An Environment for Maintaining Computation Dependency in XML Documents

Dongxi Liu
University of Tokyo
Tokyo, Japan
liu@mist.i.u-tokyo.ac.jp

Zhenjiang Hu
University of Tokyo
Tokyo, Japan
hu@mist.i.u-tokyo.ac.jp

Masato Takeichi
University of Tokyo
Tokyo, Japan
takeichi@mist.i.u-tokyo.ac.jp

ABSTRACT

In the domain of XML authoring, there have been many tools to help users to edit XML documents. These tools make it easier to produce complex documents by using such technologies as syntax-directed or presentation-oriented editing, etc. However, when an XML document contains data with some computation dependency among them, these tools cannot free users from the burden of maintaining this dependency relationship. By computation dependency, we mean that some data are gotten by computing from other data in the same document.

In this paper, we present an environment for authoring XML document, in which users can express the data dependency relationship in one document explicitly rather than implicitly in their minds. Under this environment, the dependent parts of the document are represented as expressions, which in turn can be evaluated to generate the dependent data. Therefore, users need not to compute the dependent data first and then input them manually, as required by the current authoring tools.

Categories and Subject Descriptors

D.1.1 [**Programming Techniques**]: Applicative (Functional) Programming; H.4.1 [**Information Systems Application**]: Office Automation—*spreadsheet, word processing*

General Terms

Documentation, Languages

Keywords

Computation Dependency, Functional Programming, Lazy Evaluation, Programmable Structured Document, XML

DocEng'05, November 2–4, 2005, Bristol, United Kingdom.

1. INTRODUCTION

XML is a popular format for data exchange in information systems. With this standard representation, a large number of XML documents are created and used in many different areas, like web service, e-commerce and databases. In order to facilitate the development of XML documents, many authoring technologies or tools, either in academic or in industry, are proposed. These technologies or tools improve authoring efficiency from several aspects. First, graphical interfaces are always provided by commercial tools, such as XMLSpy[3] and <oXygen/> XML Editor [18], which give many useful features to ease the work of editing. For example, editing operations can be selected from menus, and different kinds of document constructs are highlighted with different colors. Second, syntax-directed technology, either by analyzing schema [3, 18] or mining sample documents [6], is able to advise the possible elements or attributes under the current context, which helps users to focus more on contents than on document structures. Third, presentation-oriented technologies allow users to edit on appropriate document views in the WYSIWYG manner and the underlying XML documents are generated or updated automatically, which is used in tools such as XEditor [9], XMLSpy [3] and XMLmind [13].

Though these technologies and tools are useful, they cannot capture the computation dependency relationship among different parts of one document. Here, computation dependency means that some parts of the document should be computed from other parts of the same document. When using the current editors to create such kind of XML documents, the user has to keep in his mind this relationship, and updates the dependent parts manually when depended ones are modified. This case can be motivated by the following simple example.

Consider an XML file, shown in Figure 1, containing the data about a class. This class includes three students, and each student takes two courses. The average score of each student depends on the scores of the courses taken by him. This file also contains the information of how many students in this class and the number of students who have qualified average scores (greater than 60 in this example). The former is gotten by counting the number of `student` elements, and the latter depends on the number of `average` elements with values greater than 60.

When creating this kind of document with current XML editors, the users have to compute the dependent values by themselves in some ways external to the editors. For example, if the file is simple, they can do it in their minds;

```
<class>
  <students>
    <student>
      <name>Tom</name>
      <courses>
        <science>81</science><art>87<art>
      </courses>
      <average>84</average>
    </student>
    <student>
      <name>Peter</name>
      <courses>
        <science>48</science><art>70<art>
      </courses>
      <average>59</average>
    </student>
    <student>
      <name>Peter</name>
      <courses>
        <math>92</math><art>86<art>
      </courses>
      <average>88</average>
    </student>
  </students>
  <total_students>3</total_students>
  <qualified_students>2</qualified_students>
</class>
```

Figure 1: Computation Dependency in Document

otherwise, they have to write programs in XML oriented languages, such as Java [17] or CDuce [4], to compute dependent values. Obviously, these are boring and error-prone procedures for interactive authoring of XML documents.

In this paper, we propose a new environment for authoring XML documents, which not only supports explicit and concise specification of dependent relationship among tree nodes, but also efficiently solves the dependency to keep content consistency. The main contributions of the paper can be summarized as follows.

- *A simple user interface for specifying dependency.* Our environment provides a simple user interface, enabling users even with little knowledge of XML to describe data dependency among nodes in XML documents easily. Being analogous to the popular speadsheet applications like Microsoft Excel, where one can use expressions to code computation of a cell value from other cells, our environment allows users to write the contents of dependent parts as simple expressions, which may include XPath expressions to refer to other nodes. To help writing path expressions, a novel mechanism is designed to derive XPath expressions automatically by generalizing or refining the paths of elements chosen by users.
- *XML documents as Programs.* When embedded with expressions, an XML document is no longer a pure tree-structured data, and rather, it is a complicated dependency graph, which imposes difficulty in solving dependency [14]. Different from the traditional approaches whose focus is on efficient manipulation of dependency graph, our idea is to consider XML document as a program, and dependency solving (semantic solver) as the process of computing the fixed point of the program. By the lazy evaluation strategy [10], we can guarantee that the fixed point of the program can be obtained in an efficient way if it exists. We shall refer to such document as Programmable Structured Document (PSD).
- *A functional approach to document manipulation.* We show that the functional language Haskell [5] can be very useful for document manipulation in general and for design and implementation of our environment in particular, which has not been well recognized so far. First, the expressions in Haskell are concise and powerful for users to describe computation. Second, the lazy semantics of Haskell programs make them suitable to represent PSDs, and an Haskell compiler can be used by semantic solver to compute fixed points. Third, higher order functions in Haskell can be used to introduce new useful functions (for being used in expressions) to the system without need of other languages.

The remainder of the paper is organized as follows. Section 2 explains PSD, an extension of XML documents with computation. Section 3 gives an overview of the framework of our environment. Section 4 presents the way of resolving the semantics of PSDs. Section 5 provides two supporting mechanisms for this environment. Section 6 discusses some scaling issues. Related work and conclusions are given in Sections 7 and 8, respectively.

