MATHEMATICAL ENGINEERING
TECHNICAL REPORTS

Configuring Bidirectional Programs with Functions

Masato TAKEICHI

METR 2009–34 August 2009

DEPARTMENT OF MATHEMATICAL INFORMATICS
GRADUATE SCHOOL OF INFORMATION SCIENCE AND TECHNOLOGY
THE UNIVERSITY OF TOKYO
BUNKYO-KU, TOKYO 113-8656, JAPAN

WWW page: http://www.keisu.t.u-tokyo.ac.jp/research/techrep/index.html
The METR technical reports are published as a means to ensure timely dissemination of scholarly and technical work on a non-commercial basis. Copyright and all rights therein are maintained by the authors or by other copyright holders, notwithstanding that they have offered their works here electronically. It is understood that all persons copying this information will adhere to the terms and constraints invoked by each author’s copyright. These works may not be reposted without the explicit permission of the copyright holder.
Abstract

Many issues both in view-updating and in synchronization have been dealt with successfully by designing and implementing domain-specific languages with bidirectional semantics. These still lack, however, encapsulation mechanisms with which the forward and backward transformations relate to each other.

We propose in this paper a concept of bidirectional programming for problems like view-updating. In bidirectional programming, forward and backward transformations are encapsulated in a single bidirectional function which is evaluated twice in a program, once in forward and once in backward direction. The framework of bidirectional programming brings about a new approach to view-updating and synchronization problems and an efficient and reliable implementation of bidirectional transformation.

We illustrate the idea of bidirectional programming by bidirectionalizing a domain-specific library HaXml for XML processing and demonstrate the usefulness of bidirectional functions through the development of a view-updating system.

1 Introduction

We often encounter the task of maintaining persistent source data through a user’s view consisting of partial data extracted from the source. As well as replacement of data items in the source, insertion of new data items and deletion of existent entries are considered typical maintenance operations through the view. Such kind of task has been intensively studied in the database community, and is called view-updating problem.

Recently, the problem of maintaining the consistency of two pieces of structured data was brought to our attention. Synchronization of bookmarks of Web browsers is an example. In this context, the source and the view comprise a collateral pair of structured data to be kept consistent.

Though developed separately, their results turn out to be similar: the source data and the view are transformed to and from each other by a pair of functions with keeping some consistent properties. The forward and backward transformations comprise a bidirectional transformation between the source and the view. Many issues both in view-updating and in synchronization have been dealt with successfully by designing and implementing domain-specific languages with bidirectional semantics.

These still lack, however, encapsulation mechanisms with which the forward and backward transformations relate to each other. These transformations discussed so far are considered simply as components of a pair \( \langle f^>, f^< \rangle \) of a forward \( f^> \) and a backward \( f^< \) function. It is common in bidirectional transformation that the backward transformation from the modified view to the source requires the original source data as well as...
the modified view because the view is an excerpt of the source and may not convey the complete information on the source. Hence the source data might be possibly scanned again during the backward transformation. This is the reason why we should be asked to solve the inefficiency of bidirectional transformations.

We propose in this paper a concept of bidirectional programming for problems like view-updating. A bidirectional program behaves like bidirectional transformation except that it takes editing operation on the view into account as well as transformations. In a few words, the bidirectional program takes a source data as input, transforms it to produce a view on which operations are performed, and then it transforms the modified view back to put a modified source data as output. In such bidirectional programs, forward and backward transformations are encapsulated in a single bidirectional function. The bidirectional function is a higher order function which takes a function over the target view as its argument and produces a function over the source data domain.

We first consider the framework of bidirectional programming, and then propose an idea of bidirectional function with concrete implementation in Haskell. And we exemplify the advantage of our idea for bidirectional transformation through bidirectionalizing the HaXml library, and finally demonstrate a bidirectional XML viewer developed for illustration.

2 Bidirectional programming

Suppose we are about to write a program which takes a source data as input and produces a target view through which updating operations for the source data are performed. In order to reflect the changes on the view back to the source, we need another program to be performed after updating. These programs, one transforms in \textit{forward} and the other in \textit{backward} direction, comprise a \textit{bidirectional transformation}.

If we consider the whole process as a single bidirectional program, updating operations on the view should take part in the program itself as shown in Figure 1. The operations on the view may be performed by the user of the view-updating system who takes part in the whole process of the program.

![Figure 1: Bidirectional program](image)

2.1 Bidirectional programs with functions

Consider how to construct the bidirectional program with functions. In the course of evaluation of bidirectional programs, the update operation might be taken for some target view \( t \in \text{Tar} \) of the source data \( s \in \text{Src} \) rather than for the source itself, and modification on the view \( t' \in \text{Tar} \) should be brought back to the source to produce new source data \( s' \in \text{Src} \).

The \textit{forward} function \( f^\succ \) from the source to the target view and the \textit{backward} function \( f^\prec \) from the view to the source behave as if they were inverse of the other. It is, however, unrealistic to impose such bijective properties on these functions. In most programs of this kind the forward function \( f^\succ \) produces the view \( t = (f^\succ s) \) consisting only of interested excerpted information from the source \( s \). Although this makes the user manipulate the data easier than do on the huge source data, we could not assume that the backward function \( f^\prec \) brings the modification back to the source solely with the modified view \( t' \). We should put the assumption that the backward function takes the original source \( s \) as well as the modified view \( t' \)
to produce the result \( s' = (f^<_s t') \).

Thus, the part of bidirectional transformation of the bidirectional program is represented by a pair of functions \( (f^>, f^<_s) \) of which component functions have the functionality

\[
\begin{align*}
  f^> & : \text{Src} \rightarrow \text{Tar} \\
  f^<_s & : \text{Src} \rightarrow \text{Tar} \rightarrow \text{Src}
\end{align*}
\]

with properties:

- the forward function \( f^> \) transforms the source data \( s \) into its target view \( t \):
  \( t = f^>s \),
  \[
  t = f^>s,
  \]
  and

- the backward function \( f^<_s \) takes the original source data \( s \) and the modified view \( t' \) to bring the change of the target view back to produce the modified source \( s' \):
  \( s' = f^<_s t' \)
  \[
  s' = f^<_s t',
  \]

as shown in Figure 2.

![Figure 2: Bidirectional transformation](image)

As for the operation on the view, it is reasonable to represent it by a function \( h \):
\[
  t' = h t
  \]

which is also shown in Figure 2.

Of course, each transformation should satisfy certain property of **bidirectionality**. Here, we should notice that the transformation \( (f^>, f^<_s) \) along with \( h \) implements a bidirectional program that we expect from the specification, and therefore the property is to be stated in terms of relations between \( f^> \), \( f^<_s \) and \( h \).

### 2.2 Bidirectionality

Several definitions of the property of **bidirectionality** have been proposed for characterizing the bidirectional transformation under consideration. In [16, 9], the bidirectional property based on the “Get-Put” and “Put-Get” conditions is stated as a fundamental one. Here, “Get” and “Put” correspond to our \( f^> \) and \( f^<_s \), and the Put-Get condition states that \( f^> (f^<_s t) = t \) for some \( s \in \text{Src} \) and any \( t \in \text{Tar} \).

