Calculating Tree Nodes Instead of Table Cells*

-Programming Structured Documents with Generic Functions

Dongxi Liu, Yasushi Hayashi, Zhenjiang Hu, Masato Takeichi
Graduate School of Information Science and Technology, University of Tokyo
{liu,hayashi,hu,takeichi}@mist.i.u-tokyo.ac.jp

TreeCalc is an interactive calculator based on tree nodes instead of table cells. One of its main features is to support higher order generic functions, which provide a powerful and convenient way to define new computations. In this paper, we illustrate the implementation of generic functions for structured documents, explain novel features for calculating tree nodes and demonstrate the ability of TreeCalc by implementing Excel applications in TreeCalc. Calculating tree nodes allows partially evaluated functions and structured values as computation results of nodes, which is contrast to calculating table cells.

1 Introduction

A spreadsheet, like Microsoft Excel [7] and VisiCalc [8], is a program calculating on table cells. In a spreadsheet, the user can specify the cell content by writing formulae, which computes the cell value from other cells and thus maintain dependency among them. It is a widely used computer application. However, with the rapidly proliferation of Web services, the structured documents, like XML documents [1], have become the main data type exchanged in the Internet and these documents have tree structures. The calculation on table cells cannot process this kind of data because when representing them as two dimensional tables some structural information has to be dropped. In fact, table is just a special case of tree, not true vice versa.

In this paper, we define calculation on tree structured data and explain its features and advantages. Although XML is the widely accepted standard format of structured data, it cannot be used straightforwardly in our work because the data in XML documents can only contain the first order values. The expected structured documents for our purpose are Programmable Structured Documents (PSD) [2], which are functional extensions of XML documents. PSD regards computation as the first class value, and supports self-reference, that is, the nodes can refer to the documents being defined.

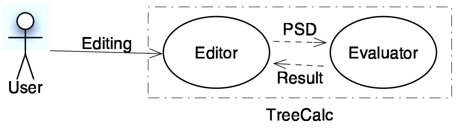


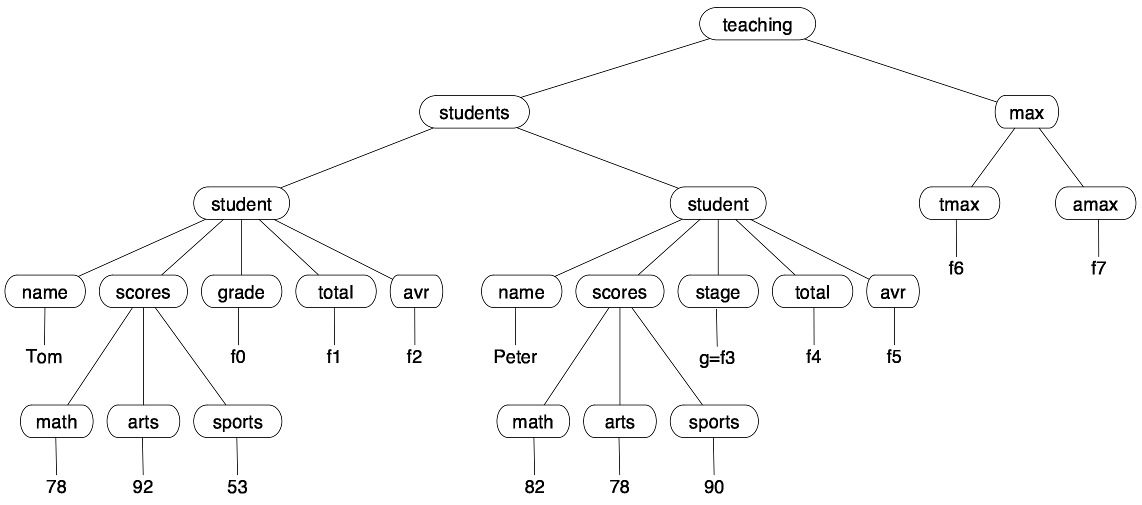
Figure 1: Framework of TreeCalc

Under the framework of PSD, we develop the notion of calculating tree nodes as a tool TreeCalc. TreeCalc includes a built-in higher order generic function named `treefold`, by which other functions on structured documents can be easily defined. In TreeCalc, the computations in nodes can return structured values or partially evaluated functions, while in spreadsheet, only basic values can be returned. In addition, we demonstrate TreeCalc can achieve the core functions of Excel by porting Excel application into TreeCalc, and moreover, it can implement some functions more easily than those implemented using VBA in Excel.