2. PSD: AN XML DOCUMENT WITH COMPUTATION

In this section, we explain PSD (programmable structured document), an extension of XML documents with computation for representing dependency relationship. We associate the elements that contain dependent values with a special attribute called `code` showing how the dependent values are generated. To distinguish the `code` attribute used as a PSD keyword, we should qualify it with a specific namespace, while we leave it out here for the ease of presentation. A dependent element is represented in the form of

$$<tag\ \texttt{code} = \text{“}expression\text{”}\ />$$

where element *tag* has empty content. An *expression* may be any Haskell expression [5]. For simplicity, we define it to be a constant value (an integer value or a text string), an XPath expression that refer to the depended elements of the current document, or a function applied to its arguments.

$$\begin{array}{lcl} expression & ::= & constants \\ & | & \texttt{XPath} \\ & | & f\ expression\ \ldots\ expression \end{array}$$

Here the language to describe the path, defined in Figure 2, is a subset of the standard XPath language. This subset is powerful enough to address any part of a tree-structured document and suitable for later automatic derivation. Although it cannot select document fragments as accurate as the full XPath, for example, due to ignoring almost all predicates, this drawback can be compensated by turning to appropriate functions to process the results of path expressions further, as shown in the third case of *expression*. A path can be written in absolute form or relative form. An absolute path is leaded by a slash `/`, while a relative path starts with one or more `..` separated by slash `/`. All path steps are standard except ϵ, which is used for the following two cases:

```
    XPath  ::=  /Steps | Ancestor/Steps
Ancestor   ::=  .. | Ancestor/..
   Steps   ::=  Step | Step/Steps
    Step   ::=  Tag | Tag[Index] | * | ϵ
     Tag   ::=  string
   Index   ::=  integer
```

Figure 2: Syntax of Supported XPath

```
<class>
  <students>
    <student>
      <name>Tom</name>
      <courses>
        <science>81</science><art>87<art>
      </courses>
      <average code="treeavr ../courses" />
    </student>
    ......
  </students>
  <total_students code =
     "childrennum /class/students/*" />
  <qualified_students code =
     "treecount (>=60) /class/students//average" />
</class>
```

Figure 3: A PSD Example

representing `Steps//Steps` by `Steps/`ϵ`/Steps`, and `Steps` by `Steps/`ϵ.

As a quick example for gaining some intuitiveness, consider the following PSD, where * is a function to multiple the integers in two elements specified by `../unitprice` and `../amount`, respectively.

```
<tomato>
   <unitprice>3</unitprice>
   <amount>10</amount>
   <price code = "../unitprice * ../amount"/>
</tomato>
```

An evaluation of this PSD is expected to produce the following result.

```
<tomato>
  <unitprice>3</unitprice>
  <amount>10</amount>
  <price code = "../unitprice * ../amount" >
     30
  </price>
</tomato>
```

Note that computation result can be an XML tree again, which is shown by the following PSD:

```
<tomato>
  <unitprice>3</unitprice>
  <amount>10</amount>
  <price code =
    "elem \"total\" (../unitprice * ../amount)"/>
</tomato>
```

where `elem` makes an XML element from a tag name and an integer content. So computation of the code will yield `<total>30</total>` instead of `30` as before.

Recalling the XML file in Figure 1, we can represent it as a PSD in Figure 3. Three functions `treeavr`, `childrennum` and `treecount` are used in this example: `treeavr` takes an element and returns the average of all integers in it; `childrennum` takes an element and returns the number of its children; `treecount` takes a predicate `(>=60)` and an element, and returns the number of integers in the element

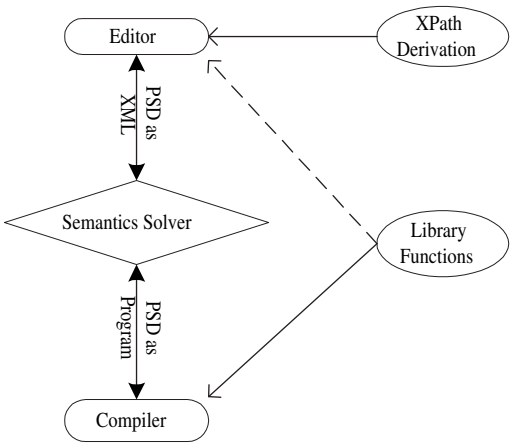

Figure 4: Architecture of the Environment

that satisfy the predicate. All element arguments for these functions are specified by XPath expressions, which refer to the corresponding depended parts.

3. OVERVIEW OF THE ENVIRONMENT

The framework of our environment is depicted in Figure 4. It includes five parts: an editor, a semantics solver, a compiler, a mechanism for XPath derivation and a library of predefined functions.

The editor is the interface to edit documents. This environment has several requirements to such editor. First, it should be able to recognize the special `code` attributes in dependent elements and provide mechanisms to let users evaluate the embedded expressions. Unlike spreadsheets, where a dependent value (always a number) is computed strictly, this environment reifies a dependent element lazily only after a evaluation procedure is explicitly triggered by users. The idea behind this design principle is that because a dependent element probably has complex content structure and its computation might take some time, so we let the user own this freedom to decide whether she/he wants this dependent element is computed. Second, when the value of an expression is wanted, it can ask the semantics solver to evaluate this expression in the context of current PSD. Third, after getting the value of an expression, it can put the dependent value back in the right place with respect to the evaluated expression. In our implementation, the editor is implemented upon xfy [11], a platform for creating and processing XML applications. Figure 5 displays the above class example in the xfy editor, where the expression in element `total_students` has been evaluated by clicking the `evaluate` button beside it. Note that only an element with `code` attribute is displayed with `evaluate` button.

The semantics solver translates PSDs from XML domain into Haskell program domain forwardly or backwardly. In ordinary XML documents, all data are basic values. So their semantics are either simply themselves or gotten by validating against corresponding schemas. However, for PSDs, we have to evaluate the inside expressions in order to determine their meanings. Our approach is to translate PSDs into Hasekll programs, and then compile and run the programs, and at last translate the generated programs back

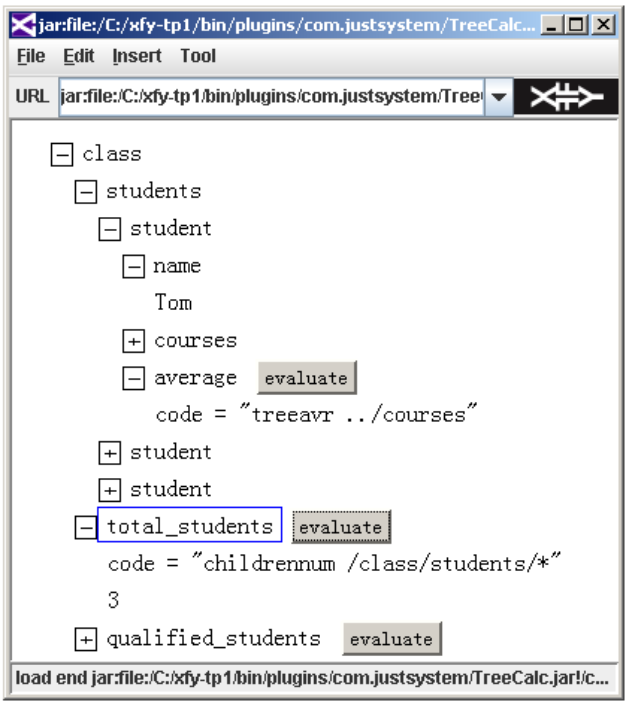

Figure 5: A PSD in xfy Editor

into PSDs as the values of the expressions. That is, our approach uses the semantics of the Haskell programs as that of PSDs by turning to the well established semantics of Haskell programming language. The next section will give a detailed description of semantics solver.