However, if we allow duplication in the view described in Section 2.3 below, the Put-Get property is too restrictive and is not always satisfied. Hence the bidirectionality is defined differently in [12] which is based on the “Get-Put-Get” and “Put-Get-Put” conditions.

As a matter of fact, the property of this kind cannot convey the whole properties of our interest. They do not refer to the operation on the view, but only assume that some modification may be performed to the view. From the standpoint of our bidirectional programming, the bidirectionality would be a consequence of the program composed of a bidirectional transformation \( (f^>, f^<_s) \) and an operation function \( h \). Hence, we do not specify here the property of bidirectionality in general, but do **define the bidirectionality by programming** primitives for bidirectional programs. We might believe that our well-written bidirectional program will in fact behave as they satisfy certain property which may be recognized as bidirectional.

In this paper, we will introduce a set of such basic bidirectional transformations in Sections 3.1 and 4.2, where we call them **filters** from the source data to the target view. The set of the basics constitutes itself as a domain-specific language with well-defined semantics. The user can understand the meaning of each transformation and knows what happens by using it for programming to solve problems. Programs written in this language enjoy the bidirectional property based on those of their components.

We state here the simplest property of **stability** as the minimum requirement of
Stability:
For any \( s \in \text{Src} \),
\[
f^< s \left( f^> s \right) = s
\]
holds.

The stability condition says that the source \( s \) remains unchanged if no modification is made in the target view as shown in Figure 3. This is the most fundamental condition that any bidirectional transformation \( (f^>, f^<) \) must satisfy. It is same as the Get-Put condition described above. Note that the condition does not refer to \( h \), or does state that it holds for the case \( h = \text{id} \), the identity function. Hence, the stability condition is considered as the weakest condition of our bidirectional programs. We need more to specify with respect to \( h \).

2.3 Duplication in bidirectional transformation
So far we have looked into bidirectional transformation by representing it with functions. We can compose functions to build larger functions as usual in functional programming. This is nothing special to bidirectional program construction compared to functional programming.

There is, however, a very useful construction mechanism called duplication proposed for bidirectional transformation [17, 18, 11, 19, 12]. This is beyond the ordinary functional composition and should be worth mentioning here.

Duplication of some source data may appear more than twice in the target view as if their occurrences were instances of the copy of a single item. These look like copies at least by their appearances in the view. But in our bidirectional settings, duplication differs from copy in that if one of the occurrences in the view is modified, the corresponding source data and the other occurrences in the view should change accordingly. In this way, duplication makes
its occurrences kept consistent if one of them is modified.

Let us consider duplication by splitting the source to produce the view composed by two bidirectional transformation \( (f^<, f^>) \) and \( (g^>, g^<) \) as shown in Figure 4. Both transformations share the source \( s \) and they produce \( t_f = (f^> s) \) and \( t_g = (g^> s) \) as parts of the view \( t = (t_f, t_g) \). Suppose that some updating operation is taken on the part \( t_f \) yielding \( t'_f \), and \( t_g \) remains unchanged; thus, the total view \( t \) is changed to \( t' = (t'_f, t_g) \). Then, we first bring the change to the source by \( f^< \) to produce the modified source \( s' = (f^< s t'_f) \) by the backward function \( f^< \). Although this makes \( s' \) updated in accordance with the change in \( t'_f \), this change has not yet propagated to the \( t_g \) part of the view \( t' \). In order to keep the view consistent, we need to perform the forward transformation again to get the view \( t'' = (f^> s', g^> s') \). Prosessing this way keeps the duplicated instances consistent in the whole view.

Here, we have the bidirectional property for this construction.

**Bidirectionality of Duplication-by-Split**

For a bidirectional transformation \( (k^<, k^>) \) which is constructed by splitting the source \( s \) with \( (f^<, f^>) \) and \( (g^>, g^<) \), and for any \( h \),

\[
\begin{align*}
    k^< s t' &= f^< s t'_f \quad \text{if } t'_f = (t'_f, g^> s) \\
    &= g^> s t'_g \quad \text{if } t'_g = (f^> s, t''_g) \\
    \text{where } (f^> s, g^> s) &= k^> s \\
    t' &= h t
\end{align*}
\]

holds.

As the duplication-by-split is widely used in transformation for XML processing, we should take it into account to deal with bidirectional transformation. It would be well understood that this mechanism could be used in synchronizing \( t_f \) and \( t_g \) by bidirectional transformation as well.

### 3 Functions for bidirectional programming

We have examined bidirectional transformation in general with its functional representation. But when we consider implementation of bidirectional programs with functions, we should take full care about the close connection of forward \( f^> \) and backward \( f^< \) functions of bidirectional transformation \( (f^>, f^<) \).

A simple way of implementing the bidirectional transformation \( (f^>, f^<) \) would be to define \( f^> \) and \( f^< \) as \( f_f \) and \( f_b \), respectively.

\[
\begin{align*}
    f_f &:: \text{Src} \rightarrow \text{Tar} \\
    f_b &:: \text{Src} \rightarrow \text{Tar} \rightarrow \text{Src}
\end{align*}
\]

A bidirectional transformation \( \text{xmin} \) for the bidirectional program for presenting the minimum looks like:

\[
\begin{align*}
    \text{xmin} :: \text{Ord a} \Rightarrow ((a,a) \rightarrow a, a \rightarrow (a,a)) \\
    \text{xmin} &= (f_f, f_b) \text{ where} \\
    f_f (x,y) &= \text{if } x<y \text{ then } x \text{ else } y \\
    f_b (x,y) t' &= \text{if } x<y \text{ then } (t',y) \text{ else } (x,t')
\end{align*}
\]

We can see that this satisfies the bidirectional property of \( \text{xmin} \) mentioned in Section 2.2. What we observe from this example is: why do we need to evaluate the expression if \( x<y \) again in backward function? We know which component should be replaced from the forward transformation for the given source data.

To solve this problem, we will try to parametrize the backward function with the source. The key idea is this: if a backward function \( f^< \) is generated for each instance of the source data \( s \) given to the forward function \( f^> \), no more evaluation of \( s \) is required in the backward transformation. This leads to our idea of bidirectional functions.
3.1 Bidirectional functions

We propose here a novel idea of encapsulating forward and backward functions in a function called $XF\text{un}$, which is a higher order function taking a function to give a function. Using the $XF\text{un}$, bidirectional transformation $(f^+, f^−)$ is expressed as a single function $XF\text{un} f$.

**Definition (Bidirectional function)**

A bidirectional function $XF\text{un} f :: XF\text{un} a b$

is a function which takes a function $h :: b \rightarrow b$ as argument and returns a function of the type $a \rightarrow a$ as its result.

The basic idea of $XF\text{un}$ is to keep the result obtained in the course of evaluation of the forward function in itself for the backward function to use it later. That is, the forward function $f$ of $XF\text{un} f$ applied to the source $s$ gives a pair consisting of the view $t$ and the backward function $f^0$ which will be used later when the backward transformation is performed.

Hence, the function $f$ of $XF\text{un} f$ works as $f^+$ of the bidirectional transformation $(f^+, f^-)$, and the function $f^0$ does for $(f^-, s)$ which is the function $f^-$ partially parametrized with the source $s$. Much the same way as the function $(f^-, s)$ need to take only the modified target $t'$ to bring the change back to the source, $f^0$ takes $t'$ only for the backward transformation.

