* by (here "..." denotes the original two branches of *zip*):

$$\begin{aligned} & \mu(X\colon \ldots \cup \forall a,b\cdot \\ & \quad ((ins\,a)^{\smallsmile} \times (ins\,b)^{\smallsmile}); X; ins\,(a,b) \;\cup \\ & \quad ((ins\,a)^{\smallsmile} \times isList); X; ins\,(a,b) \;\cup \\ & \quad (isList \times (ins\,b)^{\smallsmile}); X; ins\,(a,b) \;\cup \\ & \quad ((del\,a)^{\smallsmile} \times (del\,b)^{\smallsmile}); X; del\,(a,b) \;\cup \\ & \quad ((del\,a)^{\smallsmile} \times cons^{\smallsmile}; snd_b); X; del\,(a,b) \;\cup \\ & \quad (cons^{\smallsmile}; snd_a \times (del\,b)^{\smallsmile}); X; del\,(a,b)) \end{aligned}$$

where a and b are universally quantified, and $isList = nil^{\smallsmile}; nil \cup cons^{\smallsmile}; cons$, a subset of *id* letting through only lists having no tag at the head. Look at the branch starting with $((ins\,a)^{\smallsmile} \times (ins\,b)^{\smallsmile})$. It says that, given a pair of lists both starting with insertion tags a^{+} and b^{+}, we should deconstruct it, pass the tails of the lists to the recursive call, and put back an $(a,b)^{+}$ tag. If only the first of them is tagged (matching the branch starting with $((ins\,a)^{\smallsmile} \times isList)$), we temporarily remove the head a^{+}, recursively process the lists, and put back $(a,b)^{+}$ with a freshly generated b. It is non-deterministic which b is chosen, and might be further constrained when *zip* is further composed with other relations.

The situation is similar with deletion. In the branch starting with $((del\,a)^{\smallsmile} \times cons^{\smallsmile}; snd_b)$ where we encounter a list with an a deleted by the user, we remove an element in the other list and remember its value in b. Here universally quantified b is used to match the value — all the branches with different b's are unioned together, with only one of them resulting in a successful match. After processing it recursively, we cons the list with the head $(a,b)^{-}$ indicating that a pair (a,b) was removed from the resulting list.

It would be very tedious if the programmer has to explicitly write down these extra branches for all functions (let alone that we did not provide the construct for universal quantification.) We wish that *del, ins, fst* and *snd* do not appear in the programs, but the system can somehow derive the additional branches. Luckily, these additional branches can be derived automatically using the rules in Figure 10.

In the derivations later we will omit the semantics function $[\![\_]\!]$ and use the same notation for the language and its semantics, where no confusion would occur. This is merely for the sake of brevity.

In place of ordinary *cons*, we define two constructs addressing the dependency of structures. Firstly, the *bold* **cons** is defined by::

$$\begin{aligned} \mathbf{cons} \;=\; & cons \;\cup \\ & \textstyle\bigcup_{a::A}(snd_{a^{-}}; del\,a) \;\cup\; \bigcup_{a::A}(snd_{a^{+}}; ins\,a) \end{aligned}$$

Secondly, we define the following *sync* operator:

$$\begin{array}{lcl}
sync & = & (cons \times cons) \\
sync^{\smallsmile} & = & (cons^{\smallsmile} \times cons^{\smallsmile}) \\
\multicolumn{3}{l}{\quad \cup \ \bigcup_{a,b\in A}(((del\,a)^{\smallsmile}; snd_{a-}{}^{\smallsmile} \times (del\,b)^{\smallsmile}; snd_{b-}{}^{\smallsmile})} \\
& & \qquad \cup\,((del\,a)^{\smallsmile}; snd_{a-}{}^{\smallsmile} \times cons^{\smallsmile}; snd_{b-}; snd_{b-}{}^{\smallsmile}) \\
& & \qquad \cup\,(cons^{\smallsmile}; snd_{a-}; snd_{a-}{}^{\smallsmile} \times (del\,b)^{\smallsmile}; snd_{b-}{}^{\smallsmile})) \\
\multicolumn{3}{l}{\quad \cup \ \bigcup_{a,b\in A}(((ins\,a)^{\smallsmile}; snd_{a+}{}^{\smallsmile} \times (ins\,b)^{\smallsmile}; snd_{b+}{}^{\smallsmile})} \\
& & \qquad \cup\,((ins\,a)^{\smallsmile}; snd_{a+}{}^{\smallsmile} \times isList; snd_{b+}{}^{\smallsmile}) \\
& & \qquad \cup\,(isList; snd_{b+}{}^{\smallsmile} \times (ins\,b)^{\smallsmile}; snd_{b+}{}^{\smallsmile}))
\end{array}$$

In the definition of *zip*, we replace every singular occurence of *cons* with **cons**, and every $(cons \times cons)$ with *sync*. The definition of $sync^{\smallsmile}$ looks very complicated but we will shortly see its use in the derivation. Basically every produce correspond to one case we want to deal with: when both the lists are cons lists, or when one or both of them has a tagged value at the head.