In order to make it easier for users to write expressions, this environment provides two supporting mechanisms. One is a library of functions, and the other is a mechanism for XPath derivation, which can derive a path expression automatically to capture the tree pattern hidden in the elements clicked by users in the editor.

In the rest of this paper, we will concentrate on showing how to implement the semantics solver and the two supporting mechanisms.

4. SEMANTICS SOLVER

The semantics solver is to resolve the semantics of PSDs, that is to resolve what XML documents they finally represent or what they should be evaluated to. In this section, we present the details of how the semantics solver works. Our main idea is to map a PSD to a Haskell program and utilize the lazy evaluation mechanism of Haskell for our semantic solver. More concretely, the semantics solver is built on the HaXML [20], a collection of utilities to process XML documents using Haskell.

4.1 Representing PSD as XML Data

For a *PSD in XML domain*, it is represented by the following data structure, which is borrowed from HaXML [20] and can be gotten by a simple XML parser.

```
data Element = Elem String [(String, String)] [Content]
data Content = CString String | CElem Elemment
```

For data type `Element`, `Elem` is the constructor to create an element from a label, a list of attributes and a list of contents. In our implementation, except for the `code` attribute, the editor implicitly inserts attribute `main` to the element with code to be evaluated due to its neighboring `evaluate` button clicked by users. Data structure `Content` has constructor `CString` to contain a text value and constructor `CElem` to construct a content with a child element. As an example, element `average` in Figure 3 is represented as follows:

```
Elem "average" [("code", "treeavr ../courses")] []
```

Note that the expression is represented as a string and the element content is still empty. To get the semantics of this element under the current PSD context, the expression should be lifted one level up from a string to a computable expression and used as the content of the element. This cannot be done naively by removing the quotation marks of the expression and making it as the element content. This naive way will cause some type errors to be complained due to wrong content type of the element and wrong argument type for function `treeavr`. Moreover, the string XPath expression cannot be evaluated by Haskell, either.

4.2 Data Structures for PSD as Program

For representing a *PSD in program domain*, the data type `Element` remains the same as above, while data type `Content` needs some modifications. First, in the above data structure, only `CString` can contain basic data, that is all basic data in XML files are represented as text strings, which is enough for PSDs in XML domain, but not for PSDs in program domain. In program domain, to evaluate expressions, the basic data should be represented in a correct format. So we need to add more constructors to contain basic data of different types. In this work, for simplicity, we just consider one basic data type `Int`. Second, an expression probably returns a list of elements, so we need another constructor `CElems` to contain the result of such expression.

Therefore, data structure `Content` for PSDs in program domain becomes:

```
data Content = CInt Int | CString String
             | CElem Elemment | CElems [Element]
```

In this data structure, each constructor can contain either a basic value or an expression translated from the value of code attribute. For example, `CInt` in Figure 8 contains the expression translated from string "treeavr ../courses", the value of the code attribute in element `average`. Haskell is a lazy language, so only when the data is really used, for example by pattern matching, the expression under the constructor will be evaluated.

4.3 Recursive Feature of PSD

As shown in Figure 3, expressions in a PSD use XPath expressions to refer to the depended parts, so evaluating these expressions need to evaluate the used XPath expressions first. The semantics of an XPath expression is always determined by its evaluating context [7]. So, in order to evaluate an XPath expression, we need to know its context element, which is the element the path expression will be applied to. Actually, all XPath expressions in a PSD implicitly take this PSD as their context elements (a relative path will be resolved into an absolute one in program domain, so it also uses this PSD as its context element). Since a definition of PSD depends on itself, it is recursive in nature. This

Algorithm tran_elm.

```
Input: elem - an element in XML domain
       path - a list of strings for path steps
       index - an integer for the position of elem
               w.r.t its preceding same name siblings
Output: a string for representing PSD as program
Procedure:
  lbl   = the label of elem
  cnts  = the list of contents in elem
  atts  = the list of attributes in elem
  idx0  = if index == -1 then ""
          else ["++str_of_int( index)++"]"
  path0 = path++[lbl++idx0]
  hd = ("Elem \"")++lbl++("\" ")++str_of_list(atts)
  if elm has code attribute then
    expr = the vale of code attribute
    psd = hd++" ["++tran_exp(expr, path0)++"]"
  else
    lbls = the label list of elements in cnts
    psd = hd++" ["++tran_cnts(cnts, path0, lbls)++"]"
  return psd
```

Algorithm tran_cnts.

```
Input: cnts - a list of contents in XML domain
       path - as that in algorithm tran_elm
       lbls - a string list for element tags in cnts
Output: a string for representing content lists
Procedure:
  pos = 0 and result = ""
  for each cnt in cnts, do
    cm = if result == "" then "" else ", "
    if cnt is an element content then
      cnt has the form CElem elem
      pos = pos + 1
      lbl is the label of elem
      if lbls contains only one lbl then index = -1
      else
        index = the number of lbl in lbls
                before position pos
      elem0 = tran_elm( elem, path, index)
      result  = result++cm++"CElem ("++elem0++")"
    else
       cnt has the form CString str
      if str is a digit string then
        str0 = "CInt "++str
      else
        str0 = ("CString \"")++str++("\"")
      result  = result++cm++str0
  end for
  return result
```

Algorithm tran_exp.

```
Input: expr - a string for an expression
       path - as that in algorithm tran_elm
Output: a string for representing expression
Procedure:
  ty = the type of expr
  if ty == "Int" then constructor = "CInt "
  if ty == "String" then constructor = "CString "
  if ty == "Element" then constructor = "CElem "
  if ty == "[Element]" then constructor = "CElems "
  subexpr = divide expr into a list of strings
         by white space
  newexpr = " "
  for each str in subexpr, do
    if str is a path expression then
      newexpr = newexpr++" "++tran_path(str, path)
    else
      newexpr = newexpr++" "++str
  end for
  return constructor++"("++newexpr++")"
```

Figure 6: Algorithms of Forward Translation

feature is implicit when a PSD is in XML domain and made explicit when in program domain.