2 Framework of TreeCalc

TreeCalc is an interactive calculator of calculating tree nodes. Figure 1 shows its framework. The

*This project is supported by the *Comprehensive Development of e-Society Foundation Software program* of the Ministry of Education, Culture, Sports, Science and Technology, Japan.



```
f0 = treemap f `/teaching/students/student[0]/scores`
     where f = (\(x::Integer) -> if x > 80 then "Excellent" else if x > 60 then "Good" else "Bad")
f1 = treesum `/teaching/students/student[0]/scores`                f2 = treeavr `/teaching/students/student[0]/scores`
f3 = treefold' `/teaching/students/student[1]/scores` (\t -> \l -> sum l) id
f4 = g id                                                          f5 = div (g id) (g (\(x::Integer) -> 1))
f6 = treemax `/teaching/students/*/total                           f7 = treemax `/teaching/students/*/avr`
```

Figure 2: Students Scores

front end is an XML editor Fungus and the back end is an evaluator. The editor generates PSD and sends it to the evaluator, where the PSD is evaluated as a program. After that, the result is returned to the editor to display. At present, the evaluator is a Haskell interpreter. So from the users perspective, a PSD is an extended XML file, while for the implementer of TreeCalc, it is a Haskell program.

In TreeCalc, a user can edit not only ordinary text values, but also computation expressions. Writing computation expressions includes two aspects, choosing appropriate functions and specifying the arguments for them. The functions are either built-in or user-defined functions. The latter ask users to write Haskell programs based on data structure from HaXML[4], and we don't talk more about this since this paper will focus on built-in generic functions. On the other hand, the interesting function arguments are path expressions, by which the nodes in documents can be referred and then dependency relation among nodes can be represented. The paths are a subset of standard XPath [9] expressions, the syntax of which is as follows:

```
Path ::= /Node1/ Node2/.../Noden
Node :: = Tag | * | Tag[n]
```

where each `Tag` is a string for the name of the node. And now only absolute paths are supported.

A TreeCalc example is shown in Figure 2. In the example, we are interested in the nodes containing computations. Some of the computations have obvious meaning, for example, `treemax '/teaching/students/*/math'`, computes the maximal math score of all students; others are a little complex and will be explained later.

3 Calculating Tree Nodes

Calculating tree nodes means that in a tree some nodes contain computations that always depend on the contents of other nodes. TreeCalc is for demonstrating this concept. In TreeCalc, users can define computations by higher order generic functions, generic in the sense that they can be applied to any well-formed XML documents. These computations can return basic values, structured values or partially evaluated functions, among which the last two kinds are distinct for our work.

3.1 Higher Order Generic Functions

At present, TreeCalc includes only one primitive generic function `treefold` and other generic functions can be defined on it. The underlying data structure (simplified) [4] is as follows:

```
data Element =  Elem String [Content]
data Content =  CElem Element
               |CString CString
```

It is not hard to define the fold function on the above data structure, but the difficulty is that the inconsistency between the ways of treating the data in XML documents by the user and the TreeCalc's internal data structure. From the TreeCalc's editor, the user can edit strings and integers, for example the students' names and their scores, respectively. However, in this data structure, every data has type string (XML schema has more rich basic type [10], but TreeCalc does not supports it so far). Our principle to solve this problem is that if users want to deal with data of type integer, then digit strings are converted to integers at run time and character strings are ignored. On the other hand, if character strings are concerned, then digit strings are dropped. In order to know what data type is being considered, we design type class `View`, in which the method `getVal` returns the value with appropriate type as well as the instance type, that is, the concerned type by users. The definition of `View` class and its two instances are as follows:

```
class View a where
  getVal  :: String -> (a, String)
instance View Integer where
  getVal str = ((read str)::Integer, "Integer")
instance View String where
  getVal str =(str, "String")
```

Now, we can define `treefold` function as in Figure 3, which abstracts the pattern of recursive operations for processing XML documents.

