Bi-X Core: A General-Purpose Bidirectional Transformation Language

Dongxi Liu Keisuke Nakano Yasushi Hayashi Zhenjiang Hu Masato Takeichi
Akimasa Morihata Yingfei Xiong

Department of Mathematical Informatics, University of Tokyo
{liu,ksk,hayashi,hu,takeichi}@mist.i.u-tokyo.ac.jp
{morihata,xiong}@ipl.t.u-tokyo.ac.jp

Bi-X Core is a general-purpose bidirectional transformation language, aiming to implement various systems that need synchronization between their input data and output data. In syntax, Bi-X Core is a first-order λ-calculus extended with two structured data types, tuple and variant. For ease of use, some functional languages with more syntactic sugar can be defined based on Bi-X Core. The technical problem we solve in this paper is how to define bidirectional semantics for a general-pupose functional language. Bi-X Core is an ongoing work, and some examples are presented to demonstrate its usefulness.

1 Introduction

In the information world, data are often transformed from one format into another for the reasons like being used by different applications or devices. When some data are changed, it is desirable to keep all their replicas synchronized. Using languages like XSLT or Java, we can write programs for data transformations, but these programs cannot provide direct help for data synchronization. That means we have to think about other mechanisms to implement data synchronization. This case can be remedied by using bidirectional transformation languages, whose programs can not only implement data transformation, but also provide help to data synchronization.

The programs of bidirectional transformation languages can run in two directions, called forward direction and backward direction, respectively. In the forward direction, they transform source data into target data, as done by programs of ordinary languages. In the backward direction, from the updated target data and the original source data, they produce new source data which reflect the updates made to the target data. Due to this feature, bidirectional transformation languages have played an central role in some synchronization systems [1, 2].

There have been several bidirectional languages, each of which is designed for a particular application domain. The bidirectional languages in [3, 4, 5, 6] are for transforming tree-strucutred data or XML data; the language in [7] is used to define bidirectional queries on relational databases; and the languages in [2] and [8] are being designed for processing software models or Java source code, respectively.

Since all existing bidirectional languages are domain-specific, to address the bidirectional transformation need from a new domain, one idea is that we design a new bidirectional language for this domain. But developing a new language is not an easy work, and it becomes worse if there are many new domains that want bidirectional transformation. In this paper, we take another idea. That is, we will design a general-purpose bidirectional transformation language, and for applications in a new domain, programmers can use this general-purpose language to define domain-specific data structures and abstractions, which are then used to implement applications.

The language we will define in this paper is a functional language, called Bi-X Core, which is a first-order λ-calculus extended with two structured data types, tuple and variant. In order to make Bi-X Core a bidirectional language, we need to de-

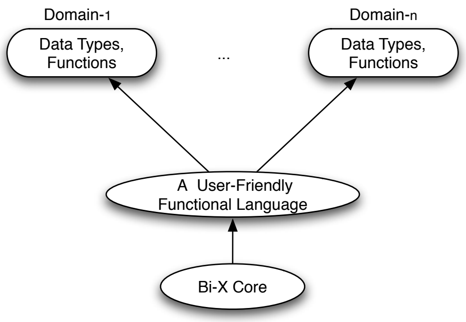

Figure 1: Framework of Using Bi-X Core

fine the bidirectional semantics for each of its constructs. Bi-X Core is not designed as a language for programmers. It aims to provide a foundation to interpret other user-friendely general-purpose functional languages.

In this paper we give a way to define bidirectional semantics for a general-purpose functional language. This is the contribution of this paper.

The rest of this paper is organized as follows. Section 2 gives the framework of using Bi-X Core to define bidirectional applications. Section 3 defines the bidirectional semantics of Bi-X Core. Section 4 introduces a functional language with more syntactic sugar that can be interpreted by Bi-X Core. With the help of this user-friendly language, Section 5 gives several bidirectional applications without using domain-specific language. Section 6 discusses the related work and Section 7 concludes the paper.