Based on the data structure in Section 4.2, a complete PSD in program domain is represented as a fixed point:

$$\texttt{fix}\ \mathit{self}.\ \mathit{doc}$$

where *self* is a variable representing the PSD to be defined, which is used as the context element for evaluating XPath expressions in *doc*. And *doc* is a data built with the constructors of type `Element` and `Content` with the use of free variable *self*. This fixed point can be gotten by solving the equation:

$$\mathit{self} = \mathit{doc}$$

Fortunately, we do not need to solve it by ourselves. Semantics solver just provides this equation and the remaining work is done by the Haskell compiler. If the equation has fixed point, then the Haskell evaluator will guarantee to compute the solution due to its lazy evaluation mechanism [5, 10].

It should be noted that some equation may not have a fixed point. Consider the following *bad* PSD.

```
Elem "badpsd" [] [
  CElem (Elem "dependent"
              [("code", "treesum /badpsd")] [])]
```

Evaluating `treesum '/badpsd'` tries to sum all integers under element `badpsd`, which will in turn incur another evaluation procedure for this expression due to the semantics of `dependent` needs its value. This leads to an infinite loop, so this PSD has no fixed point. In Section 4.7, we will give a conservative condition for the existence of fixed points.

4.4 Forward Translation

As said before, semantics solver needs to prepare the equation for the fixed point. The left-hand side of this equation is always the variable *self*, so the main work is to generate the right side by translating PSD in XML domain.

The algorithms are listed in Figure 6. They generate target programs that will be compiled (by Haskell compiler) and run after the current stage, so they are meta programs [15]. Here, the target programs using data structures in Section 4.2 are represented by strings, which will be output as Haskell source code. In these algorithms, operator ++ is used to concatenate two strings or two lists; when including a quotation mark " or " in a string, it is escaped by a backward slash; `str_of_int` and `str_of_list` convert an integer and a list of string pairs into strings, respectively.

The main algorithm is `tran_elm`. The argument *path* is a list of strings, each of which is a path step from the root element to the parent of element *elem*. The argument *index* indicates the ordinal position of element *elem* with respect to its preceding same label sibling elements. This algorithm translates element *elem* into program domain. If *elem* contains code attribute, then `tran_exp` is used to translate the expression and the result is made as the content of *elem*; otherwise, its whole contents are translated by algorithm `tran_cnts`. In any case, the *path* argument is extended with the label of *elem* and the ordinal index if it is not -1, which means that *elem* has no same name siblings.

Algorithm `tran_cnts` is to translate a list of contents. Argument *lbls* contains a list of labels of all elements in the content list *cnts*, which is used to determined the index of

each element. The translation of string content needs some explanation. If a string consists only of digits, this algorithm uses constructor `CInt` to contain it. In target program, it is an integer since its quotation marks have been removed by string concatenation. For example, "CInt "++"10" result in "CInt 10". Generally, whether an string is an integer should be determined by validating against an XML Schema [16], but we take this way for simplicity.

Algorithm `tran_exp` translates expression *expr* from a string to a type-correct expression recognizable by Haskell compiler. It does two main work: one is to choose an appropriate constructor to contain the translated expression according to the type of expression *expr*; the other is to translate path arguments into HaXML combinators, and thus to be evaluated by Haskell. For the first work, if the expression consists only of a digit string, then it has type "Int" for the same reason as above; if it is an XPath, then it has type "Element"; if it applies a function to some arguments, then it has the same type as the resulting type of the used function; otherwise, it has type "String". For example, `treeavr` has type `Element→Int`, which is a function type with `Element` as the type for argument and `Int` as the type for result, so `CInt` will be chosen to contain expression `treeavr` *element*. To do the second work, algorithm `tran_path` is applied to each path parameter, which will be explained in next section.

4.5 Encoding XPath Expressions

In XML domain, an XPath expression is represented as a string. To interpret them in Haskell, instead of implementing an interpreter from scratch, we turn to the `path` combinator in HaXML [20], which can express XPath query. Our work is to encode XPath expressions with `path` and other combinators.

HaXML combinators `tag`, `elm`, `children`, `position`, `mkElem`, `multi` and `path` are used to encode XPath expressions. They are explained informally as follows.

- `tag` *lbl content*: if the element in *content* has label *lbl*, then returns list [*content*], else returns empty content [].
- `elm` *content*: if *content* contains an element, then returns list [*content*], else returns empty content [].
- `children` *content*: returns the children of the elements in *content* as a content list.
- `position` *n combinator content*: returns a content list consisting of the *n*'th content from the result of applying *combinator* to *content*.
- `mkElem` *label* [*combinator*] *content*: builds a list containing an element content with tag *label* and the content list obtained by applying *combinator* (singleton list [*combinator*] is enough for our purpose) to *content*.
- `multi` *combinator content*: returns a content list obtained by concatenating the result of applying *combinator* to each descendant in *content*.
- `path` *combinator_list content*: returns a content list gotten by applying the first combinator in *combinator_list* to *content* and the next combinator to each content generated by its preceding combinator in *combinator_list*.

Algorithms for encoding XPath expressions are listed in Figure 7. Algorithm `tran_path` encodes XPath expression *pathexp*, and the result is put under an extra label "arg", so the result is always an element. As discussed in Section 4.3, variable *self* is used as the evaluating context of this expression. Since only *self* (indicating the whole document) can be used as the context element, relative path expressions are needed to be resolved into absolute ones. Before processing, *pathexp* is divided into a list of path steps, and if the first step is an empty string, then it is an absolute path; otherwise it is a relative path starting with one or more "..". For example, "../courses" is divided into step list ["..", "courses"], and if argument *path* is ["class", "students", "student[0]", "average"], then the relative path "../courses" will be resolved into ["class", "students", "student[0]", "courses"] before encoded by `tran_steps`.

Algorithm `tran_steps` translates each path step into the corresponding combinator. Empty step "" corresponds to combinator "multi elm" since it is generated due to expression "//"; step "*" corresponds to combinator "children" since it does not care element label; step *lbl* without index is encoded as "children, tag *lbl*", which means that filtering the child elements with name *lbl*; step *lbl*[*index*] is translated into "position *index* (path [*acc*, children, tag *lbl*])", which returns the *index*'th result of the path combinator, which takes as arguments the accumulated encoding result of the preceding steps and the combinator for the current step.

As an example, the expression "/a/b[2]//c[1]/*" is first changed into the step list ["a", "b[2]", "", "c[1]", "*"]. And then it is processed by `tran_steps` as the following recursions:

```
r0: "a"    ==> children, tag "a"
r1: "b[2]" ==> position 2 (path [r0, children, tag "b"])
r2: ""     ==> r1, multi elm
r3: "c[1]" ==> position 1 (path [r2, children, tag "c"])
r4: "*"    ==> r3, children
```

Note that `tran_steps` always inserts the `children` combinator before root element, like that before `tag` "a". So a combinator, `mkElem ''doc'' [elm]`, is inserted in head by algorithm `tran_path` to cancel the effect of this `children` combinator.