The function `treefold` has a very similar semantics as the fold function on rose trees, like that in textbook [11]. The difference is that `treefold` can selectively deal with the data in an input tree. Given a function `elm` of type `String -> [b] -> c`, a function `celm` of type `c -> b` and a function `lf` of type `a -> b`, the function `treefold elm celm lf` will take as argument an element and return a value of type `c` by replacing the `Elem` constructor with `elm`, the `CElem` constructor with `celm` and for the `CString` constructor, if `a` has type `Integer` and it happens to contain a digit string (judged by predicate `isDig`), then replacing `CString` with `lf`, otherwise ignoring it, in this case it contributes nothing to the final outcome; similar processing if `a` has type `String`.

Using `treefold` we can define other generic functions. For instance, Figure 4 gives some arithmetic operations based on `treefold`, which are helpful to write arithmetic formulae on structured documents.

3.2 Returning Structured Values

Computations in tree nodes can return structured values, while Excel formulae can only return basic ones. The usefulness of returning structured value can be illustrated by `treemap`, which is widely used in TreeCalc applications. `treemap` selectively applies a function to some leaves of the document without changing the structure of the documents.

```
treemap :: (View a, View b, Show b) => (a -> b)
                                 -> Element -> Element
treemap f  = treefold Elem CElem (CString . show . f)
```

We can see that `treemap` is defined by `treefold`. The argument `f` has type `a -> b`, where `a` is an instances of `View` class, so `treemap` can process separately the digit strings or the character strings in documents according to `a`. In Figure 2, node `grade` uses `treemap` to convert each integer score into one of "Excellent", "Good" and "Bad" and generate a new score tree.

Besides using `treemap`, users can also define operations that returns a newly constructed structure. For example, when computing the table of contents for an article from its body, the structure of table of contents is constructed dynamically and different from the structure of article body.

3.3 Partially Evaluated Functions

In TreeCalc, the value of a node can be a partially evaluated function, by which the users can share some static inputs of higher order functions. When programming structured documents, it is very usual to do several different operations on the same nodes.

```
treefold :: (View a) => (String -> [b] -> c) -> (c -> b) -> (a -> b)-> Element -> c
treefold elm celm lf (Elem title contents) = elm title ((map opt . filter choose) contents)
       where opt (CElem e) = (celm . treefold elm celm lf) e
             opt (CString str) = let (v, _) = (getVal str) in lf v
             choose (CElem e) = True
             choose (CString str) = let (v, a) = (getVal str) in let _ = lf v in
                    if a == "Integer" && isDig(str) then True
                    else if a == "String" && not(isDig str) then True else False
```

Figure 3: Definition of treefold

```
treesum :: Element -> Integer
treesum  = treefold (\t -> \l -> sum l) (id) (id::Integer->Integer)
treemax :: Element -> Integer
treemax  = treefold (\t -> \l -> foldr1 max l) (id) (id::Integer->Integer)
treecount :: (Integer -> Bool) -> Element -> Integer
treecount p = treefold (\t -> \l -> sum l) (id) (\x -> if p x then 1 else 0)
treeavr :: Element -> Integer
treeavr e = div (treesum e) (treecount (\x -> True) e)
```

Figure 4: Arithmetic operations defined by treefold

For example, in Figure 2, the summation and average are both done on each student's scores. However, `treefold` has the processed node as its last argument, so we need to change its arguments order so to be specialized with tree nodes. The new function is as follows:

```
treefold' :: (View a) => Element ->(String -> [b] -> c)
                         -> (c -> b) -> (a -> b) -> c
treefold' e elm celm lf = treefold elm celm lf e
```

In order to sum and average the scores for student Peter, we first define a partially evaluated function as follows. We give this function name g, so as to refer to it conveniently.

```
g = treefold' '/teaching/students/student[1]/scores'
  (\t -> \l -> sum l) id
```

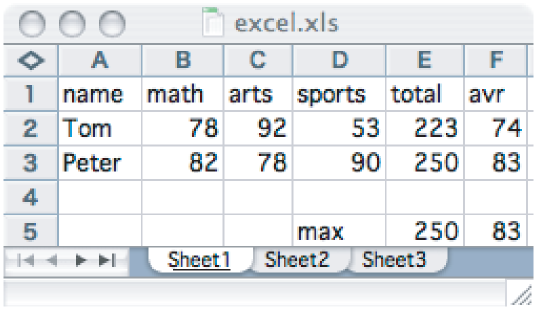
Next, the summation and average can be gotten by applying g to their particular additional arguments, respectively.

```
g id
div (g id) (g (\(x::Integer) -> 1))
```

These codes can be found under the node of student Peter in Figure 2.