2 Framework for General-Purpose Bidirectional Transformation

Figure 1 shows the framework of using Bi-X Core to implement bidirectional transformation in different domains. For ease of programming, Bi-X Core is generally wrapped by another user-friendly functional language with more syntactic sugar. For example, this user-friendly language can have the same syntax as Haskell or OCaml.

Each domain consists of some data types to describe the data structure used in this domain and some functions to define a set of abstractions specific to this domain. For example, for bidirectional XML transformation, the data type can be a tree representing XML elements, and the functions include those for getting or setting the element tags, getting or setting the element contents, and so on.

$$
\begin{array}{lcl}
v & ::= & n \mid (v_i^{i\in 1..n})^u \mid \ < l, v >^u \\
u & ::= & \texttt{non} \mid \texttt{mod} \mid \texttt{ins} \mid \texttt{del} \\
e & ::= & x \mid n \mid \texttt{let}\ f(x) = e' \ \texttt{in}\ e \mid f(e) \\
 & & \mid (e_i^{i\in 1..n}) \mid e.i \mid \ < l, e > \\
 & & \mid \texttt{case}\ e\ \texttt{of}\ < l_i, x_i > [f_i] => e_i^{i\in 1..n}
\end{array}
$$

Figure 2: Syntax of Bi-X Core

3 Bi-X Core and Its Bidirectional Semantics

The syntax of Bi-X Core is given in Figure 2. An expression e in this language can be a variable, an integer, an expression with a local function declaration, a function application, or an expression for constructing or deconstructing tuples or variants. All values in Bi-X Core are annotated with the updating status u: `non` for the original values, `mod` for the values that have been modified, `del` for values to be deleted and `ins` for newly inserted values. Each variant expression or value has a label l, which is used by the `case` expression to classify the value in a variant. In this and the next section, the notation $\Box_i^{i\in 1..n}$ is used to indicate the syntax object $\Box$ can appear zero or more times. In the following, we will define the bidirectional semantics of Bi-X Core in the form of big-step operational semantics.

3.1 Notations

There are two judgments used to define the bidirectional semantics for Bi-X Core. The first judgment is given below, used for forward semantics.

$$D; C \rhd e \Rightarrow v$$

which means the expression e evaluates to the value v under the function-declaration context D and the variable-binding context C.

The second judgment is for backward semantics, with the following form.

$$D; E; v \rhd e \Rightarrow E'$$

which means under the contexts D, E and the updated value v, backward execution of e will generate a new context E'. The context E (and E') maps variables to pairs of values, each of which consists of the original value and the updated value for the bound variable.

The backward execution of some expressions need to invoke forward execution of their subexpressions. At this time, we need a context C, which can be built from E in the current backward execution by keeping only the first component of the value for each bound variable. This operation is written as $E.1$.

An empty context is represented by a dot ".". Two contexts C_1 and C_2 (or a context C and a variable binding $x \mapsto v$) can be concatenated into a new context, written as C_1, C_2 (or C, $x \mapsto v$). Concatenations for contexts E and D are also represented in this way.

3.2 Bidirectional Semantics

We will define the bidirectional semantics for each form of expression. In this section, we only consider updated values that are modified or deleted.

The bidirectional semantics of a variable x is defined by the following two rules. The forward execution returns the value of the most recent variable x; the backward execution updates the value of x by merging updates made to its different replicas. This expression corresponds to the variable reference combinator in [6]. The detailed definition of mg operator is omitted in this paper.

$$\frac{C = C_1, x \mapsto v, C_2 \quad x \notin \mathit{Dom}(C_2)}{D; C \rhd x \Rightarrow v}$$

$$\frac{x \notin \mathit{Dom}(E_2) \quad E' = E_1, x \mapsto (v_1, \mathtt{mg}(v_2, v)), E_2}{D; E_1, x \mapsto (v_1, v_2), E_2; v \rhd x \Rightarrow E'}$$