Now, we have all knowledge to translate a PSD from XML domain into program domain. For example, PSD in Figure 8 is in Haskell program domain, which is translated from that in Figure 3 by the algorithms in this section.

4.6 Backward Translation

After getting a PSD in program domain, semantic solver will extract and return the content of the dependent element if it contains the expression users want to evaluate, which causes the expression to execute due to the evaluation strategy of Haskell. For example, if `Elem "average" [`*attr*`] [CInt` *exp*`]` contains the concerned expression, then *exp* will be evaluated and the result is returned.

If the result is just a string or an integer, semantic solver can return it simply. However, if it is an element or a list of elements, which probably contains data structures and expressions in program domain, semantic solver has to translate them back into XML domain. Back translation algorithm traverses the whole resulting element or element list and processes each node. Below, we mention only some critical points.

Algorithm tran_path.
Input: *pathexp* - a string for a path expression
path - as that in algorithm tran_elm
Output: a string for encoding *pathexp*
Procedure:

```
steps0 = divide pathexp into a list of strings
         by slash '/'
if the first string in steps0 is "" then
  steps1 = the result of removing the first string
           from steps0
else
  n = the number of leading ".." in steps0
  path0 =the result of removing the first n strings
         from steps0
  path1 = the result of removing the last n strings
         from path
  steps1 = path1++path0
acc = "mkElem \"doc\" [elm]"
combinator = tran_steps(steps1, acc)
result = "(Elem \"arg\" (path ["++combinator
         ++"] (CElem self)))"
return result
```

Algorithm tran_steps.
Input: *steps* - a list of strings for path steps
acc - a string for the accumulated encoding result
Output: a string for combinators
Procedure:

```
if steps is an empty list then
  result = acc
else
  step = the first string in steps
  steps0 = the result of removing the first string
           of steps
  if step == "" then acc1 = acc++", multi elm"
  if step == "*" then acc1 = acc++", children"
  if step is just a label lbl then
    acc1 = acc++", children, tag \""++lbl++"\""
  if step has the form lbl[ index] then
    res  = acc++", children, tag \""++lbl++"\""
    acc1 = "position "++index++ " (path ["
             ++res++"])"
  result = tran_steps(steps0, acc1)
return result
```

Figure 7: Algorithm of XPath Encoding

```
self = Elem "class" [] [
  CElem (Elem "students" [] [
    CElem ( Elem "student" [] [
      CElem (Elem "name" [] [CString "Tom"]),
      CElem (Elem "courses" [][
        CElem (Elem "science" [] [CInt 81]),
        CElem (Elem "art" [] [CInt 87])]),
      CElem (Elem "average" [("code",
        "treeavr ../courses")] [
        CInt (treeavr (Elem "arg"
          (path [mkElem "doc" [elm], position 0
             (path [children, tag "class", children,
              tag "students", children, tag "student"]
             ), children, tag "courses"](CElem self)
          )))]),
    ......
  ]),
  ......
]
```

Figure 8: A PSD as A Program

For the dependent child elements in the result, their contents are not empty because `tran_elm` puts the translated expressions here. However, in XML domain, the translated expressions should be hidden from users. So in backward translation, the contents of dependent elements in the result should be removed.

The constructors `CString` and `CElem` are also existed in XML domain, so they remain the same; `CElems` appears only in the content of dependent elements, so this kind of constructor has been removed after removing the content of the dependent elements; `CInt` can contain an integer valued expression under a dependent element or an integer from the conversion of a digit string. The former case is processed like `CElems`, while the latter case needs to replace `CInt` with `CString` and convert the contained integer back into a string.

4.7 Well-Behaved PSD

As introduced in Section 4.3, a PSD is represented as an equation in program domain, and this equation sometimes does not have fixed point solution because the evaluated expression causes infinite loops. Whether this bad thing happens depends on the behavior of the function used in this expression and its path arguments. For element "badpsd" in Section 4.3, if `treesum` is changed into a constant function, it then becomes a *good* PSD. In this section, we give a conservative condition to avoid such loops.

This condition depends on an extended tree structure for PSD, defined as follows:

DEFINITION 4.1. *A tree structure for a PSD is a triple (N, E, R), where N is a set of nodes, E and R two sets of edges, with the following conditions:*
1) $n \in N$ is an element in the PSD;
2) $E \subseteq N \times N$ and $(n_1, n_2) \in E$ mean that n_1 is the parent element of n_2;
3) $R \subseteq N \times N$ and $(n_1, n_2) \in R$ mean that n_1 refers to n_2 by XPath expressions.

In order to model an PSD in this structure, we have to keep each element names distinct in N. One method is to use XPath expressions to specify elements instead of element tags. As an example, element "badpsd" has the following structure:

```
({0, 1}, {(0, 1)}, {(1, 0)}),
where 0 = /badpsd  1 = /badpsd/dependent
```

The following theorem describes whether a PSD includes expressions that can cause infinite loop evaluations.

THEOREM 4.1. *Suppose a PSD is modelled by structure (N, E, R). If there are no cycles composed by edges in $E \cup R$, then evaluating the expressions in the PSD does not cause infinite loops.*

Note that this condition is conservative since it rules out some PSDs that do not lead to loop evaluations. Back to the above example, the edges (0, 1) and (1, 0) make up a cycle, so it is *conservatively* regarded as a *bad* PSD.

5. SUPPORTING MECHANISMS

In this section, we introduce two supporting mechanisms that make the environment more usable: one is automatic path derivation and the other is a library of predefined functions.