After the substitution, all the branches can be derived by algebraic reasoning. The rules we need are listed in Figure 10. Only rules for *assocl* are listed. Free identifiers are universally quantified. The rules for *assocr* can be obtained by pre-composing *assocr* to both sides and use $asscor; assocl = id$. To derive the first branch for insertion, for example, we reason:

$$\begin{array}{ll}
 & zip \\
\supseteq & \quad \{\text{fixed-point}\} \\
 & sync^{\smallsmile}; trans; (id \times zip); \mathbf{cons} \\
\supseteq & \quad \{\text{since } sync^{\smallsmile} \supseteq ((ins\,a)^{\smallsmile}; snd_{a+}{}^{\smallsmile} \\
 & \qquad \times (ins\,b)^{\smallsmile}; snd_{b+}{}^{\smallsmile}) \text{ for all } a, b\} \\
 & ((ins\,a)^{\smallsmile} \times (ins\,b)^{\smallsmile}); (snd_{a+}{}^{\smallsmile} \times snd_{b+}{}^{\smallsmile}); \\
 & trans; (id \times zip); \mathbf{cons} \\
\supseteq & \quad \{\text{claim: } (snd_{a+}{}^{\smallsmile} \times snd_{b+}{}^{\smallsmile}); trans = (snd_{(a,b)+})^{\smallsmile}\} \\
 & ((ins\,a)^{\smallsmile} \times (ins\,b)^{\smallsmile}); (snd_{(a,b)+})^{\smallsmile}; (id \times zip); \mathbf{cons} \\
= & \quad \{\text{since } (f \times g); snd_{f\,a} = snd_a; g \text{ for total } f\} \\
 & ((ins\,a)^{\smallsmile} \times (ins\,b)^{\smallsmile}); zip; (snd_{(a,b)+})^{\smallsmile}; \mathbf{cons} \\
\supseteq & \quad \{\text{since } \mathbf{cons} \supseteq snd_{(a,b)+}; ins\,(a, b)\} \\
 & ((ins\,a)^{\smallsmile} \times (ins\,b)^{\smallsmile}); \\
 & zip; (snd_{(a,b)+})^{\smallsmile}; snd_{(a,b)+}; ins\,(a, b) \\
= & \quad \{\text{since } snd_x{}^{\smallsmile}; snd_x = id\} \\
 & ((ins\,a)^{\smallsmile} \times (ins\,b)^{\smallsmile}); zip; ins\,(a, b)
\end{array}$$

We get the first branch. The claim that $trans^{\smallsmile}; (snd_{a+} \times snd_{b+}) = snd_{(a,b)+}$ can be verified by the rules in Figure 10. In a similar fashion, all the branches can be derived dynamically.

The situation with *catx* is similar: $(cons \times cons)$ is interpreted as *sync*, and from which we can derive other branches needed to deal with tagged values. The good thing is that the particular choice *catx* make is inferred from the algebraic rules.

The algebraic rules can be applied both forwards and backwards, which seem to cause problems for automatic transformation. Luckily, it is possible to integrate these rules in an Inv interpreter. The details are given in [18].

$$\begin{array}{rcl}
(f \times g); fst_{(g\ b)} & = & fst_b; f, \ \text{ if } g \text{ is total} \\
(f \times g); snd_{(f\ a)} & = & snd_a; g, \ \text{ if } f \text{ is total} \\
swap; snd_a & = & fst_a \\
snd_a\breve{}; eq\, nil & = & (\lambda\, [\,] \rightarrow a) \\
assocl; (fst_b \times id) & = & (id \times snd_b) \\
assocl; (snd_a \times id) & = & (snd_a \cup snd_a) \\
assocl; snd_{(a,b)^\square} & = & snd_{a^\square}; (snd_{b^\square} \cup snd_b)
\end{array}$$

Figure 10. Algebraic rules. Here $(\lambda\, [\,] \rightarrow a)$ is a function mapping only empty list to a. The $\square$ may denote either a $(\_)^+$ or a $(\_)^-$ or nothing.

