Purely Applicative XML Cursor
純作用型XMLカーソル

by

Kazuhiro Inaba

稲葉 一浩

A Senior Thesis

卒業論文

Submitted to

the Department of Information Science

the Faculty of Science, the University of Tokyo

on February 9, 2004

in Partial Fulfillment of the Requirements
for the Degree of Bachelor of Science

Thesis Supervisor: Akihiko Takano　高野 明彦
Title: Professor of Information Science

ABSTRACT

Cursor model is a relatively new approach for XML processing. In this model, a *cursor* acts like a lens that focuses on one node. You can freely move the cursor back and forth in an XML document, and edit the node it indicates. This model can be easily implemented in imperative language like C or Java, by using a pointer to subtree in the XML tree as the cursor. In a fully applicative setting, however, this simple scheme does not work since subtree modification through pointers breaks the principle of referential transparency.
We propose a purely functional data structure named “Slit” to realize a cursor on a tree efficiently in applicative manner. Slit is similar to the zipper data structure introduced by Huet, but has some improvements compared to the zipper in terms of efficiency and expressiveness while handling a tree with variadic child nodes. Using the slit, we implement an XML processing framework based on the cursor model. We also show a generalization of this framework to give an XML view for non XML data.

論文要旨

カーソルモデルは、XML 処理に関する比較的新しい手法の一つである。このモデルでは、「カーソル」を前後に動かすことで XML 文書中の要素を選択し、選択した要素に対する局所的な操作を通して文書の編集を行う。C や Java など一般的な手続き型言語では、このモデルは部分木を指すポインタをカーソルとして扱うことで容易に実現可能である。しかしながら、ポインタによる部分木の書き換えは参照透過性の原理と相容れないため、この簡単な方法によっては完全な作用型言語でカーソルモデルを記述することはできない。

本論文では、作用的な方法でカーソルを効率的に実現するために、関数型データ構造 “Slit” を提案する。Slit は、Huet により提案されたデータ構造 Zipper に、可変個の子要素を持つ木構造を扱う際の効率及び表現力の面で改良を加えたものである。この Slit を用いてカーソルモデルに基づく XML 処理の枠組みの実装を行い、またさらにこの枠組みを一般化することで、単純な変換規則の指定により非 XML データを XML として操作する方法を示す。

Acknowledgements

I would like to thank Professor Akihiko Takano for his invaluable comments and suggestions.

Contents

1 Introduction **1**

2 Cursor Model **3**

2.1 Overview 3

2.2 Advantages compared to other models 4

2.3 Feasibility in applicative setting 5

3 Applicative Cursor on Trees **7**

3.1 The Zipper data structure 7

3.1.1 Zipper for a list and a binary tree 7

3.1.2 Zipper for a labeled variadic arity tree 10

3.1.3 Conclusion 12

3.2 The Slit data structure 13

3.2.1 Slit basics 13

3.2.2 Cursor navigation operations 14

3.2.3 Editing operations 15

3.2.4 Read only slit 16

3.2.5 Slit with dirty flags 16

3.2.6 Conclusion 18

4 Implementation of XML Cursor Framework **20**

4.1 Signature 20

4.2 Cursor for XML data 21

4.3 Cursor for non-XML data 22

4.3.1 Example 1: Cursor for a user defined data structure 22

4.3.2 Example 2: Cursor for a table 23

5 Example **25**

5.1 XML editor shell 25

6 Conclusion and Future Work **27**

A Example of XML Shell Usage **31**

A.1 Input XML 31

A.2 Execution Sample 32

A.3 Output XML 33

A.4 Input CSV 33

A.5 Output CSV 34

Chapter 1

Introduction

Recently, XML [6] has become widely used in many fields of computer industry, especially in the field of the Internet technology. For example, SOAP (Simple Object Access Protocol) and WSDL (Web Services Description Language), which are the data formats based on XML, are the standard protocol for web services. The ontology system RDF (Resource Description Framework) uses XML as its syntax, and is the main technology for the Semantic Web concept. This markup language is commonly adopted as a common structured data storage format as well as the use in the web.

