TreeCalc : Towards Programmable Structured Documents

Masato Takeichi, Zhenjiang Hu, Kazuhiko Kakehi
Yasushi Hayashi, Shin-Cheng Mu, Keisuke Nakano

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

A *programmable structured document* is a structured document with dynamically computable components that can be specified by users in a functional programming language. TreeCalc, a tree version of spreadsheet, is an experimental system demonstrating the notion, basing on the XML viewer and editor, Fungus, developed by Justsystem. TreeCalc takes an XML document representing, for example, a math expression and displays the expression together with the computed result, or a document with tagged chapter and section titles and produces the table of contents of the document. The result of the expression or the table of contents is automatically updated when the user edits the document.

1 Introduction

XML (Extensible Markup Language)[3] has been widely adopted as a standard in describing structured document. Although the popularity of XML owes to its simplicity and wide applicability, the way of looking structured documents as just the data with tree structures leads to the following limitations.

- An XML document is basically first order, in the sense that the element (node) values are of basic types (which do not include function types). This makes things complicated; if we want to introduce new functions, we have to ask for help from other languages. In fact, even with XSLT (XSL Transformations)[4], it is not easy to define complex functions.
- Documents with self-reference can neither be concisely described nor easily manipulated in XML. XLink (XML Linking Language)[6] can be used to refer to some (or whole) part of the document, but there is no means to manipulate it with user-defined functions.

A direct solution for these problems would be a functional extension of XML so that function values may be allowed to appear in the node (for the first problem), and self-reference mechanism through function values may be introduced to describe and manipulate cyclic structures (for the second problem). Particularly, it is possible to provide polytypic higher order functions[8] so that functions on documents can be easily defined. So the main issue to resolve these problems becomes how to manipulate a document with first and higher order functions and self-reference (cyclic structure).

In this paper, we propose the concept of *Programmable Structured Documents (PSD)*, by which we mean the following

- A document may contain computation: a document itself is a program.
- A document can be manipulated by itself: a document becomes a meta program.
- A document keeps type correctness: a document is reliable and reusable.

Note that practically our documents should be programs in a well-ordered form. It would be possible to apply the theory of constructive algorithmics[2] to manipulate the structured documents. Furthermore, *calculation-carrying* mechanism[12] could be introduced to describe transformation in a flexible way. We believe that these extensions will signif-

icantly enhance usability of the structured document.

TreeCalc described in this paper is the first step towards PSD. Here, only function values and self-reference among the features mentioned above are considered. To be concrete, we aim at a tree version of a functional spreadsheet[5], where users can easily create and delete a structure, and express calculation over trees with expressions. To implement this, we embed Haskell to XML, and we make use of an XML editor. When the user edits the documents, the computed parts are automatically updated when necessary. A very trivial inverse computation to bridge the gap between raw data and view is considered.

2 Documents with computations

An XML document represents a tree. For example, the following XML data represents an element `person` having two elements `name` and `pocket` as its children. The latter in turn have three children elements `tool`:

```
<person>
  <name>doraemon</name>
  <pocket>
   <tool>take koputa</tool>
   <tool>dokodemo door</tool>
   <tool>small light</tool>
  </pocket>
</person>
```

In the conventional way an XML document is written, each branches of the tree is written explicitly and their values are independent from each other. In functional languages, on the other hand, it is not uncommon that parts of a structure depends on some other parts. Allowing some parts of an XML document to be dependent on other parts enhances flexibility. Figure 1 shows an article written in XML, whose table of contents is computed from the body of the article. The `hselem` tags acts as place holders for computations to be performed. The article contains three sections. Each `section` element has a `title` element as its first child. The second section is given a name `s2`. The

```
<article> <title> TreeCalc </title>
  <hselem>mkToC root</hselem>
<section>
   <title> Intro. </title>
   ..</section>
<section label="s2">
   <title> Body </title>
   ..</section>
<hselem>capitalise s2</hselem>

<haskell>
mkToC (Article cs) =
     ToC (map mkEntry (zip [1..] titles))
  where titles = map scTitle cs
        scTitle (Section (Title t,_)) = t
        mkEntry (n,t) =
          Item (show n ++ "." ++ t)

capitalise (Section (Title t,cs)) =
  (Section (Title (map toUpper t), cs))

</haskell>
</article>
```

Figure 1: An article whose table of contents is dynamically computed.

third section, merely for the sake of demonstrating, is computed from the second section by converting the title to capital letters. Where the table of contents should be is occupied by a place holder `<hselem>mkToC root</hselem>`. We assume that the root element of the entire document is given the name `root`. The function `mkToC` is defined in the `haskell` element in the end of the document. Written in Haskell notation[1], it scans through the document, collect section names, numbers them by the `zip` function, and constructs a `toc` element from them. After evaluation, it should yield the following:

```
  <toc>
```

[1]Due to space and clarity, the code shown here is a simplification of the real code. In particular, currently we are using generic datatypes, rather than specific data constructors like `Article` to represent a document. It will change in later versions of TreeCalc.