$$\frac{\texttt{Steps}_1 \oplus \texttt{Steps}_2 \Longrightarrow \texttt{Steps}_3}{/\texttt{Steps}_1 \oplus /\texttt{Steps}_2 \Longrightarrow /\texttt{Steps}_3}$$

$$\frac{}{\epsilon \oplus \texttt{Steps} \Longrightarrow \texttt{Steps}}$$

$$\frac{}{\texttt{Steps} \oplus \epsilon \Longrightarrow \texttt{Steps}}$$

$$\frac{\texttt{if Tag}_1 \neq \texttt{Tag}_2 \texttt{ then Tag}_3 = * \texttt{ else Tag}_3 = \texttt{Tag}_1}{\texttt{Tag}_1 \oplus \texttt{Tag}_2 \Longrightarrow \texttt{Tag}_3}$$

$$\frac{\texttt{if Tag}_1 \neq \texttt{Tag}_2 \texttt{ then Tag}_3 = * \texttt{ else Tag}_3 = \texttt{Tag}_1}{\texttt{Tag}_1 \oplus \texttt{Tag}_2[\texttt{Index}] \Longrightarrow \texttt{Tag}_3}$$

$$\frac{\texttt{if Tag}_1 \neq \texttt{Tag}_2 \texttt{ then Tag}_3 = * \texttt{ else Tag}_3 = \texttt{Tag}_2}{\texttt{Tag}_1[\texttt{Index}] \oplus \texttt{Tag}_2 \Longrightarrow \texttt{Tag}_3}$$

$$\frac{\begin{array}{l}\texttt{if Tag}_1 \neq \texttt{Tag}_2 \texttt{ then Tag}_3 = * \\ \texttt{else if Index}_1 \neq \texttt{Index}_2 \texttt{ then Tag}_3 = \texttt{Tag}_1 \texttt{ else Tag}_3[\texttt{Index}_1]\end{array}}{\texttt{Tag}_1[\texttt{Index}_1] \oplus \texttt{Tag}_2[\texttt{Index}_2] \Longrightarrow \texttt{Tag}_3}$$

$$\frac{\begin{array}{l}\texttt{Step}_1 \neq * \;\; \texttt{Step}_1 \neq \epsilon \;\; \texttt{Step}_2 \neq \epsilon \\ \texttt{Step}_1 \oplus \texttt{Step}_2 \Longrightarrow \texttt{Step}_3 \;\; \texttt{Step}_3 \neq * \\ \texttt{Steps}_1 \oplus \texttt{Steps}_2 \Longrightarrow \texttt{Steps}_3\end{array}}{\texttt{Step}_1/\texttt{Steps}_1 \oplus \texttt{Step}_2/\texttt{Steps}_2 \Longrightarrow \texttt{Step}_3/\texttt{Steps}_3}$$

$$\frac{\begin{array}{l}\texttt{Step}_1 \neq * \;\; \texttt{Step}_1 \neq \epsilon \;\; \texttt{Step}_2 \neq \epsilon \\ \texttt{Step}_1 \oplus \texttt{Step}_2 \Longrightarrow * \;\; \texttt{Steps}_1 \approx \texttt{Steps}_2 \\ \texttt{Steps}_1 \oplus \texttt{Steps}_2 \Longrightarrow \texttt{Steps}_3\end{array}}{\texttt{Step}_1/\texttt{Steps}_1 \oplus \texttt{Step}_2/\texttt{Steps}_2 \Longrightarrow */\texttt{Steps}_3}$$

$$\frac{\begin{array}{l}\texttt{Step}_1 \neq * \;\; \texttt{Step}_1 \neq \epsilon \;\; \texttt{Step}_2 \neq \epsilon \\ \texttt{Step}_1 \oplus \texttt{Step}_2 \Longrightarrow * \;\; \texttt{Steps}_1 \not\approx \texttt{Steps}_2 \\ \texttt{let m} = \texttt{length}(\texttt{Step}_1/\texttt{Steps}_1) \texttt{ and n} = \texttt{length}(\texttt{Step}_2/\texttt{Steps}_2) \\ \texttt{if } \exists\, \texttt{i, j.sub}(\texttt{Step}_1/\texttt{Steps}_1, \texttt{i, m}) \approx \texttt{sub}(\texttt{Step}_2/\texttt{Steps}_2, \texttt{j, n}) \\ \texttt{then Steps}_3 = \epsilon/\texttt{sub}(\texttt{Step}_1/\texttt{Steps}_1, \texttt{i, m}) \oplus \texttt{sub}(\texttt{Step}_2/\texttt{Steps}_2, \texttt{j, n}) \\ \texttt{else Steps}_3 = \texttt{Step}_2/\texttt{Steps}_2\end{array}}{\texttt{Step}_1/\texttt{Steps}_1 \oplus \texttt{Step}_2/\texttt{Steps}_2 \Longrightarrow \texttt{Steps}_3}$$

$$\frac{\texttt{Step}_2 \neq \epsilon \;\; \texttt{Steps}_1 \oplus \texttt{Steps}_2 \Longrightarrow \texttt{Steps}_3}{*/\texttt{Steps}_1 \oplus \texttt{Step}_2/\texttt{Steps}_2 \Longrightarrow */\texttt{Steps}_3}$$

$$\frac{\begin{array}{l}\texttt{Step}_2 \neq \epsilon \;\; \texttt{let n} = \texttt{length}(\texttt{Steps}_2) \\ \texttt{if } \exists\, \texttt{i, j.Steps}_1 \approx \texttt{sub}(\texttt{Steps}_2, \texttt{i, j}) \\ \texttt{then Steps}_3 = \epsilon/\texttt{Steps}_1 \oplus \texttt{sub}(\texttt{Steps}_2, \texttt{i, n}) \\ \texttt{else Steps}_3 = \texttt{Steps}_2\end{array}}{\epsilon/\texttt{Steps}_1 \oplus \texttt{Steps}_2 \Longrightarrow \texttt{Steps}_3}$$

Figure 9: Rules for XPath Fusion

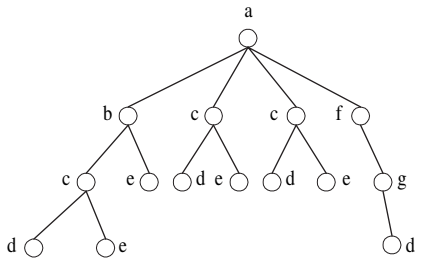

Figure 10: An Example Tree

5.1 Automatic Path Derivation

When inputting an expression, users always need to write XPath expressions to refer to the depended document fragments. In our environment, this work can be done automatically. When users click on some interested elements, an XPath expression that generalizes or refines the path information of these elements will be generated by path derivation mechanism.

This mechanism works like this: the user clicks on the first interested element, and the mechanism returns a path for this element (the Editor is required to provide the path for the selected element, which consists of only step `tag` or `tag` with index since it is for a specific element); and then clicking another element, and at this time instead of returning the path for the second element, the mechanism fuses the first element's path with that of the second element and returns a path that is either more generalized to cover these two elements (in horizontal direction) or more refined by extending the the first path (in vertical direction). For example, on the tree in Figure 10 (also used in later example), path `/a/c/d` is more generalized than `/a/c[0]/d` and `/a//c/d` is more refined than `/a//c`. In general, this mechanism is to fuse a generalized path for history selected elements and that of a newly selected one, and return a more generalized or refined XPath expression.