We can define $XF\text{un}$ in Haskell as:

```
newtype XFun a b = XFun (a -> (b,b -> a)).
```

And we may give a definition of bidirectional function $xmin$:

```
xmin :: Ord a => XFun (a,a) a
xmin = XFun f where
  f (x,y) | x<y = let f' t' = (t', y) in (x,f')
  | otherwise = let f' t' = (x, t') in (y,f')
```

1 Although a bidirectional function might be defined simply by an ordinary function, introduction of a functional datatype $XF\text{un}$ would make us concerned with this specific kind of higher order functions.

Here, we can easily observe that no redundant evaluation of the source occurs in backward transformation. It should be noted, however, that the bidirectional function $XF\text{un}$ produces a functional closure as its intermediate result and may consume space proportional to the number of occurrences of $XF\text{un}s$ in the course of evaluation of forward functions.

**Construction of bidirectional programs**

A bidirectional program is constructed from a bidirectional function $XF\text{un} f :: XF\text{un} a b$

and a function $h :: b \rightarrow b$

by the combinator $(\cdot)$. The expression $(XF\text{un} f (\cdot)) h$ is a function over the source data and behaves as a function of the type $(a \rightarrow a)$ as shown in Figure 5.

![Figure 5: Construction with (\cdot)](image)

The construction combinator $(\cdot)$ is implemented in Haskell as:

```
\([\cdot]:: XFun a b \rightarrow (b \rightarrow b) \rightarrow a \rightarrow a\)
\([XF\text{un} f \cdot] h≡\ \let\ t f s\ = f' (h t)
```

**Composition of bidirectional functions**

Bidirectional functions are combined together by the combinator $(\cdot)$. The expression $(XF\text{un} f (\cdot) XF\text{un} g)$ is a bidirectional function composed of $XF\text{un} f$ and $XF\text{un} g$ in this order, that is, first apply $f$ to the source and then apply $g$ to that result in forward transformation. Of course, the order of backward application is the reverse of the forward (Figure 6).
Bidirectional duplication by split

As we have seen in Section 2.3, our bidirectional programming allows construction for duplication. More concretely, we provide a construction mechanism which combines two bidirectional functions to generate a new bidirectional function; the bidirectional split construction \((\text{XFun } f \hat{\times} \text{XFun } g)\) accepts a single source for \(f\) and \(g\) to produce the view respectively and combine the results into a pair in its forward transformation. The backward transformation reflect the changed result if any as the result of the split (Figure 7).

Our bidirectional duplication can be implemented by the split construction.

\[
\begin{align*}
((\hat{\times})) :& \text{XFun } a \ b \rightarrow \text{XFun } b \ c \rightarrow \text{XFun } a \ c \\
(\text{XFun } f \hat{\times} \text{XFun } g) &= \text{XFun } k \text{ where} \\
k \ &= \ (z, \ f' \ g') \\
(z, \ g') &= \ y \\
(y, \ f') &= \ f \ x \\
\end{align*}
\]

This satisfies the bidirectional property of duplication-by-split mentioned in Section 2.3. Note that we assume here that both of the pair are never modified at a time. This is the meaning of our duplication, while some conditions about disjointness of the domains for \(tf\) and \(tg\) may loosen this condition.

### 3.2 Bidirectional functions on lists

If we are provided bidirectional functions for lists in the standard Haskell library, we can write bidirectional programs on lists for free. That is, we can get bidirectionality simply by writing programs which transform the source in forward direction; we do not need to write functions for backward transformation.

Some examples of defining bidirectional functions follow; \(\text{xyyy}\) corresponds to the Haskell standard function \(\text{yyy}\).

\[
\begin{align*}
\text{xfoldr} :& \text{XFun } (a,b) \ b \rightarrow \text{XFun } [a] \ b \\
\text{xfoldr} &\text{ (XFun } f \text{)} \ e = \text{XFun } k \text{ where} \\
k \ &= \ (y, \ f' \ fs') \\
(y, \ fs') &= \ \text{foldr } g \ (e,[]) \ xs \\
g \times (z,fs')&= \\
\text{let } (z',f')=f(x,z) \text{ in } (z',f':fs') \\
k' \ [ ] \ y' &= [ ] \\
k' \ (f':fs') \ y' &= x':k' \ fs' \ z' \\
\text{where } (x',z') &= f' \ y' \\
\end{align*}
\]

\[
\begin{align*}
\text{xmap} :& \text{XFun } a \ b \rightarrow \text{XFun } [a] \ [b] \\
\text{xmap} \ (\text{XFun } f) &= \text{XFun } k \text{ where} \\
k \ &= \ (ys, \ k') \\
(ys, \ fs') &= \ \text{unzip } (\text{map } f \ xs) \\
k' \ ys'=\text{zipWith } (\text{f' y' → f' y'}) \ fs' \ ys' \\
\end{align*}
\]

\[
\begin{align*}
\text{xfilter} :& \ (a \rightarrow \text{Bool}) \rightarrow \text{XFun } [a] \ [a] \\
\text{xfilter} \ p &= \text{XFun } k \text{ where} \\
k \ [ ] &= ([], \ \text{id}) \\
k \ (c:xs) &= \text{where} \\
(y, \ g') &= k \ xs \\
k' \ (y'\!::ys') &= y' : g' \ ys' \\
k'' \ ys' &= x : g' \ ys' \\
\text{xconcat} :& \text{XFun } [[a]] \ [a] \\
\text{xconcat} &= \text{XFun } k \text{ where} \\
k \ xs &= \ (\text{concat } xs) \\
\text{where} \\
\text{ls} &= \ \text{map length } \text{xs} \\
\end{align*}
\]
We can observe that these definitions make full use of intermediate results of forward transformation for the backward. It should be noted that we assume the length of the list produced for the target view remains as it was before modification by editing operations.

In addition to the standard Haskell library functions, we can write more complex but useful functions such as sorting.

\[
x_{\text{sort}} :: \text{Ord } a \Rightarrow \text{XFun } [a] \to [a]
\]

\[
x_{\text{sort}} = x_{\text{foldr}} \text{xinsert } []
\]

\[
x_{\text{insert}} :: \text{Ord } a \Rightarrow \text{XFun } (a,[a]) \to [a]
\]

\[
x_{\text{insert}} = \text{XFun } k \text{ where}
\]

\[
k (x, []) = ([x], k')
\]

\[
\text{where } k' [y] = (y, [])
\]

\[
k (x, x:xs@(_:xs')) | x \leq x' = (x:xs, k')
\]

\[
\text{otherwise } = (x':ys, k'')
\]

\[
\text{where}
\]

\[
k' (y':ys') = (y', ys')
\]

\[
(ys, g') = k (x, xs')
\]

\[
k'' (y':ys') = y':g' (ys')
\]

\[
\text{Given these bidirectional functions, we can demonstrate bidirectional programs such as swapping the first two smallest elements of the list as:}
\]

\[
x_{\text{fst}} :: \text{Ord } a \Rightarrow [a] -> [a]
\]

\[
x_{\text{fst}} = x_{\text{sort}} \varnothing \text{ swapfsts}
\]

\[
\text{where}
\]

\[
\text{swapfsts } (x:y:zs) = y:x:zs
\]