The bidirectional semantics of a constant n is simple. Forward execution returns this constant, and the backward execution returns the current context. A constant is not allowed to change.

$$\frac{}{D; C \rhd n \Rightarrow n^{\mathtt{ori}}} \quad \frac{}{D; E; n^{\mathtt{ori}} \rhd n \Rightarrow E}$$

For an expression with a local function declaration, $\mathtt{let}\ f(x) = e'\ \mathtt{in}\ e$, we first append this function declaration to the rear of the current context D, and then evaluate the scope expression in forward or backward directions.

$$\frac{D, f(x) = e'; C \rhd e \Rightarrow v}{D; C \rhd \mathtt{let}\ f(x) = e'\ \mathtt{in}\ e \Rightarrow v}$$

$$\frac{D, f(x) = e'; E; v \rhd e \Rightarrow E'}{D; E; v \rhd \mathtt{let}\ f(x) = e'\ \mathtt{in}\ e \Rightarrow E'}$$

The expression $f(e)$ has a forward semantics same as the usual function applications, executing the argument e first, and then executing the function body with the function parameter bound to the value of e. Its backward semantics is defined by executing the function body first and then executing the argument e with the updated value generated by the backward execution of function body. This way of defining bidirectional function calls is also used in [6].

$$\frac{\begin{array}{l} D = D_1, f(x) = e', D_2 \quad f \notin \mathit{Dom}(D_2) \\ D; C \rhd e \Rightarrow v \quad D; C, x \mapsto v \rhd e' \Rightarrow v' \end{array}}{D; C \rhd f(e) \Rightarrow v'}$$

$$\frac{\begin{array}{l} D = D_1, f(x) = e', D_2 \quad f \notin \mathit{Dom}(D_2) \\ D; E.1 \rhd e \Rightarrow v' \\ D; E, x \mapsto (v', v'); v \rhd e' \Rightarrow E', x \mapsto (v', v'') \\ D; E'; v'' \rhd e \Rightarrow E'' \end{array}}{D; E; v \rhd f(e) \Rightarrow E''}$$

The rest of expressions are related to construct and deconstruct tuples and variants. They are the main constructs in Bi-X Core to represent other data structures, such as trees or graphs.

The expression $(e_i^{i \in 1..n})$ is to construct tuples in its forward execution by executing each subexpression e_i, which generates the ith tuple component. In its backward execution, all subexpressions will be executed backwardly one by one to get an environment that reflects all updates made to all components in the updated tuple.

$$\frac{D; C \rhd e_i \Rightarrow v_i \quad (i \in 1..n)}{D; C \rhd (e_i^{i\in 1..n}) \Rightarrow (v_i^{i\in 1..n})^{\texttt{ori}}}$$

$$\frac{\begin{array}{l} D; E; v_1 \rhd e_1 \Rightarrow E_1 \\ ... \\ D; E_{n-1}; v_n \rhd e_n \Rightarrow E_n \end{array}}{D; E; (v_i^{i\in 1..n})^u \rhd (e_i^{i\in 1..n}) \Rightarrow E_n}$$

The expression $e.i$ is to extract the ith component of the tuple returned by e in its forward execution. And in the backward execution, this tuple is updated by replacing its ith component with the updated component v, and then the updated tuple will be used by e to get an updated context.

$$\frac{D; C \rhd e \Rightarrow (v_i^{i\in 1..n})^{\texttt{ori}}}{D; C \rhd e.i \Rightarrow v_i}$$

$$\frac{\begin{array}{l} D; E.1 \rhd e \Rightarrow (v_i^{i\in 1..n})^{\texttt{ori}} \\ D; E; (v_h^{h\in 1..i-1}, v, v_j^{j\in i+1..n})^{\texttt{ori}} \rhd e \Rightarrow E' \end{array}}{D; E; v \rhd e.i \Rightarrow E'}$$