5. Bidirectionalisation Embedding

We are now ready to show how tree transformations in HaXML can be embedded into the bidirectional transformation language Inv. We call the type of source documents S and that of views V. They are both embedded in the type for trees but we nevertheless distinguish them for clarity. The trick is that every HaXML construct is embedded as an Inv expression denoting, in the injective semantics, a function of type $S \rightarrow (S \times [V])$ 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 $(S \times [V]) \rightarrow S$, maps the original source and its corresponding views back to the source. In the extended semantics, however, when given the original source and an *edited* view, the Inv expression magically produces an updated source consistent with the transform.

The embedding is presented in Section 5.1, 5.2, and 5.3. A pair of forward/backward transformations has to satisfy a set of healthiness constraints. This is given in Section 5.4.

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:

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

where $wrap = dupNil; cons$, wrapping an item into a singleton list. Other "singleton" filters — those returning either an empty list or a singleton list, are also defined in terms of *dup*, *dupNil*, and *wrap*:

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

where $isNode = node\breve{}; node$. The filter elm returns a singleton list only if the input is a node, while txt returns a singleton list only if the input is a string. In both cases *dup* is used to copy the input.

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

$$\begin{array}{rcl}
\lceil \mathsf{tag}\, s \rceil & = & node\breve{}; (strEq\, s \times id); node; dup; (id \times wrap) \cup \\
 & & node\breve{}; (strNEq\, s \times id); node; dup; (id \times wrap) \cup \\
 & & isStr; dupNil
\end{array}$$

where *strEq s* and *strNEq s* check whether the given string equals or not equals s. They are defined by:

$$\begin{array}{lcl} \mathit{strEq}\, s & = & \mathit{dupStr}\, s; \delta^{\smile} \\ \mathit{strNEq}\, s & = & \mathit{dupStr}\, s; \mathit{neq}; (\mathit{dupStr}\ s)^{\smile} \end{array}$$

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

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

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 $[(B \times [C])]$.

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

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:

$$\begin{array}{lcl} \lceil f \,\hat{;}\, g \rceil & = & \lceil f \rceil \rhd \lceil g \rceil \\ f \rhd g & = & f; (\mathit{id} \times \mathit{map}\, g; \mathit{pull}; (\mathit{map}\, g^{\smile} \times \mathit{id})); \\ & & \mathit{assocl}; (f^{\smile} \times \mathit{id}) \end{array}$$

We isolate the definition of $\rhd$ 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 the composition $\mathsf{children} \,\hat{;}\, \mathsf{children}$, given the input $t = \mathsf{N\,a}\,[\mathsf{N\,b}\,[\mathtt{c}, \mathtt{d}], \mathsf{N\,e}\,[\mathtt{f}, \mathtt{g}]]$. 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 $\mathsf{children} \,\hat{;}\, \mathsf{children}$ backwards.

According to the definition, we first apply $(\lceil \mathsf{children} \rceil \times \mathit{id}); \mathit{assocr}; (\mathit{map}\ \lceil \mathsf{children} \rceil \times \mathit{id})$ to $(t, [\mathtt{c}, \mathtt{d}, \mathtt{h}^{+}, \mathtt{f}, \mathtt{g}])$, yielding:

$$\begin{array}{l} (t, ([(\mathsf{N\,b}\,[\mathtt{c}, \mathtt{d}], [\mathtt{c}, \mathtt{d}]), (\mathsf{N\,e}\,[\mathtt{f}, \mathtt{g}], [\mathtt{f}, \mathtt{g}])], \\ \quad [\mathtt{c}, \mathtt{d}, \mathtt{h}^{+}, \mathtt{f}, \mathtt{g}])) \end{array}$$