In this paper, we only introduce how to generate absolute paths. It is easy to translate an absolute path to a relative path by removing some of its leading steps and then adding some .. steps. Our focus here is to design fusion operations, shown in Figure 9, to generate all used path steps in a reasonable fashion. Each operation is defined by a rule with one or more premises above the horizontal line and one conclusion under the line. The judgment has the form $\texttt{Steps}_1 \oplus \texttt{Steps}_2 \Longrightarrow \texttt{Steps}_3$, where $\oplus$ indicates the fusion operation, with arguments $\texttt{Steps}_1$ the path steps for history selected elements and $\texttt{Steps}_2$ the path steps for the newly selected element; $\texttt{Steps}_3$ is the fusion result, which is a more generalized or refined path.

The first rule fuses two absolute paths, and the result is still an absolute path with steps fused from $\texttt{Steps}_1$ and $\texttt{Steps}_2$. The second and the third rule deal with the case where one path is ended while the other not, and they return the remaining steps of the longer path.

The next four rules are used to fuse two tags with or without indices. If the two tags are different, then they are generalized to step `*`, and whether this generalization is valid depends on their following steps, as processed in the ninth and the tenth rules; if the tags are same but with different indices (if they have), then indices are ignored and tag is kept as the generalized step, which is always valid in this mechanism. For example, if user first clicks on node `/a/c[0]`, and then `/a/c[1]`, the generalized path is `/a/c`, that is all nodes `c` under `a` are covered; and for another example that also uses the third rule, node `/a/c[0]/d` and `/a/c[1]` are generalized to `/a/c/d`.

The following three rules deal with the case where $\texttt{Step}_1$ and $\texttt{Step}_2$ are tag or tag with index, not `*` or ϵ. The eighth rule is simple, in which $\texttt{Step}_1$ and $\texttt{Step}_2$ have valid generalized step, so it is put in front of the fusion result of the remaining steps. This rule can generate `*` at last step. For example, paths `a/c[0]/d` and `a/c[1]/e` are fused as `a/c/*`.

In the ninth rule, $\texttt{Step}_1$ and $\texttt{Step}_2$ are fused as a step `*` that is valid under the condition $\texttt{Steps}_1 \approx \texttt{Steps}_2$, which is

defined below. This rule can generate * at the intermediate or last step. As an example, paths a/b/e and a/c[0]/e are fused as a/*/e.

$$
\begin{array}{rcl}
\epsilon & \approx & \epsilon \\
\texttt{Tag} & \approx & \texttt{Tag} \\
\texttt{Tag[Index]} & \approx & \texttt{Tag} \\
\texttt{Tag} & \approx & \texttt{Tag[Index]} \\
\texttt{Tag[Index}_1\texttt{]} & \approx & \texttt{Tag[Index}_2\texttt{]} \\
\texttt{Tag[Index}_2\texttt{]} & \approx & \texttt{Tag[Index}_1\texttt{]} \\
\texttt{Step}_1\texttt{/Steps}_1 & \approx & \texttt{Step}_2\texttt{/Steps}_2 \text{ where } \texttt{Step}_1 \approx \texttt{Step}_2 \\
 & & \text{and } \texttt{Steps}_1 \approx \texttt{Steps}_2
\end{array}
$$

Some of the remaining rules depend on two auxiliary functions: length($Steps$) and sub($Steps, i, j$). Informally, the former returns the numbers of steps in $Steps$ without considering all ϵ in the end, and the latter returns the sub-steps between i and j in $Steps$ with $1 \leq i \leq j \leq$ length($Steps$).

The tenth rule takes place when condition $\texttt{Steps}_1 \approx \texttt{Steps}_2$ in the ninth rule does not hold. This rule checks whether $\texttt{Step}_1/\texttt{Steps}_1$ and $\texttt{Step}_2/\texttt{Steps}_2$ have tail steps that are equivalent. If yes, a ϵ is put in front of the fusion result of their tail steps; otherwise, only the path steps of the newly selected element are returned. That is, when no reasonable generalized steps are available, the path steps of the current element are kept and that for history elements are abandoned. By this design choice, the mechanism always responds a path to user's actions. This rule can generate "//" in paths. For example, if the user first chooses node a/b/c/d, and then a/f/g/d, he can get the generalized path a/ϵ/d (or a//d); however, if the second choice is node a/f/g, then only a/f/g is returned.

The eleventh rule deals with the case where the path steps for history elements start with *. * is surely more general than $\texttt{Step}_2$ since it is either a tag or a tag with index, therefore fusion of them returns in *. This rule can help to construct paths with many * steps. For example, if the user clicks on four elements a/b, a/f, a/b/c[1]/d and a/f/g/d sequentially, then the generalized paths a/*, a/*/c/d and a/*/*/d are produced correspondingly.

The last step is applied when meeting with ϵ in the path steps for history elements. If $\texttt{Steps}_2$ has sub-steps that is equivalent to $\texttt{Steps}_1$, then this sub-steps with the remaining steps in $\texttt{Steps}_2$ will be fused with $\texttt{Steps}_1$ as the new refined steps; otherwise, there is no reasonable refined steps, so $\texttt{Steps}_2$ replaces $\epsilon/\texttt{Steps}_1$ as a response to the user. This rule can refine paths that include "//". For example, paths a/b/c and a/c[1] are fused as a//c, and then fusing a//c with a/b/c/d will produce a refined path a//c/d.

5.2 Library Functions

This environment provides a library of functions, which abstract some common operations on XML documents, such as elements sorting and elements recursive processing. These functions are helpful to reduce the work of users to develop PSDs, and particular useful to those who do not know about programming.

At present, this library includes ten categories of functions: math, text processing, date, searching/sorting, statistics, finance, information, accumulation, higher order and auxiliary functions. Below, we just introduce a few important library functions.

In higher order functions, treefold encapsulates the recursive processing patterns on trees, like foldr on list [5], which can be used to define other functions. For example:

```
treesum elm = treefold (\t -> \l -> sum l)(\x -> 0) id elm
treecount p elm = treefold (\t -> \l -> sum l) (\x -> 0)
                           (\x -> if p x then 1 else 0) elm
```

in which treesum sums all integers under element elm, and treecount has been used in Figure 3; and informally, treefold can be understood as generating an expression gotten by replacing constructor Elem in argument elm with its first argument, CString with its second argument, CInt with its third argument, CElem and CElems with identity functions, and ignoring the attributes of elm.

In searching/sorting functions, treesort elm tag sorts the children of element elm in a descending manner according to the value labelled by tag in each child. For example, expression treesort /class/students/* ``name'' will sort students according to their names for the document in Figure 3. Another interesting function in this category is treelookup elm $\texttt{tag}_1$ p $\texttt{tag}_2$, which returns element $\texttt{tag}_2$ in each child of elm if the value of $\texttt{tag}_1$ satisfies predicate p. For example, treelookup /class/students/* ``average'' (<60) ``name'' queries the names of the students with average scores less than 60.