The expression $< l, e >$ is to construct a labeled value. The value is the result of executing e forwardly, with the label l. Note that the label cannot be changed in the updated value in the backward execution.

$$\frac{D; C \rhd e \Rightarrow v}{D; C \rhd < l, e > \Rightarrow < l, v >^{\texttt{ori}}}$$

$$\frac{D; E; v \rhd e \Rightarrow E'}{D; E; < l, v >^u \rhd < l, e > \Rightarrow E'}$$

The `case` expression distinguishes the label on the labeled value returned by e and for each possible label l_i, there is a corresponding branch $< l_i, x_i > [f_i] => e_i$. If a branch maches, then the corresponding expression e_i will be executed with x_i bound to this value without label, and moreover the function f_i must return `true`, which represents the variant $< \text{true}, () >$, on the resulting value of e_i. In the backward execution, the variable binding in this expression is processed as that in function applications.

$$\frac{\begin{array}{l} D; C \rhd e \Rightarrow < l_i, v_i >^{\texttt{ori}} \\ D; C, x_i \to v_i \rhd e_i \Rightarrow v \\ D; . \rhd f_i(v) \Rightarrow \texttt{true} \end{array}}{D; C \rhd \texttt{case}\ e\ \texttt{of}\ < l_i, x_i > [f_i] => e_i^{i\in 1..n} \Rightarrow v}$$

$$\frac{\begin{array}{l} D; E.1 \rhd e \Rightarrow < l_i, v_i >^{\texttt{ori}} \\ D; E, x_i \to (v_i, v_i); v \rhd e_i \Rightarrow E', x_i \to (v_i, v_i') \\ D; E'; < l_i, v_i' >^{\texttt{ori}} \rhd e \Rightarrow E'' \end{array}}{D; E; v \rhd \texttt{case}\ e\ \texttt{of}\ < l_i, x_i > [f_i] => e_i^{i\in 1..n} \Rightarrow E''}$$

3.3 Discussion of Insertion

It is hard to put inserted values back since there are no corresponding original source values available to guide the construction of updated source values. In order to to process insertions, we first add a special value Ω to the syntax of values, which means a non-existing value. This special value is also used in [3]. The semantics of some deconstructing expressions have to be extended to consider this special value. For example, the semantics of case expression on variants is extended by the following two rules. Note that the condition function f is used to choose branch in the backward execution.

$$\frac{D; C \rhd e \Rightarrow \Omega}{D; C \rhd \texttt{case}\ e\ \texttt{of}\ < l_i, x_i > [f_i] => e_i^{i\in 1..n} \Rightarrow \Omega}$$

$$\frac{\begin{array}{l} D; E.1 \rhd e \Rightarrow \Omega \quad D; . \rhd f_i(v) = \texttt{true} \\ D; E, x_i \to (\Omega, \Omega); v \rhd e_i \Rightarrow E', x_i \to (\Omega, v_i') \\ D; E'; < l_i, v_i' >^{\texttt{ins}} \rhd e \Rightarrow E'' \end{array}}{D; E; v \rhd \texttt{case}\ e\ \texttt{of}\ < l_i, x_i > [f_i] => e_i^{i\in 1..n} \Rightarrow E''}$$

Another point we noticed for insertions is that a new value can be inserted into a structure only if this value can appear in this structure regularly. For example, an integer list contains the regular occurrences of integers, so a new integer can be inserted into this list. To verify our ideas, we add into Bi-X Core some introduction forms and elimination forms about list. In particular, the construct `map` is directly responsible for processing inserted elements, whose backward semantics for insertion is defined as the following rule, which depends on another auxiliary rule, also shown below.

$$\frac{\begin{array}{l} D; E.1 \rhd e \Rightarrow v \\ D; E; v_1 \Diamond f, e \Rightarrow E' \\ D; E'; v_2 \rhd \texttt{map}\ f\ e \Rightarrow E'' \end{array}}{D; E; \texttt{cons}^{\texttt{ins}}\ v_1\ v_2 \rhd \texttt{map}\ f\ e \Rightarrow E''}$$