An example session:

\[
* \text{Main} > x_{\text{fst}} [2,5,1,4]
[1,5,2,4]
* \text{Main} > x_{\text{fst}} [3,4,1,2]
[3,4,2,1]
\]

Another useful function is \(x_{\text{remdups}}\) which removes adjacent duplicate elements.

\[
x_{\text{remdups}} :: \text{Ord } a \Rightarrow \text{XFun } [a] \to [a]
\]

\[
x_{\text{remdups}} = \text{XFun } k \text{ where}
\]

\[
k [] = ([], \text{id})
\]

\[
k [x] = ([x], \text{id})
\]

\[
k (x:xs@(x':xs')) | x == x' = (ys, k')
\]

\[
\text{where}
\]

\[
k' (ys') = y':g' (ys')
\]

\[
\text{An example session:
* \text{Main} > x_{\text{remdups}} \varnothing [3,1,2,3,4,1,1]
[3,0,2,3,4,0,0]
* \text{Main} > x_{\text{remdups}} 4 [3,1,2,3,4,1,1]
[3,4,2,3,4,4,4]
\]

Note that corresponding Haskell functions \(\text{sort}\) and \(\text{remdups}\) give

\[
* \text{Main} > (\text{remdups} . \text{sort}) [3,1,2,3,4,1,1]
[1,2,3,4]
* \text{Main} > (\text{swapfst} 0 . \text{remdups} . \text{sort}) [3,1,2,3,4,1,1]
[0,2,3,4]
\]

And the forward transformation of the bidirectional program \(x_{\text{remdups}} 0\) behaves like this, while the backward transformation of \(x_{\text{remdups}} 0\) brings the first 0 to the locations where the element 1 appears.

## 4 Bidirectionalizing HaXml

In this section we will briefly review the core of the combinator library of HaXml [26], followed by bidirectionalizing HaXml by bidirectional functions.

The bidirectional HaXml called \(\text{BiHaXml}\) brings about great productivity and reliability for bidirectional programming. That is, we need only to write forward functions for intended applications and we get corresponding backward functions for free.

To this purpose, the BiHaXml library respects HaXml’s datatypes and names as much as possible.

HaXml deals with internal representation of XML by the datatype Content:
Figure 8: An example of XML document

<authorlist>
  <author>
    <name>Sachiko Kizu</name>
    <email>sachiko@ipl</email>
  </author>
  <author>
    <name>Masato Takeichi</name>
    <email>takeichi@ipl</email>
  </author>
  <author>
    <name>Zhenjiang Hu</name>
    <email>hu@nii</email>
  </author>
</authorlist>

Figure 9: An example of Content

data Content = CElem Element
| CString Bool String
| CRef Reference
| CMisc Misc

data Element =
  Elem String [Attribute] [Content]
type Attribute = (String, AttValue)
data AttValue =
  AttValue [Either String Reference]
data Reference = ... (omitted)

An XML tree is represented by a Content tree, which is either a CElem or a CString. A CElem node has its Element consisting of the XML tag of type String, an optional Attribute list, and an optional children list of type Content. An Attribute is represented by a pair of a String key with associated value of type AttValue, which may be either a String or a Reference. A CString is a leaf of the Content tree. We omit here the Reference for simplicity. Attributes will be internally extended in Section 5 to record user editing.

Figure 9 rewrites an XML document of Figure 8 in the HaXml representation.

4.1 HaXml combinators

Combinators in HaXml are called filters. Filters have type

type CFilter = Content → [Content]

taking a Content and returning a possibly empty sequence of Content.

Basic Filters

A set of basic filters in HaXML is given in Figure 10. The simplest filters are keep and none; keep takes any Content tree and returns just that tree, and none fails on any input (returning an empty list).

keep, none :: CFilter
keep x = [x]
none x = []

The filter elm returns just this item if it is a CElem element, otherwise it fails. Conversely, txt returns this item only if the item is a CString or a CRef, that is, not a CElem element.

elm, txt :: CFilter
elm x@(CElem (Elem n _)) = [x]
elm _ = []
txt x@(CRef _ ) = [x]
txt _ = []

The filter (tag t) returns the input only if it is a CElem which has the tag name t.

tag :: String → CFilter
tag t x@(CElem(Elem n _)) | t==n = [x]
tag t _ = []

Content Constructors

The filter (literal s) always returns a CString of s. The construction combinator (mkElem n cfs) builds a CElem element with the tag n; the argument cfs
is a list of filters, each of which is applied to the current item. The results are concatenated by the \texttt{cat} combinator described below, and become the children of the created element. \texttt{(replaceTag \textit{n})} changes the node label \textit{n} if the input is a node, and returns empty list otherwise.

\begin{verbatim}
literal :: String -> CFilter
literal s = const [CString False s]

mkElem :: String -> [CFilter] -> CFilter
mkElem n cfs t = [CElem(Elem n [ ] (cat cfs t))]

replaceTag :: String -> CFilter
replaceTag n (CElem(Elem cs)) = cs
replaceTag n = []
\end{verbatim}

The filters so far return either a singleton list of \texttt{Content} or an empty list. An empty list is sometimes used to represent “failure” in filter application or the \texttt{False} value in predicates.

Other filters do not have constraints on the length of the output.

\section*{Content Selector}

The filter \texttt{children} returns the immediate children of the tree, if any.

\begin{verbatim}
children :: CFilter
children (CElem (Elem _ cs)) = cs
children_ = []
\end{verbatim}

\section*{Filter Combinators}

Figure 10 also lists basic combinators to compose \texttt{CFilter}s out of simpler ones. The sequential composition \texttt{(f \text{"o"} g)} applies \texttt{g} to the input, before applying \texttt{f} to each of the output and concatenating the results. For example, \texttt{(children \text{"o"} tag \text{s})} returns all the children immediately enclosed by the input, provided that the input is a \texttt{CElem} element with the tag \texttt{s}.

\begin{verbatim}
o :: CFilter -> CFilter -> CFilter
f \text{"o"} g = concatMap f \text{"o"} g
\end{verbatim}

The expression \texttt{(concatMap \textit{f})} first maps the filter \textit{f} to each element of type \texttt{Content} of the given list, and then concatenates the result of type \texttt{[[Content]]} to get a list of \texttt{Content}. Note that the \texttt{CFilter} \textit{g} produces a value of \texttt{[Content]} from one of \texttt{Content}.

\begin{verbatim}
(\textit{f \text{"union"}} \textit{g}) concatenates the splitted results of filters \textit{f} and \textit{g}, while \texttt{(cat \textit{f}s)} is its generalization to a list of filters. In fact, \texttt{cat} is defined as \texttt{cat=}foldr1 union using a standard Haskell function foldr1 in the HaXml library [26].

The combinator expression \texttt{(f \text{"with"} g)} acts as a guard on the results of \texttt{f}, keeping only those that are productive (yielding non-empty results) under \texttt{g}. Its dual, \texttt{(f \text{"without"} g)}, excludes those results of \texttt{f} that are productive under \texttt{g}.

The expression \texttt{(f \text{"et"} g)} applies \texttt{f} to the input if it is a \texttt{CString}, and applies \texttt{g} otherwise. The expression \texttt{(p?\text{"f";}\text{"g"})} represents conditional branches; if the (predicate) filter \texttt{p} is productive given the input, the filter \texttt{f} is applied to the input, otherwise \texttt{g} is applied.