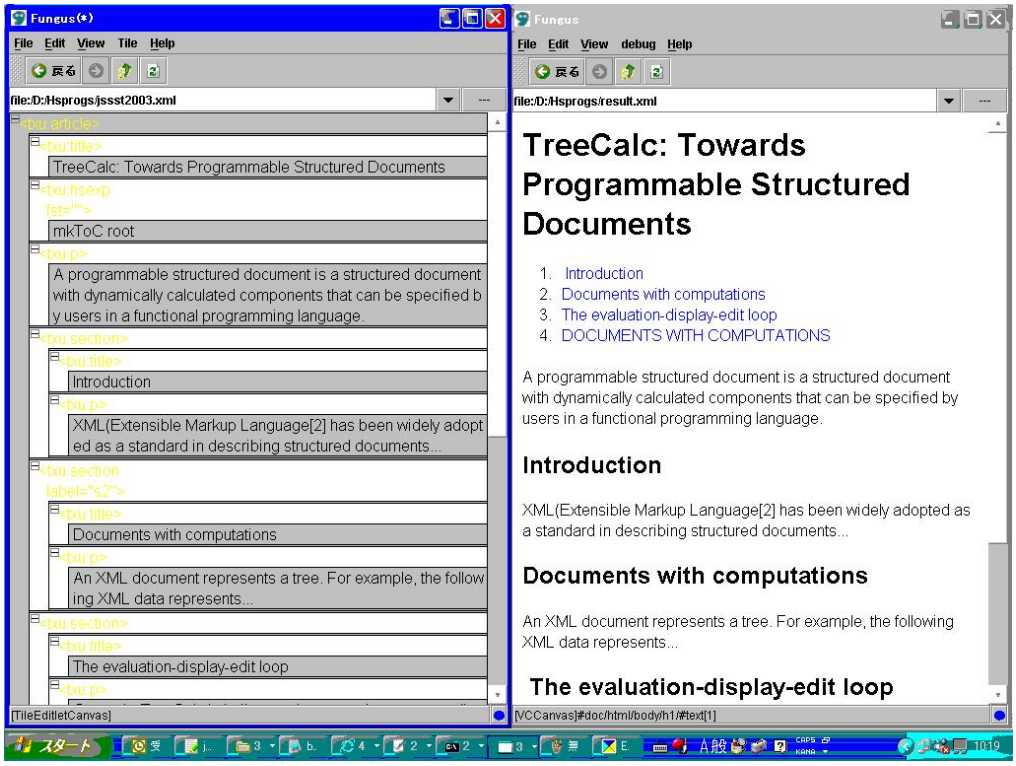

Figure 2: A window for editing and displaying in TreeCalc

```
    <item> Intro. </item>
    <item> Body </item>
    <item> BODY </item>
  </toc>
```

Note that to evaluate the table of contents, `capitalise s2` needs to be evaluated first. This is all taken care of by the data dependency.

In fact, such a mechanism even allows us to construct infinite documents. In the following XML definition, the pocket, labeled `p`, refers to itself as an item in the pocket[2]:

```
<person>
  <name>doraemon</name>
  <pocket label="p">
   <tool>take koputa</tool>
   <tool>dokodemo door</tool>
   <tool>small light</tool>
   <hselem> p </hselem>
  </pocket>
</person>
```

[2]If the DTD allows so, of course. After all, it is a magic pocket!

The tree representing the pocket expands infinitely when being printed. For those who familiar with non-strict functional programming, such a style of definition is not an anomaly but a useful programming technique. The following famous example computes the (infinite) list of Fibonacci numbers, where the first two elements are given, and the rest are constructed recursively from the list itself!

$$fibs = 1 : 1 : zipWith\ (+)\ fibs\ (tail\ fibs)$$

Certainly, eagerly evaluating the list results in non-termination. Therefore, in a non-strict functional language, the parts in a data structure is evaluated only when necessary, e.g., when being printed.

3 The evaluate-display-edit loop

Currently, TreeCalc is built upon the general-purpose application editor, Fungus, developed by Justsystem. The editing process in TreeCalc can be described as an evaluate-display-edit loop. Given an XML document containing computation, the

Haskell evaluator computes the dynamic parts of the document. Currently the evaluator simply eagerly produces the entire document. In the future we will have the evaluator be guided by the displayer such that only the parts being displayed are computed, thereby dealing with possibly infinite documents.

The displaying and editing in taken care of by Fungus,which can present WYSIWYG editing environment for XML documents. In TreeCalc, Fungus displays and edits the XML document like a word processor, as shown in Figure 2.

When the editing actions from the user occurs, Fungus changes the original XML document. The changes then trigger next phase of the loop. The dynamic parts of the document that depend on the modified parts are incrementally updated, and the changes are reflected in the displayed presentation on Fungus.

4 Implementation

The first prototype implementation of TreeCalc is rather simple and preliminary. Currently, an XML document is sent to a macro processor and converted to a Haskell program, which, when being executed, produces the evaluated document. The expressions representing the computation are simply inlined. Both the macro processor and the resulting program makes use of the generic XML datatype and tools defined in HaXml. Due to technical reasons, we are currently using only the generic mechanisms of HaXml.