$Prog$	::=	$TopDecl_i^{i \in 1..n}$
$TopDecl$	::=	$TypeDecl \mid FDecl$
$FDecl$	::=	$FunDecl \mid FunSigDecl$
$TypeDecl$	::=	$\texttt{data}\ DName\ Var_i^{i \in 1..n} = Contrs$
$Constrs$	::=	$CName\ Ty_i^{i \in 1..n}$
		$\mid CName\ Ty_i^{i \in 1..n}\ Bar\ Constrs$
Ty	::=	$\texttt{Int} \mid \texttt{Char} \mid \texttt{Bool}$
		$[Ty] \mid (Ty_i^{i \in 1..n})$
		$\mid DName \mid \{Lbl_i : Ty_i^{i \in 1..n}\}$
$FunDecl$	::=	$FName\ Pat_i^{i \in 1..n} = FunBody$
$FunBody$	::=	$Exp \mid \{FName\}\ Exp$
		$\mid Exp\ \texttt{where}\ FDecl_i^{i \in 1..n}$
		$\mid \{FName\}\ Exp\ \texttt{where}\ FDecl_i^{i \in 1..n}$
Pat	::=	$Var \mid \_ \mid [] \mid Pat; Pat$
		$\mid Constr\ Pat_i^{i \in 1..n}$
		$\mid (Pat_i^{i \in 1..n}) \mid \{Lbl_i = Pat_i^{i \in 1..n}\}$
Exp	::=	$Var \mid n \mid c \mid \texttt{True} \mid \texttt{False}$
		$\mid \texttt{let}\ FunDecl_i^{i \in 1..n}\ \texttt{in}\ Exp$
		$\mid Name\ Exp_i^{i \in 1..n}$
		$\mid \{Lbl_i = Exp_i^{i \in 1..n}\} \mid Exp.Lbl$
		$\mid \texttt{if}\ Exp\ \texttt{then}\ Exp\ \texttt{else}\ Exp$
		$\mid (Exp_i^{i \in 1..n}) \mid [Exp_i^{i \in 1..n}]$
		$\mid Exp : Exp \mid \texttt{map}\ Name\ Exp$
		$\mid \texttt{case}\ Exp\ \texttt{of}\ Branch_i^{i \in 1..n}$
$Branch$	::=	$Pat \Rightarrow Exp \mid Pat\ \{Name\} \Rightarrow Exp$
$FunSigDecl$	::=	$Name :: FunTy$
$FunTy$	::=	$Ty \mid Ty \rightarrow FunTy$
Bar	::=	$\mid$

Figure 3: Syntax of A User-Friendly Language

$$\frac{\begin{array}{l} D = D_1, f(x) = e', D_2 \quad f \notin Dom(D_2) \\ D; E, x \mapsto (\Omega, \Omega); v \rhd e' \Rightarrow E', x \mapsto (\Omega, v') \\ D; E'; v' \rhd e \Rightarrow E'' \end{array}}{D; E; v \Diamond f, e \Rightarrow E''}$$

4 A User-Friendly Functional Language

Figure 3 shows a general-purpose functional language that can be interpreted with Bi-X Core and has more syntactic sugar. A program of this language is composed of a list of data type declarations, function declarations and function signature declarations, at its top level. The design of this language get much inspired by the syntax of Haskell and OCaml. However, compared with these primary functional languages, this language needs the following syntactic option to support backward execution: overloaded functions and branches in case expressions might be annotated with guards if they are expected to accept inserted values in backward execution. The guards take the form of $\{Name\}$, where $Name$ is a boolean-valued function, and this function must hold on the value returned by the guarded function or case branch. These guards will be translated into the guards of `case` expressions in Bi-X Core, which will help determine which case branch will be chosen to process an inserted value. If programmers do not care the guards on functions or case branches, then they can omit them when programming.