The expression \texttt{(chip \textit{f})} applies \texttt{f} to the immediate children of the input. The results are concatenated as new children of the \texttt{CElem} element.

\section*{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 \texttt{(/>) and \text{"/"}.}

\begin{verbatim}
f /g = g \text{"o"} children \text{"o"} f
f g = f \text{"with"} (g \text{"o"} children)
\end{verbatim}

Both of them apply \texttt{f} to the input and prune away those subelements of the result that does not make \texttt{g} productive (i.e., \texttt{g} does not fail); \texttt{(/>)} can be seen as selecting a subtree given a path. It is an ‘interior’ selector, returning the inner structure, while \texttt{(/)} is an ‘exterior’ selector, returning the outer structure.

Another class of useful filter combinators allows one to process trees recursively. The combinator \texttt{deep} defined by

\begin{verbatim}
deep f = f ?>(f :>(deep f \text{"o"} children))
\end{verbatim}

potentially pushes the action of filter \texttt{f} deep inside the document subelement. It first tries the given filter on the current \texttt{Content}: if the filter is productive then
it stops, otherwise it moves to the children recursively. Another powerful recursion combinator is foldXml; the expression \( \text{foldXml } f \) applies the filter \( f \) to every level of the Content tree, from the leaves upwards to the root.

\[
\text{foldXm } f = f \ 'o' \ \text{chip} \ (\text{foldXml } f)
\]

### 4.2 BiHaXml combinators

Our bidirectional combinators have type

\[
\text{type XFilter} = \text{XFun Content [Content]}.
\]

The bidirectional filter \( \text{XFun } f \) of BiHaXml applied to Content data \( s \) produces the result \( (t, f') \) where \( t \) is the result of forward transformation of \( f \), and \( f' \) is the corresponding backward transformation.

#### Basic Filters

The basic combinators of HaXml are bidirectionalized with XFuns to define new XFilters.

\[
\begin{align*}
\text{keep, none :: XFilter} & \\
\text{keep} & = \text{XFun } k \text{ where } \ k \ x = ([x], \ \text{head}) \\
\text{none} & = \text{XFun } k \text{ where } \ k \ x = ([], \ \text{const } x)
\end{align*}
\]

The filters \( \text{elm} \) and \( \text{txt} \) are easily made bidirectional with \( \text{XFun} \) as follows.

\[
\begin{align*}
\text{elm} & = \text{XFun } k \text{ where } \\
& \ k \ x@\text{(CElem n)} = ([x], \ \text{head}) \\
& \ k \ x = ([], \ \text{const } x)
\end{align*}
\]

The bidirectional \( (\text{tag } t) \) returns the result of the forward transformation. According to its cases, the rules appear in the above combinators apply; it returns \( ([x], \ \text{head}) \) if the argument is acceptable, and does \( ([], \ \text{const } x) \) otherwise.

\[
\begin{align*}
\text{tag } t & = \text{XFun } k \text{ where } \\
& \ k \ x@\text{(CElem (Elem n _ _))} \\
& \qquad | \ t == n = ([x], \ \text{head}) \\
& \qquad | \ otherwise = ([], \ \text{const } x) \\
& \ k \ x = ([], \ \text{const } x)
\end{align*}
\]

#### Content Constructors

The filter \( (\text{literal } s) \) always returns a leaf labeled \( s \) with the backward function which returns the original regardless of
the change of the view. The construction combinator \( \text{mkElem } n \text{ cf}s \) builds a tree with the node label \( n \); the argument \( \text{cf}s \) is a list of filters, each of which produces its forward result and its backward transformation. The forward transformation by \( \text{XFun} \) is captured by the \( \text{cat} \) combinator of \( \text{XFilter} \) described below. The forward results are concatenated and become the children of the created element, with the backward function produced by \( \text{cat} \) is kept in this \( \text{XFun} \) for the future use of the backward transformation of \( \text{mkElem} \). The backward transformation of \( \text{replaceTag } n \) is simple.

\[
\begin{align*}
\text{literal} & :: \text{String} \to \text{XFilter} \\
\text{literal } s & = \text{XFun } k \text{ where } \\
k x & = ((\text{CString } \text{False } s), \const x)
\end{align*}
\]

\[
\begin{align*}
\text{mkElem} & :: \text{String} \to [\text{XFilter}] \to \text{XFilter} \\
\text{mkElem } n \text{ cf}s & = \text{XFun } k \text{ where } \\
k s & = ([\text{CElem } (\text{Elem } n \text{ [ ] } t \text{ ) }], k') \\
& \text{ where } \\
\text{XFun } g & = \text{cat cf}s \\
(t, g') & = g s \\
k' [\text{CElem } (\text{Elem } n \text{ , } t')] & = g' t'
\end{align*}
\]

\[
\begin{align*}
\text{replaceTag} & :: \text{String} \to \text{XFilter} \\
\text{replaceTag } n & = \text{XFun } k \text{ where } \\
k (\text{CElem } (\text{Elem } m \text{ as } cs)) & = (\text{CElem } (\text{Elem } n \text{ as } cs)), k') \\
& \text{ where } \\
k' [\text{CElem } (\text{Elem } \text{ , as } cs)] & = \\
\text{CElem } (\text{Elem } m \text{ as } cs') \\
k s & = ([], \const s)
\end{align*}
\]

**Content Selector**

The bidirectional filter \( \text{children} \) returns the immediate children in forward direction and returns them in place in the following backward transformation.

\[
\begin{align*}
\text{children} & :: \text{XFilter} \\
\text{children} & = \text{XFun } k \text{ where } \\
k (\text{CElem } (\text{Elem } n \text{ as } cs)) & = (cs, k') \\
& \text{ where } \\
k' cs' & = \text{CElem } (\text{Elem } n \text{ as } cs') \\
\text{children } s & = ([], \const s)
\end{align*}
\]

**Filter Combinators**

Bidirectional combinators of \( \text{XFilter} \) make full use of the characteristic property of \( \text{XFun} \) in that the backward function is instantiated by partial parametrization with the input of the forward function.

The bidirectional sequential composition \( \text{f o g} \) can be specified with auxiliary bidirectional functions \( \text{(\_)}, \text{xmap}, \text{and } \text{xconcat} \) described in Sections 3.1 and 3.2. These functions correspond to Haskell functions \( \text{(.)}, \text{map}, \text{and } \text{concat} \) used in the definition of the HaXml combination \( \text{(f o g)}=(\text{concatMap f . g}), \text{or } (\text{concat . map f . g}). \)

\[
\begin{align*}
o :: \text{XFilter} \to \text{XFilter} \to \text{XFilter} \\
f' o' g & = g (\text{\_}) \text{xmap f (\_)} \text{xconcat}
\end{align*}
\]

Note that application of the \( \text{XFilter} \) composition \( \text{(\_)} \) is performed from left to right while \( \text{CFilter} \) composition \( o \) from right to left.