The advantage of this inlining approach is that we have at our hands all the feature of Haskell, a general purpose programming language. The disadvantage, however, is that Haskell is not a language specifically designed for the job. The programs are more verbose then we had wished, and we are not getting enough help from the type system to ensure the validity of documents.

5 Related works

Quite a few efforts have been devoted into XML processing, transforming and editing. Space here only allows us to mention those by which our work are influenced the most. Editing XML in its raw form is far from user friendly. An XML editor aims at displaying an XML document in a more comprehensible format for the user to edit. Considering the popularity of XML, it is not surprising that there exist quite a number of XML editors. One that is related to the functional programming community is the Proxima project[11]. Since an XML document itself does not specify how it is to be displayed, it is usually specified by a CSS stylesheet or an XSLT script. The designer of Proxima proposed the separation of a *transformation language* and a *presentation language*. The latter describes the layout of a page, while the former specifies how an XML document is transformed to the latter. Our work can be seen as proposing the addition of another layer, a computation language.

HaXml[13] is a library developed for processing XML documents in Haskell, the now standard language for non-strict functional programming. HaXml allows two modes of XML processing. On the one hand, HaXml provides generic datatypes and combinators to parse and transform XML documents. On the other hand, given a DTD, HaXml produces a corresponding Haskell datatype and parsing/printing functions. One can then manipulate an XML document like ordinary Haskell trees, while the type system of Haskell guarantees validity of documents.

Other functional languages specifically designed to process XML documents are also being developed, such as [9, 7, 1]. Usually, an XML document is a supported datatype in the language. Special type systems are adopted to ensure validity of generated document and prevent possible bugs.

6 Conclusion and future work

When implementing the first prototype of TreeCalc, we attempted to reuse existing tools as much as possible. Thus we simply used Haskell as the computation language. Integrating the prototype into Fungus implying that we use an existing transformation and representation language.

This is certainly a temporary solution. One

possible next move is to make use of the type-based translation of HaXml. Another possibility is moving to more domain-specific computation and transformation languages. Lots of lessons can be learnt from previous designs such as XMLambda [9], XDuce[7], or Cduce[1].

After an XML document is transformed to a displayable presentation, TreeCalc expects editing actions from the user and needs to reflect the changes in the original document. One approach to do so is to construct the inverse transform from the presentation back to an calculation-carrying XML document. Certainly, not all computations are invertible. However, it seems that most practical operations involved in XML manipulation are either invertible or can be made invertible. It will be a interesting future topic to identify this subset of invertible computation.

After an editing phase, the XML document needs to be incrementally updated. Existing techniques for incremental updating of attribute grammar ought to be very helpful for our need. In [10], it was described how to convert an XML transforming function to an attribute grammar. We expect to apply similar techniques to TreeCalc.

Acknowledgment

This work has been done under collaboration between University of Tokyo and Justsystem Corporation. The project is supported by the *Comprehensive Development of e-Society Foundation Software* program of the Ministry of Education, Culture, Sports, Science and Technology, Japan.

References

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

[2] R. S. Bird and O. de Moor. *Algebra of Programming*. International Series in Computer Science. Prentice Hall, 1997.

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

[4] J. Clark. XSL Transformations (XSLT) Version 1.0, 16 November 1999. `http://www.w3.org/TR/xslt`.

[5] W. A. C. A. J. de Hoon, L. M. W. J. Rutten, and M. C. J. D. van Eekelen. Implementing a functional spreadsheet in Clean. *Journal of Functional Programming*, 5(3):383–414, 1995.

[6] S. DeRose, M. Eve, and O. David. XML Linking Language (XLink) Version 1.0 , 27 June 2001. `http://www.w3.org/TR/xlink`.

[7] H. Hosoya and B. C. Pierce. XDuce: A typed XML processing language (preliminary report). In G. Vossen and D. Suciu, editors, *Proceedings of Third International Workshop on the Web and Databases (WebDB2000)*, number 1997 in Lecture Notes in Computer Science, pages 226–244. Springer-Verlag, May 2000.

[8] P. Jansson and J. T. Jeuring. PolyP - a polytypic programming language extension. In *Proceedings of the 24rd ACM Symposium on Principles of Programming Languages*, pages 470–482, Paris, France, January 1997. ACM Press.

[9] E. Meijer and M. Shields. XMλ: a functional language for constructing and manipulating XML documents. (Draft), 1999.

[10] K. Nakano. Toward design of XML transformation language intended for stream processing (in Japanese). In *The 20th JSSST Annual Conference*, 2003.

[11] M. M. Schrage and J. T. Jeuring. Proxima: a generic presentation-oriented XML editor. `http://www.cs.uu.nl/research/projects/proxima/`.

[12] M. Takeichi and Z. Hu. Calculation Carrying Programs: How to Code Program Transformations . In *Proceedings of International Sumposium on Principles of Software Evolution (ISPSE 2000)*, pages 250–259, Kanazawa, Japan, 2000. IEEE Press.

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