In comparison with other bidirectional transformation languages, this user-friendly language allows programmers to define their own data types. This is the feature provided by Bi-X Core . We will see several examples of this language in the next section. Note that list type $[Ty]$ and the function `map` are pre-defined in this language. As discussed in the previous section, they are specially treated to process inserted values.

The semantics of this user-friendly language is defined by translating it into Bi-X Core. This translation is quite similar to the translation, for instance, from Haskell into Haskell kernel [9], or from Standard ML to λ-calculus [10]. This is an ongoing work, and we will not give its details here.

5 Applications of Bidirectional Transformation

In this section, we will demonstrate Bi-X Core through several examples with the help of the user-friendly language introduced in the previous section. These examples need different data structures, which can be defined as algebraic data types by using the keyword `data` in the user-friendly language. Some examples below need the type `String`, which can be defined as [`Char`]. For convenience, we write a string as, for instance, “abc”, rather than [‘a’, ‘b’, ‘c’].

5.1 Bidirectional XML Transformation

This example demonstrates how to define XML transformation with Bi-X Core. The data types for representing XML documents and some functions

```
data Element = Elm String [Cont]
data Text    =  Txt String
data Cont =  ElmCont Element | TxtCont Text

getTag:: Element -> String
getTag Elm tag _ = tag

setTag:: String -> Element -> Element
setTag newtag Elm tag cont
  = Elm newtag cont

getCont:: Element -> [Cont]
getCont Elm _ cont =  cont

setCont:: [Cont] -> Element -> Element
setCont newcont Elm tag cont
  = Elm newtag newcont

getContElm:: Content -> Element
getContElm ElmCont elm =  elm
getContElm TxtCont str =  Elm "null" []

getContTxt:: Content -> Text
getContTxt ElmCont elm =  Txt "null"
getContTxt TxtCont txt =  txt
```

Figure 4: Data Types and Functions for XML Transformation

to process them are given in Figure 4.

Suppose we have the following XML data including two elements.

```
<book>
  <title>Algorithm</title>
  <price>38</price>
</book>
<journal>
  <title>Cooking</title>
  <price>10</price>
</journal>
```

Using the data types and functions in Figure 4, users can represent the above data and write a transformation to extract all book or journal titles, as shown in Figure 5.

5.2 Other Applications

Following the way of defining bidirectional XML transformation, we can implement a bidirectional code query system [8] by first defining the data structure to represent the abstract syntax tree of Java programs, and then for each type of data, implementing suitable processing functions. An an ex-

```
[
Element "book" [
  ElmCont
    Elm "title" [TxtCont Txt "Algorithm"],
  ElmCont
    Elm "price" [TxtCont Txt "38"] ],
Element "magazine" [
  ElmCont
    Elm "title" [TxtCont Txt "Cooking"],
  ElemCont
    Elm "price" [TxtCont Txt "10"] ]
]

allTitles :: [Element] -> [Element]
allTitles bjs = map title bjs
  where
    title::Element -> Element
    title elm =  case getContent  elm
      of cont:_  => getContElm cont
```

Figure 5: An Example of XML transformation

ample, for Java classes, the functions for getting and setting methods are needed.

Another example is to define bidirectional model transformation system [2]. The idea is still similar, first defining the data structure to represent software models, and then defining functions to process the data of each type. For example, for each model entity, there should be functions for getting or setting its attributes by names.

6 Related Work

Foster et al. [3] first propose the way of defining bidirectional language by giving each language construct bidirectional semantics. But the language in [3] includes only a collection of domain-specific combinators. It is not clear how expressive these combinators could be, and what combinators should be added into their language to increase its expressiveness. In contrast, Bi-X Core is a general-purpose functional language. Hence, by comparing it with other functional languages, we can know easily its expressiveness and what new constructs should be added for more expressiveness. For example, it can be extended with monad as that in Haskell, or reference as that in OCaml.