So far we are simply reproducing the intermediate values that were generated in the forward run. Then we apply $\mathit{pull}^{\smile}$ to the second component of the pair. The list of lists $[[\mathtt{c}, \mathtt{d}], [\mathtt{f}, \mathtt{g}]]$ is compared against $[\mathtt{c}, \mathtt{d}, \mathtt{h}^{+}, \mathtt{f}, \mathtt{g}]$ in $\mathit{concatx}^{\smile}$, resulting in $[[\mathtt{c}, \mathtt{d}], [\mathtt{h}^{+}, \mathtt{f}, \mathtt{g}]]$. While performing $\mathit{map}\ \lceil \mathit{children} \rceil^{\smile}$, the pair $(\mathsf{N\,e}\,[\mathtt{f}, \mathtt{g}], [\mathtt{h}^{+}, \mathtt{f}, \mathtt{g}]])$ is unified into $\mathsf{N\,e}\,[\mathtt{h}^{+}, \mathtt{f}, \mathtt{g}]$, due to the use of *dup* in $\mathsf{children}$. The updated source is $\mathsf{N\,a}\,[\mathsf{N\,b}\,[\mathtt{c}, \mathtt{d}], \mathsf{N\,e}\,[\mathtt{h}^{+}, \mathtt{f}, \mathtt{g}]]$.

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 [16, 9] by hand. The updating behaviour in our embedding, on the other hand, follow naturally from the definition of composition and its extended semantics, although the situation is complicated by the fact that filters return a list of results.

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 $children \fatsemi txt\,\mathsf{a}$ and input $t' = \mathsf{N\,r}\,[\mathsf{a},\mathsf{a}]$, 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 distinguish between singleton and non-singleton filters, and perform a dynamic check to 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.

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

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 deconstructor $node^{\smile}$, and use the auxiliary function *cap* to place the result under the original root.

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

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

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

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.

$$\begin{array}{lcl}
\lceil f\ \mathsf{with}\ g \rceil & = & \lceil f \rceil \rhd dom\ \lceil g \rceil \\
\quad \textbf{where}\ \ dom\,g & = & g; (dupNil^{\smile}; dupNil \cup \\
 & & \quad (id \times cons^{\smile}; cons); dupfst; (g^{\smile} \times wrap)) \\
\lceil f\ \mathsf{without}\ g \rceil & = & \lceil f \rceil \rhd notdom\ \lceil g \rceil \\
\quad \textbf{where}\ \ notdom\,g & = & g; ((id \times cons^{\smile}; cons); dupNil \cup \\
 & & \quad dupNil^{\smile}; dupNil; dupfst; (id \times wrap))
\end{array}$$

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:

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

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

Figure 11. Definition of *norm*.

5.4 Get and Put

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

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

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 11. We start with a source document and uses get_x to produce an initial view. After each editing action, put_x is called (with a cached copy of the source) to produce an updated source. We then call 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 [18] is that the following properties hold:

$$
\begin{array}{rcl}
\mathit{untagged}; ⟦x⟧; ⟦x^{\smile}⟧; \mathit{norm} & = & \mathit{untagged}; \mathit{dom}\,⟦x⟧ \\
⟦x^{\smile}⟧; \mathit{norm}; ⟦x⟧; ⟦x^{\smile}⟧; \mathit{norm} & \subseteq & ⟦x^{\smile}⟧; \mathit{norm}
\end{array}
$$

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

Now assume that the user deletes b in the view $\mathsf{N}\,\mathtt{m}\,[a, b, a]$ and we perform a put_f. Firstly, $\lceil f \rceil^{\smile}\,(t, [\mathsf{N}\,\mathtt{m}\,[a, b^{-}, a]])$ results in $\mathsf{N}\,\mathtt{r}\,[b^{-}, a]$. 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 $\mathsf{N}\,\mathtt{r}\,[a]$.

Applying $\lceil f \rceil^{\smile}$ to $(t, [\mathsf{N}\,\mathtt{m}\,[a, {b_1}^{+}, b, a]])$ yields $\mathsf{N}\,\mathtt{r}\,[{b_1}^{+}, b, a]$. The newly inserted b_1, by the biased choice of *concatx*, is assumed to be the result of $\mathsf{children}$. It is also the same if we insert a_1 instead — $\lceil f \rceil^{\smile}\,(t, [\mathsf{N}\,\mathtt{m}\,[a, {a_1}^{+}, b, a]])$ yields $\mathsf{N}\,\mathtt{r}\,[{a_1}^{+}, b, a]$. The next get_f thus results in $\mathsf{N}\,\mathtt{m}\,[a_1, a, a_1, b, a]$ 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 $\mathsf{children} \mathbin{\hat{;}} \mathsf{tag}\,\mathtt{a}$. Indeed, $\lceil f \rceil^{\smile}\,(t, [\mathsf{N}\,\mathtt{m}\,[{a_1}^{+}, a, b, a]])$ yields $\mathsf{N}\,\mathtt{r}\,[b, {a_1}^{+}, a]$ because a_1 shall be inserted in front of a. Furthermore, elements inserted before the first a in the view must have an $\mathtt{a}$ label too, otherwise it could not have been the result of $\mathsf{tag}\,\mathtt{a}$.