The forward transformation of the combinator union gives a concatenated list of the results of its operand filters. The filter \( \text{(XFun f \text{ union } XFun g)} \) takes source data \( s \) which is split and shared by the two filters \( \text{XFun f} \) and \( \text{XFun g} \). The two filters produce their results \( (t_f, f') \) and \( (t_g, g') \) independently. Although the duplication transformation described in Section 2.3 produces a pair of the results of two functions, the expression \( \text{(XFun f \text{ union } XFun g)} \) combines the results \( t_f \) and \( t_g \) into a single list by concatenation. Hence, when dealing with backward transformation, we need to identify which part of the target \( \text{Content} \) has been modified. Of course, we were to compute \( t_f \) and \( t_g \) again from the source, but it will cost too much and cause inefficiency in backward transformation. We therefore keep information of the intermediate results \( t_f \) and \( t_g \) inside the \( \text{XFun} \) bidirectional function for the later use in backward transformation.

If the part of the view corresponding to \( t_f \) has been modified, then the backward transformation \( f' \) can bring back to the source, and similarly with \( g \). An implementation of union looks like:

\[
\begin{align*}
\text{union} & :: \text{XFilter} \to \text{XFilter} \to \text{XFilter} \\
\text{XFun f} \text{ union } \text{XFun g} & = \text{XFun } k \text{ where } \\
k s & = (t_f, t_g, k') \\
& \text{ where } \\
(t_f, f') & = f s \\
(t_g, g') & = g s
\end{align*}
\]
\[ l = \text{length} \ t_f \]
\[ k' = \text{length} \ t_g' \]
\[ (t_g', t_f') = (f', t_f') \]
\[ (t_f', t_g) = (g', t_g) \]

where

- \[ t_f' = \text{take} \ l' \ t' \]
- \[ t_g' = \text{drop} \ l' \ t' \]

Here, we assume that the both of \( t_f \) and \( t_g \) are never modified on the target view \( t \); at most one of them can be modified. We can define the bidirectional \( \text{cat} \) with the use of bidirectionalized \( \text{foldr1} \) and union as \( \text{cat} = \text{foldr1 union} \). Defining bidirectional \( \text{cat} \) in this way is, however, inefficient than that by manipulating intermediate information as follows.

\[
\text{cat} :: [\text{XFilter}] \rightarrow \text{XFilter}
\]
\[
\text{cat} [] = \text{XFun} \ k \text{ where } k s = ([], \text{const} s)
\]
\[
\text{cat} xfs = \text{XFun} \ k \text{ where }
\]
\[
\begin{align*}
\text{let } & k s = (\text{concat} \ tss, k') \\
& (tss, gs') = \text{unzip} \ (\text{map} (\lambda (\text{XFun} \ g) \rightarrow g \ s) \ xfs) \\
& \text{ls} = \text{map length} \ tss \\
& k' ts' = h \ tss \text{ ts'} \text{ gs'}
\end{align*}
\]

where

- \[ tss' = \text{unconcat} \ \text{ls} \text{ ts'} \]
- \[ \text{unconcat} [] = [] \]
- \[ \text{unconcat} (\text{ls} : \text{ts'}) = \text{take} \ l \text{ ts'} : \text{unconcat} \ (\text{drop} \ l \text{ ts'}) \]
- \[ h [] ... = s \]
- \[ h (\text{ts} : \text{tss}) (\text{ts'} : \text{tss'}) (\text{gs} : \text{gs'}) \]
- \[ \text{otherwise} = \text{gs'} \text{ ts'} \]

Other filter combinators have been bidirectionalized to produce a \( \text{XFilter} \).

### Derived Combinators

Having bidirectionalized the basic filters and filter combinators in terms of \( \text{XFun} \), most of high level combinators such as recursive ones come out to be bidirectional as they are by definition. These are same as ones in the HaXml library. Examples follow:

- \[ \text{deep} \ f = f \ ?(f :> (\text{deep} \ f \ 'o' \ \text{children})) \]
- \[ \text{foldXm} \ f = f \ 'o' \ \text{chip} \ (\text{fold Xm} \ f) \]

Finally, we have made a complete set of bidirectionalized HaXml filters and combinators, which counted over forty.

We can use this Bidirectional HaXml Library \( \text{BiHaXml} \) to write bidirectional programs for XML documents as easily as we write programs for transform the document in HaXml. Of course, we may use BiHaXml for HaXml; evaluating Bi-HaXml programs to the half way corresponds to the HaXml evaluation.

### 5 A Bidirectional XML Viewer

A bidirectional XML Viewer was built as an illustrative example of using the Bi-HaXml library for view-updating(Figure 11). The combinators defined in the Text.XML.HaXml.Combinators module of HaXml are replaced by the new \( \text{XCombinators} \) module for bidirectional programming. As the names of the combinators of the BiHaXml follow the original HaXml, existent HaXml code can be used as it is in view-updating for free.

The module \( \text{XCombinators} \) contains filters and combinators of BiHaXml, or bidirectionalized ones of the HaXml module. Several additional combinators are included for editing and combinators for Xtract query [25].

An example of bidirectional view-updating program is shown in Figure 12.
module Main where
import Graphics.UI.WX
import XCombinators
import XParse (xtract)
import XViewer (xViewer)

main :: IO ()
main = start (xViewer xfs) where
  xfs = ["Members", members],
       ["Emails", emails]
  members = mkElem "member"
             [ keep /
               > tag "name" /
               > txt ]
  emails = xtract "*/email"

Figure 12: A view-updating program

Editor operations and filters

View-updating through the target view
allows simple editing operations—“Insert”, “Delete”, and “Replace”.

Any CString element may be replaced
with a new text without any restri-
tion. But we require that inserting a new
CElem or deleting an existent CElem are
allowed provided that the related item
has been marked as “editable”. This
is because we like to keep the XML
structure with no additional information
about the document of our concern. If
we were to use type information like
DTD or XML Scheme, more elaborate
editing becomes available. But for sim-
plicity, we do not take this here.

Hence, editing in our Bidirectional
XML Viewer requires that the element
which may be deleted or inserted need to
be marked before editing, and insertion
produces a copy of the existent item
selected by the mouse. Thus, our insertion
and deletion look like a loose-leaf binder
to which a sheet of paper is inserted or
from which one is removed.

For the element to be possibly made
inserted or deleted, the filter (loosen t)
marks the element with tag t by adding
an attribute to corresponding elements,
which is effective only in the target view.
The insertion operation makes a copy of
the selected element in its place, which
will be edited further by editing opera-
tions.

Bidirectional Xtract combinators

The combinators defined in Xtract mod-
ules of HaXml are called “Double filter”
DFilter.

type DFilter = Content → Content → [Content].

A double filter always takes the whole
document as an extra argument and we
can traverse it again from inner location
within the document. These combina-
tors are used in processing Xtract queries
which are similar to XPath. In our Bi-
HaXml, the DFilters of HaXml are bidi-
rectionalized as WFilter of type

newtype WFun a b = WFun (a → a → (b, b → a))
type WFilter = WFun Content → [Content].

We have made several modifications
on the XPare module to generate
WFilters from Xtract expressions. The combi-
nator (xtract s) is a bidirectional XFilter
which returns the target view extracted
according to an Xtract expression s.

Bidirectional list filters

There is no reason why we do not include
bidirectional functions for list processing
described in Section 3.2. Our XCom-
binators module includes useful bidirec-
tional functions for lists which may be
used with original HaXml combinators.