If the library functions are not enough for users' applications, this environment allows to develop their application-specific functions in a module and then import it into PSDs by preserved tag <psdfun>import $ModuleName$</psdfun>.

6. DISCUSSION OF SCALING ISSUES

In this section, we discuss some scaling issues encountered when applying PSD technique to large documents. The basic idea of PSD works well for small documents, such as the examples in Section 2. For such documents, it takes about 3 seconds to evaluate an embedded expression on a PowerBook G4 computer. We also tested some XML files generated with xmlgen [1] by embedding into them the expressions to compute the average prices of prodcuts at closed auctions. The result is that, when the size of the sample file reaches 60KB, the time for evaluating an expression is about 30 seconds, which is too long for this interactive environment.

In these experiments, cimpiling the generated Haskell programs dominates the whole process of the semantic solver. Hence, in order to make the semantic solver scalable to large documents, it is necessary to reduce the sizes of Haskell programs representing PSDs. Fortunately, this is possible since in a large PSD, an expression generally needs some parts of the document for evaluation. For example, in Figure 3, when computing Tom's average score, the student elements for other students are actually not used. Based on this observation, some of the authors have proposed a techinique to prune PSDs before outputing them to the semantics solver, and it is proved effective in [8].

7. RELATED WORK

Our environment maintains the computation dependency by allowing inserting code in XML documents, which results in PSDs. The concept of PSD has been demonstrated by some ad hoc applications in [19]. While in this work, we give a fundamental and systematic treatment of the syntax and semantics of PSDs. Moreover, this work also proposes two supporting mechanisms to help to develop PSDs. In the

following, we introduce some other works that also extend XML documents with code or script.

Active XML (AXML) documents [2] are the XML documents, where some parts are given by code that calls to Web services. Similarly, the semantics of AXML documents is also dependent on the result of executing embedded code, but the code is specific for calling Web service. Compared with Active XML, PSD allows one element to refer to elements that are in turn computed from other elements in the same document, which is not allowed by Active XML. Moreover, our environment can be used to author documents that need to merge data from Web services or other documents only if the appropriate functions are used.

XEXPR [12] is a scripting language taking XML as its syntax and can be embedded easily in XML documents. It defines some preserved tags for such language constructs as logical and arithmetical operators, loop and conditional control structures, etc. XEXPR scripts can do many useful computation in documents. However, XEXPR does not have the mechanism to refer to other parts of the same documents, so it cannot express the dependency relationship.

8. CONCLUSIONS

We believe that authoring documents with computation dependency is not a pleasant work in existing tools. In this paper, we present a new authoring environment to address this problem. The underlying theory is to extend XML documents with expressions in a clear and systematic way and resolve the semantics of this kind of documents by exploiting the lazy evaluation mechanism of the functional programming language Haskell. In this environment, when meeting with a dependent value, the user just needs to write an expression to represent it. Therefore, the content consistency in this environment can be guaranteed by just checking the expressions, which is much easier than checking the data directly for their consistency. By our experiences of using it, the two supporting mechanisms are quite useful to make it be accepted by end users.

The technologies proposed in this work are orthogonal to those used in other XML authoring tools, so this environment can also be used to enhance the existing tools, such as XMLSpy, which is a work we are exploring.

9. ACKNOWLEDGMENT

Thanks to the PSD project members in the University of Tokyo for stimulating discussion on this work. This work is partially supported by *Comprehensive Development of e-Society Foundation Software* Program of the Ministry of Education, Culture, Sports, Science and Technology, Japan. We are also grateful to the referees for detailed and helpful comments.

10. REFERENCES

[1] XMark-An XML Benchmark Project. http://monetdb.cwi.nl/xml/index.html.

[2] S. Abiteboul, O. Benjelloun, and T. Milo. Positive active XML. In *Proceedings of the twenty-third ACM symposium on Principles of database systems*, 2004.

[3] Altova. XMLSpy. via http://www.altova.com.

[4] V. Benzaken, G. Castagna, and A. Frisch. CDuce: an XML-centric general-purpose language. In *Proceedings of the eighth ACM international conference on Functional programming*, 2003.

[5] R. Bird. *Introduction to Functional Programming using Haskell*. Prentice Hall, 1999.

[6] B. Chidlovskii. A structural advisor for the XML document authoring. In *ACM Symposium of Document Engineering*, 2003.

[7] P. Geneves and J.-Y. Vion-Dury. XPath formal semantics and beyond: A coq-based approach. In *International Conference on Theorem Proving in Higher Order Logics (TPHOLs), Emerging Trends*, 2004.

[8] Y. Hayashi, Z. Hu, M. Takeichi, N. Wake, M. Hara, and N. Oshima. Pruning DOM trees for structured document processing. In *Proceedings of JSSST*, 2004.

[9] Z. Hu, S.-C. Mu, and M. Takeichi. A programmable editor for developing structured documents based on bidirectional transformations. In *Proceedings of the 2004 ACM symposium on Partial evaluation and semantics-based program manipulation*, 2004.

[10] T. Johnsson. Efficient compilation of lazy evaluation. *SIGPLAN Notices*, 19(6):58–69, June 1984.

[11] Justsystem. xfy XML Management Architecture. via http://www.xfytec.com/index.html, 2004.

[12] G. T. Nicol. XEXPR - A Scripting Language for XML. via http://www.w3.org/TR/2000/NOTE-xexpr-20001121/.

[13] Pixware. XMLmind. via http://www.xmlmind.com.

[14] B. Ronen, M. A. Palley, and J. Henry C. Lucas. Spreadsheet analysis and design. *Commun. ACM*, 32(1):84–93, 1989.

[15] T. Sheard. Accomplishments and research challenges in meta-programming. In *Proceedings of the Workshop on Semantics, Applications and Implementation of Program Generation (SAIG'01), volume 2196 of LNCS*, 2001.

[16] J. Simeon and P. Wadler. The essence of XML. In *Proceedings of the 30th ACM symposium on Principles of programming languages*, 2003.

[17] Sun Microsystems. Java Architecture for XML Binding (JAXB). via http://java.sun.com/xml/jaxb.

[18] SyncRO Soft Ltd. oXygen XML Editor. via http://www.oxygenxml.com.

[19] M. Takeichi, Z. Hu, K. Kakehi, Y. Hayashi, S.-C. Mu, and K. Nakano. TreeCalc : towards programmable structured documents. In *Proceedings of JSSST*, 2003.

[20] M. Wallace and C. Runciman. Haskell and XML: Generic combinators or type-based translation? In *Proceedings of the eighth ACM international conference on Functional programming*, 1999.