The language in [6] has considered bidirectional variable bindings and function calls, but that language does not allow programmers to define their own data types. Bi-X Core also takes the same way to define the bidirectional semantics of variable bindings and function calls, but it does this for a general-purpose language, and moreover it allows user-defined data types.

Some other related work include the injective language in [11] and the reversible language in [12]. These languages can only express injective functions, so that their programs can be inverted. Bi-X Core can express functions not necessarily injective, with the cost that the backward execution may fail due to, for instance, conflicting updates.

7 Conclusion

In this paper, we described a general-purpose bidirectional transformation language, Bi-X Core, which is a first-order λ-calculus extended with two structured data types, tuple and variant. With this language, there is no need to design new domain-specific languages for new domains that want bidirectional transformation. By designing Bi-X Core, we showed a way of how to define bidirectional semantics for general-purpose functional language. Bi-X Core is still under development. Some applications in different domains are being developed to validate its usefulness.

8 Acknowledgment

This work is supported by Comprehensive Development of e-Society Foundation Software Program of the Ministry of Education, Culture, Sports, Science and Technology, Japan. We thank Izumi Mihashi and Kiminori Matsuzaki for providing the helpful comments and LaTeX tips.

References

[1] Harmony Project. http://www.seas.upenn.edu/ harmony.

[2] Yingfei Xiong, Dongxi Liu, Zhenjiang Hu, and Masato Takeichi. A bidirectional transformation approach towards automatic model synchronization. In *Participants Workshop of GTTSE Summer School*, pages 359–360, 2007.

[3] J. Nathan Foster, Michael B. Greenwald, Jonathan T. Moore, Benjamin C. Pierce, and Alan Schmitt. Combinators for bi-directional tree transformations: a linguistic approach to the view update problem. In *Proceedings of the 32nd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 233–246. ACM Press, 2005.

[4] Zhenjiang Hu, Shin-Cheng Mu, and Masato Takeichi. A programmable editor for developing structured documents based on bidirectional transformations. In *Proceedings of the 2004 ACM SIGPLAN symposium on Partial evaluation and semantics-based program manipulation*, 2004.

[5] Dongxi Liu, Zhenjiang Hu, Masato Takeichi, Kazuhiko Kakehi, and Hao Wang. A java library for bidirectional XML transformation. *JSSST Computer Software*, 24(2):164–177, 2007.

[6] Dongxi Liu, Zhenjiang Hu, and Masato Takeichi. Bidirectional interpretation of xquery. In *PEPM '07: Proceedings of the 2007 ACM SIGPLAN workshop on Partial evaluation and semantics-based program manipulation*, pages 21–30. ACM Press, 2007.

[7] Aaron Bohannon, Jeffrey A. Vaughan, and Benjamin C. Pierce. Relational lenses: A language for updateable views. In *Proceedings of the 25th ACM symposium on Principles of Database Systems*, 2006.

[8] Dongxi Liu, Yingfei Xiong, Zhenjiang Hu, and Masato Takeichi. Bi-CQ: A bidirectional code query language. In *Participants Workshop of GTTSE Summer School*, pages 348–349, 2007.

[9] Haskell 98 Language and Libraries, 2002. http://haskell.org/onlinereport/index.html.

[10] Robert Harper and Chris Stone. A type-theoretic interpretation of Standard ML. In Gordon Plotkin, Colin Stirling, and Mads Tofte, editors, *Proof, Language, and Interaction: Essays in Honor of Robin Milner*. MIT Press, 2000.

[11] Shin-Cheng Mu, Zhenjiang Hu, and Masato Takeichi. An algebraic approach to bi-directional updating. In *APLAS*, volume 3302, pages 2–20, 2004.

[12] Tetsuo Yokoyama and Robert Glück. A reversible programming language and its invertible self-interpreter. In *PEPM '07: Proceedings of the 2007 ACM SIGPLAN symposium on Partial evaluation and semantics-based program manipulation*, pages 144–153. ACM Press, 2007.