6 Related Work

View-updating to correctly reflect the
modification on the view back to the
database [2, 6, 8, 21, 1] is an old problem
in the database community. In recent
years, however, the need to synchronize
data have been recognized by researchers
from different fields. And it is claimed
that multiple views of the same program
help to deal with in aspect-oriented pro-
gramming [13]. Recently, we have pro-
posed novel ideas on model synchroniza-
tion problems in software development
[27, 28, 29]. These all are related to bidi-
rectional transformation.

In the context of data synchroniza-
tion, [9] coined the “bidirectional up-
dating” 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 synchronization system Harmony [22]. Another 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 on bidirectional transformation was to build a theoretical foundation for presentation-oriented editors supporting interactive development of XML documents, [24] for example, under the “Dependable XML Processing Project” conducted by the author. Along with this project, another project proposed a presentation-oriented generic editor Proxima [23] 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 formalization 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 $X$ is basically a point-free functional language closely related to the languages in [16] and [9]. However, the treatments with duplication and alignment were not satisfactory. Besides XML processing languages, more primitive language $Inv$ with bidirectional semantics has been developed [18]. In order to resolve the problem of duplication and alignment, we attempt to embed both HaXml and $X$ into $Inv$. The embedding of HaXml is recorded in [19], which was preceded by a draft of implementation [10]. That for $X$ is described in [12], which is an extended version of an earlier publication [11].

From the viewpoint of domain-specific languages, another attempt of defining a language called $Bi-X$ [14] produced results of useful system for practical applications. The language Bi-$X$ shares the idea of $X$ and $Inv$ and implemented as a Java library Bi-$X$J for use in applications. We developed a tool for translating XQuery expressions into $Bi-X$ code for bidirectional queries with updates [15]. The Bi-$X$J library was used for developing an XML-based Web publishing tool called Vu-$X$ [20] which provides us for updating Web pages through browsers based on bidirectionality.

Among them, the most influential work for this paper would be [12]. The approach of this work is, however, different from the previous work on bidirectional transformation. We did not follow the way of embedding a language into another, but have defined directly in a general purpose functional language. The framework of bidirectional programming makes aware of defining bidirectional transformation with a higher order function $XFun$, which encapsulates forward and backward functions in a single function. This will make bidirectional programming be applied to wider range of problems.

7 Conclusions

We have presented a novel idea of defining bidirectional functions which comprises bidirectional programs with its application to implementation of the Bi-HaXml for XML processing. We also developed a Bidirectional XML Viewer with GUI to exemplify our idea.

With the bidirectionalization of the existent HaXml library, any HaXml transformations gain bidirectionality for free. This makes HaXml be a more powerful transformation language than it was first designed for.

We believe that bidirectional programming approach will be applied to wider domains of data update problems. Given a forward transformation, we get the backward transformation by defining bidirectional functions. A well-designed domain-specific library will free us from writing new bidirectional functions and make bidirectional programming unidi-
rectional; that is, we need only consider how to transform the source into the target with the library functions.

However, we have an issue in designing domain-specific languages. Although designing a new domain-specific language with bidirectional semantics would be an attractive approach, we always suffer from the problem of descriptive power, i.e., to what extent it works by itself. Our approach here provides a domain-specific library, BiHaXml for XML in a general purpose functional language Haskell and we are free to use language features not provided in BiHaXml. The point-free style of programming is also attractive for transformation and reasoning about programs, but fails in concise description of modularization. We may want to use definitions or bindings of expressions with identifiers, which may appear as free variables in combinator expressions. We can write a simple code for resolving ID/IDREF’s in an XML document as

\[
\text{dfilter} \quad \text{(ofoldXmlo(oiffindo "pref" lookfor okeepo))}
\]

where

\[
\text{lookfor } v = \text{global(deep (attrval("pid",AttValue[Left v])))}
\]

Here, the identifiers except \text{lookfor} are combinators of BiHaXml (and of course of HaXml), and the variable binding of \text{lookfor} appears for the function definition. Without any binding mechanism, it would be difficult to solve the problem by such simple expressions.

Another issue is a semantic one. We assume that the view produced by duplication might be modified in one data item at a time. This restriction is explained in Sections 2.3 and 3.1. We cannot use our bidirectional programs in off-line environment where several modifications are made before backward transformation. Neither we allow editing operation that changes more than two data items at a time. The restriction comes from a technical requirement for defining the semantics of backward transformation in consistent as well as concise description. As long as we update the view online using simple operations like in our XML Viewer, we do not care about it. But in more general situations we might be required to deal with more complex editing operations. Anyway, this problem can be characterized by properties of function \(h\) in bidirectional programs.

We need study more on these issues through practical applications of bidirectional programming.

Acknowledgments

The author would like to thank thank to Zhenjiang Hu and Shin-Cheng Mu for discussion on bidirectional transformation, and Sachiko Kizu for BiHaXml implementation. Also thanks to the members of the IPL laboratory, University of Tokyo.

This work is partially supported by the Japan Society for the Promotion of Science, Grant-in-Aid for Scientific Research (A) 19200002 on “Bidirectional Transformation and its Application”.

References


A The use of Bidirectional XML Viewer

The Bidirectional XML Viewer is implemented with GHC version 6.10.3 and wxHaskell 0.11.1.2 on Mac OS X, and with GHC version 6.8.3 and wxHaskell 0.10.3 on Microsoft Windows operating system. The HaXml library version 1.13.3 and the viewer module are common to both systems.

The viewer is implemented as a separate module provided as XViewer. The XFilters of BiHaXml are defined in a module XCombinators which replaces Text.XML.HaXml.Combinators module of HaXml. We needed to make small changes in the module Text.XML.HaXml.Xtract.Parse for our WFilters instead of DFilters of HaXml Xtract, and renamed the module as XParse. These are the all modules for our Bidirectional XML Viewer for view-updating.

The Bidirectional Viewer consists of several panels in a GUI frame as shown in Figure 13. This is a screenshot on Windows XP, while other screenshots will be taken on Mac OS.

![Figure 13: Bidirectional XML Viewer](image)

The topmost panel contains icons for file input and output as well as one for undoing the operation, and the status bar at the bottom of the frame indicates the XFilter currently applied to the source.

The main part of the frame consists of two panels for displaying the source and the target Content in tree form. The tree may be collapsed and expanded by pressing the node.

The Content tree for XML may contain white spaces for layout as text elements which appear most often after reading an XML document. But in most transformation of our interest, we will be happy if we disregard these white spaces from the Content by pressing the “Layout Off” button placed above the source tree area. And the layout spaces for indentation are properly inserted by pressing the “Layout On” button.

The target tree frame is associated with three buttons “Insert”, “Delete”, and “Replace”. These buttons are pressed for editing the selected element on the target tree. Pressing the Insert button makes a copy of the selected element be inserted beside that element, provided the element has been loosen (See Section 5). Similarly, the Delete button deletes the selected element among the loosened elements. Notice that deletion of the single child element of some element cannot get it back by insertion because insertion requires the element to be copied. When the Replace button is pressed, a textbox appears to accept the string to be replaced for the selected text element. Replacement is always possible for any text element unless it has been produced only for reference. See Section B below for ID/IDREF resolution.

The panel between the source and the target tree areas is allocated for the buttons placed for each XFilter to be applied to the source tree.