4 Excel in TreeCalc

TreeCalc can implement the core functions of Excel. We demonstrate this by giving a procedure to



	A	B	C	D	E	F
1	name	math	arts	sports	total	avr
2	Tom	78	92	53	223	74
3	Peter	82	78	90	250	83
4						
5				max	250	83

Figure 5: Students scores in Excel

port Excel applications into TreeCalc. This work includes two steps: transform tables into trees and change the formulae accordingly. After transforming a table, changing formulae is direct, so we only introduce the first step using a particular example. This example is shown in Figure 5, the Excel version of that in Figure 2. The formulae in cell E2 is "=sum(B2:D2)", similar to F2, E3, F3, E5, F5.

We can transform this table into a tree by the following steps:

1) Determine the root. By the purpose of the example, we can tag the root with `teaching`.

2) Determine the children of `teaching`. The rows (except the row of titles) can be divided into two groups: two rows about students and a row about the maximal scores. So its children can be

`students` and `max`.

3) Determine the children of `students`. Obviously, its children include each `student`.

4) Determine the children for each `student`. According to the kinds of columns, its children can be `name, scores, total` and `average`. And `scores` has children `math, arts` and `sports`.

5) Node `max` will have two children `tmax` and `amax` for maximum of summation and average of each student's scores.

After this procedure, we get the tree structure almost same as that in Figure 2. In addition, TreeCalc can do some work more easily. For example, to sum the discounted prices of goods in a document, we can simply compose `treesum` and `treemap`, but in Excel this have to be implemented by turning to Visual Basic for Applications (VBA).

5 Related Work

TreeCalc has been introduced in [2]. This paper is a continuing work of [2] with two improvements. The first is to give TreeCalc a clear user interface like the description in Section 2, so even the users who know nothing about PSD and the implementation of TreeCalc can use it as a calculator to define their practical applications. The second is to support higher order generic functions, which makes the programming easier. For example, the function to compute the table of contents in [2] can now be implemented more easily using `treefold`.

There are other works to implement spreadsheets using functional language, such as [5], but they still use table cells as computation model. In [3], the authors propose a method to let users define functions in Excel, while still not on tree data model. Proxima[6] supports functions on structured documents, but it seems that only simple arithmetic functions on basic values can be defined.

6 Conclusion

In this paper, we have described TreeCalc to demonstrate calculating tree nodes. It can be seen that calculating tree nodes is more flexible and general, so TreeCalc can implement the core functions of Excel and do works Excel cannot. With the help of higher order generic functions, programming structured documents in TreeCalc becomes much easier, and the end users can use it even without the knowledge of Haskell language. TreeCalc is still under developing. The problems we are considering include, for example, implementing upward or downward accumulations on trees.

Acknowledgment

Thanks Shingo Nishioka, Shin-Cheng Mu, Keisuke Nakano, Kazuhiko Kakehi and Tetsuo Yokoyama for many discussions and comments. This work has been done under collaboration between University of Tokyo and Justsystem Corporation.

References

[1] T. Bray, J. Paoli and C. Sperberg-McQueen. Extensible Markup Language (XML). http://www.3c.org/TR/1998/.

[2] M. Takeichi, et. al. TreeCalc : towards programmable structured documents. In *Proceedings of JSSST*, 2003.

[3] S.P. Jones, A. Blackwell and M. Burnett. A user-centered to functions in Excell. In *Proceedings of the ACM International Conference on Functional Programming*, 2003.

[4] M. Wallace and C. Runciman. Haskell and XML: generic combinators or type-based translation?. In *Proceedings of the ACM International Conference on Functional Programming*, 1999.

[5] W. de Hoon, L. Rutten and M. van Eekelon. Implementing a functional spreadsheet in Clean. *Journal of Functional Programming*, 5(3):383-414, 1995.

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

[7] Microsoft. Excel. http://www.microsoft.com/excel.

[8] D. Bricklin and B. Frankston. VisiCalc. http://danbricklin.com/visicalc.htm.

[9] W3C. XML Path Language (XPath) Version 1.0. http://www.w3.org/TR/xpath.html.

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

[11] R. Bird. *Introduction to Functional Programming Using Haskell.* Prentice Hall, 1998.