If we want to allow the user to insert arbitrary elements, we need to use a different embedding of $\mathsf{tag}\,\mathtt{a}$ whose forward semantics is the same, but allows anything to go through in the backward direction. 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.

For a bigger example, recall the transformation in Figure 7. 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. There are more than one way to build the same table. For example, one can scan through the address book and replace every person with tr, the fields with td, by foldXml tabling where tabling is defined by:

```
tabling  =  tag person?⟩ replaceTag tr :⟩
             (tag name?⟩ repalceTag td :⟩
              (tag email?⟩ repalceTag td :⟩
               (tag tel?⟩ repalceTag td :⟩
                keep)))
```

However, this transformation does not enforce enough structure on the input. From the transformation we cannot infer how the source is supposed to look like. Its reverse, therefore, does not yield meaningful results.

6. Related Work

View-updating: to correctly reflect the modification on the view back to the database [3, 6, 8, 19, 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 [14]. In editors such as [22, 21] the user edits a view computed from the source by a transformation. Recent research on code clone [10] argues that a certain proportion of code in a software resembles each other. We may develop software maintenance tools to 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 [15] that such *coupled transformation* problems are widespread and diverse.

In the context of data synchronisation, similar challenge was identified by [9] and coined the "bidirectional updating" problem. In [9, 7], a semantic foundation and a programming language (the "lenses") for bidirectional transformations are given. They form the core of the data synchronisation system Harmony [20]. Another very much related language was given by Meertens [16] 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, 22, 21]. Proxima [21] 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 [16] and [9], is currently used in our XML editor [12] as the language to describe transformations with. In [12], the semantics of X was given without the use of Inv. In a preliminary work [11] in a non-refereed workshop, we drafted an implementation of bidirectional HaXML. In both cases, however, the treatments with 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 [13].

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.

A question is: what can we say about the updated 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 [16], 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 [21] 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. At present, we have an Inv interpreter implemented in Haskell, in which both X [12, 13] and HaXML is embedded. The prototype is available from the authors' homepage. 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. From these experience we wish to learn what language design/features are suitable for bidirectionalisation.

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

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

[3] F. Bancilhon and N. Spyratos. Update semantics of relational views. *ACM Transactions on Database Systems*, 6(4):557–575, December 1981.

[4] C. H. Bennett. Logical reversibility of computation. *IBM Journal of Research and Development*, 17(6):525–532, 1973.

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

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

[7] J. N. Foster, M. B. Greenwald, J. T. Moore, B. C. Pierce, and A. Schmitt. 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)*, pages 233–246, Long Beach, California, 2005. ACM Press.

[8] G. Gottlob, P. Paolini, and R. Zicari. Properties and update semantics of consistent views. *ACM Transactions on Database Systems*, 13(4):486–524, December 1988.

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

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

[11] Z. Hu, K. Emoto, S.-C. Mu, and M. Takeichi. Bidirectionalizing tree tranformations. In *Workshop on New Approaches to Software Construction (WNASC 2004)*, Komaba, Tokyo, Japan, September 13-14, 2004.

[12] Z. Hu, S.-C. Mu, and M. Takeichi. 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, August 2004. ACM Press.

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

[14] D. Janzen and K. de Volder. Programming with crosscutting effective views. In *ECOOP 2004 - Object-Oriented Programming, 18th European Conference*, number 3086 in Lecture Notes in Computer Science, pages 195–218. Springer-Verlag, June 14-18, 2004.

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

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

[17] S.-C. Mu, Z. Hu, and M. Takeichi. An injective language for reversible computation. In *Seventh International Conference on Mathematics of Program Construction*, number 3125 in Lecture Notes in Computer Science. Springer-Verlag, July 2004.

[18] S.-C. Mu, Z. Hu, and M. Takeichi. An algebraic approach to bi-directional updating. In W.-N. Chin, editor, *The Second Asian Symposium on Programming Language and Systems*, number 3302 in Lecture Notes in Computer Science, pages 2–20. Springer-Verlag, November 4-6, 2004.

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

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

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

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

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