The topmost button for “Xtract” is provided for interactive querying by Xtract patterns, which the user gives through a popup textbox. And the “Keep” filter is also predefined as the default, which keeps the source to produce the target as it is.

Other filters are programmed by the user as shown in Figure 14. XFilters are defined by a variable associated with a combinator expression and a name is given for displaying it in the button. The name “Authors” is given to the filter def-
inition of “authorlist” as shown in the filter list xfs, which is the argument of the viewer xViewer.

module Main where
import Graphics.UI.WX
import Text.XML.HaXml.Types
import XCombinators
import XParse (xtract)
import XViewer (xViewer)
main = start (xViewer xfs) where
  xfs = [ ("Authors", authorlist),
         ("Papers", paperlist) ]
  authorlist = xtract "//author" xsort xremdups
  paperlist = loosen ["paper", "author"

Figure 14: XFilters for View-updating

The editing operations on the target view are immediately reflected on the source and then on the target as described in Section 2.2. If you want to see what happens, the “Undo” button will help showing the process backward in a step-by-step way: first the forward transformation from the modified source with the modification of the source by the backward transformation is canceled, and then the modification on the target tree is canceled.

In the followings, we exemplify the use of the viewer with a small XML document excerpted from the publication list of the references, which contains 7 publications and of the form as Figure 15. We will learn bidirectional programming with BiHaXml through several XFilter examples.

We start editing by reading the XML file and get the initial screen with the same source and the target trees.

A.1 Querying by Xtract

The Xtract tool of the HaXml can be used as a kind of “XML-grep” at the command line as well as a CFFilter. We can use it in our Bidirectional XML Viewer by simply putting query patterns interactively. The result of the query is shown as the target view of forward transformation. And editing operations may be taken for updating the source through this view by bidirectional settings. When we press the “Xtract” filter button, another textbox appears for the user to put an Xtract pattern (Figure 16). Putting the pattern to query, we will get the result on the target tree area. Figure 17 shows the result of the query with the pattern “//title”, which means the title elements found at any depth of the publist, the root of the source tree.

Note that our XFilter may generate a list with more than two Content elements as observed in this case. The viewer makes a virtual root with the tag * to make a tree with these elements as children.

Interactive query by Xtract patterns is useful for designing HaXml programs through examination of access paths to the element.

A.2 Making them unique

We could make a query for the authors in the publication list with the pattern
“//author”. The XFilter (xtract “//author”) does this much the same way as in interactive query. Combined this with bidirectional functions xsort and xremdups described in Section 3.2, we may define a XFilter authorlist as shown in Figure 14.

This filter first collects author’s names which appear in any authors’ lists of any papers by the extract filter. Then these names are sorted, and finally adjacent duplicates are removed. Hence only the different names in the original documents appear in the target view. Figure 18 shows the result applied to our XML document. Here, we can see that similar names appear; one is written with initial of the first name but the other written in full name.

Such inconsistency is corrected with ease by replacing the name with correct one. To do this, select the text element and press the Replace button. A textbox is popped up for text input (Figure 19).

When “OK” button is pressed, the backward transformation brings the change to the source and then do the forward transformation again to get the view. This time, only the name with initial appears (Figure 20).
A.3 Inserting new items

As far as text editing, we can freely replace the text element as shown above. However, we need to invent some mechanism of editing for structured elements. As described in the leading part of the Appendix, we provide a XFilter for marking the “editable” element. A simple code (loosen ["paper", "author"]) marks all the element with tags paper or author at any depth of the current element.

Hence, the filter called “Papers” defined in Figure 14 applied to the source document produces the target shown in Figure 21. The mark is shown as ="*", which is implemented internally as an attribute with the “empty” key of the value *.

If we want to insert a new “paper” element in front of the first one, select the first element and press the Insert button. This produces a copy of the first element inserted in front both in the target and the source trees as shown in Figure 22.

Now we can change the text elements under the inserted copy, or may insert or delete the “author” elements which have been marked editable, too.

We can see how this loosening mechanism works well if we undo the operation. The internal attribute value for the key "=" is changed to "+" or to "-" respectively for insertion or deletion before backward transformation. This keeps the number of concatenated elements produced by the forward transformation. As described in Section 4.2, the number of the results of component filters is the key issue of backward transformation. Insertion and deletion is performed in effect as the backward transformation of the loosen filter.

A.4 Publishing Web Pages

An attractive application of XML documents would be XML-based Web Publishing. Web publishing based on HTML has problems is how to maintain the consistency of the contents. Consider, for example, the author’s name need to be changed for some reasons in Web pages of the publication list. If the name is originated from a single instance in a XML document and the HTML views are produced from that document, what we have to do is to change the single instance in that document. Otherwise, we need to find out where the name appears in the HTML document; there may scatter in many places in many pages and we have to take the trouble to make them consistently updated. Thus, we apply
our bidirectionality with duplication by splitting to produce Web pages from an XML document.

An example bidirectional program with BiHaXml is shown here as Figure 23.

```html
publish_html=
  mkElem "html" [
    mkElem "body" [
      mkElem "h3" [literal "Authors"],
      mkElem "ul"
        [replaceTag "li" 'o' authorlist],
      mkElem "h3" [literal "Papers"],
      mkElem "ol"
        [mkElem "li"
           [replaceTag "li" 'o'
              (keep /* tag "title"),
              literal "", 
              replaceTag "i" 'o' (keep /* tag "year"),
              mkElem "li" [
                replaceTag "li" 'o'
                  (deep (tag "author"))]]
          'o'
          (keep /* tag "paper"))]]
    ]
]
```

We have a view produced by the code publish_html for our Publist.xml as shown in Figure 24.

The view contains duplications for the authors' name. One appears in the list of the authors, and several occurrences in the authors' part in each paper. Of course, we may modify the contents by editing the target view with XHTML tags and the changes will be back to the source as examples above show. This is the basic idea of our previous work on Vu-X [20], which is provided an interface for editing through the view on Web browsers.
By pressing the “Take” icon and giving the file name, we get the HTML file which can be viewed with popular Web browsers like Safari, Internet Explorer, and Firefox. Figure 25 is an example displayed on the Safari browser.

B Resolving ID/IDREF

In designing the database of publications, the uniqueness of the authors’ name would be realized by the use of the ID/IDREF mechanism in some ways.

Given an XML document which corresponds to the view in HTML format above. We assume that we have an XML document which contains the Authors element keeping the names of the authors, and the Papers element for the publications. The paper elements of the Papers element contains the author elements which refer to an author element in the authors element.

This reference is represented by the ID/IDREF mechanism with the use of attribute; we use ID key of pid and IDREF key of pref. Figure 26 shows the XML file for our publication lists with ID/IDREF.

As mentioned in Conclusions, the filter
dfilter
(ofoldXmlo(ofindoo."pref" lookfor okeepo))
where
lookfor v = global(deep (attrval("pid",AttValue[Left v])))
makes the IDREF element replaced by the corresponding ID element which is shown in Figure 27.

We should notice that the author elements under the Papers element should not be modified on the target view, because those elements are references to the elements which define the unique names. In fact, in our implementation of the Wfilters, the text element in the target view cannot be replaced with. But there remain the problems how to build XML documents with ID/IDREF from simple ones, for example.

Figure 26: A Document with ID/IDREF

Figure 27: Resolving ID/IDREF