To cover these increasing needs, many XML processing models for programming languages have been developed. The most famous model named DOM (Document Object Model) [7] represents a document as a tree structure, and allows a program to update the content of XML documents dynamically through the manipulation of DOM tree structure. This model is general enought to carry out any operation on XML documents, but it requires some runtime cost to keep the whole tree structure on memory. Another well-known example is the SAX (Simple API for XML) [8]. In this model, parsing events like start and end of XML-elements are reported to the application program through callbacks instead of creating the whole tree structure. Accessing documents through SAX is limited in the sense that it is read-only and sequential, but this model is so simple that a lightweight implementations is possible. Other than those above, several models [9] with intermediate characteristics, such as 'Pull model' or 'Cursor model' have also been designed. They are more flexible than the SAX, and less expensive than the DOM. These new models are not just an theoretical discover, but they have working implementations in a number of languages such as Java, C#, C++, and so on.

Most of those implementations, however, heavily relies on the imperative feature of the target languages. In other words, they are implemented by means of the destructive update of data structures. This implies that there is no simple way to port them to purely applicative settings. On the other hand, commonly used XML representations in functional programming languages are basically just a plain algebraic tree, and in some cases this is less efficient and less intuitive than imperative pointer-based tree of the DOM, especially when it involves the local modification of the tree structure.

In this paper, we show a purely applicative implementation of the cursor model XML processing library which enables us to naturally and efficiently write down the operations with local tree modifications in applicative manner. The new tree manipulation method we developed and used for this library is discussed with special emphasis.

The organization of this paper is as follows. Chapter 2 briefly reviews the cursor model for XML processing, which model we adopt as the design of our framework. Relationship between this model and applicative programming is also discussed. Chapter 3 explains two methods to achieve the purely applicative cursor on tree structure. One is the representation of focused-trees introduced by Huet [1, 2] called “Zipper”. The another is the new data structure we developed, named “Slit”. This makes refinements on the zipper in terms of efficiency and expressiveness while handling a tree with variadic child nodes. XML document trees fall into this type of trees. Chapter 4 shows an implementation of a set of APIs for XML processing based on the slit, and its generalization to handle non-XML data by giving them XML-views. Chapter 5 is an example of utilization of our framework. Chapter 6 discuss them. All source code is written in the Objective Caml system [12], but the code is easily traslatable to any other programming languages whether it is functional or not.

Chapter 2

Cursor Model

2.1 Overview

Cursor model is a relatively new approach for XML processing. In this model, all XML manipulations are done through a *cursor* which points some position in an XML document tree. Set of APIs typically provided in this model is categorized into two types. One is for *cursor navigation*, and the another is for *editing* the content of the document.

Cursor navigation API provides the functionalities to move the position of a cursor in an XML document tree. Four primitive operations which belongs to this category are “move to the first child of current node”, “move to parent node”, “move to following sibling node”, or “move to preceding sibling node” relative to the current position of the cursor.

Editing API provides the functionalities to modify or obtain some information from the target XML document through the cursor. In this category, there are primitive operations which only affects on the element pointed by the cursor, such as “get the name of the element”, “get the list of attributes set to the element”, “rewrite the name of the element”, or “assign an attribute to the element”. Insertion and deletion of XML elements are also supported. For instance, “insert a new node before the cursor”, “insert a new node after the cursor”, or “remove the pointed element from document”.

And on top of these bases, several convenient operations can also be implemented, such as “move to the root of the XML tree” or “move to the first child with specified element name”.

Most recently published XML frameworks tend to be based on the cursor model. XPathNavigator class in Microsoft .NET Framework 1.0 [13] is a read-only instance of the XML cursor, and this class has been announced to have the methods for write access in the coming .NET 2.0. Another example is XMLBeans [14] from Apache Project. Its XMLCursor interface models the XML cursor stated above.

2.2 Advantages compared to other models

The XML cursor model is intended to be a happy medium between the SAX-like ‘push model’ and the DOM-like ‘tree model’. It shares some advantages of those two models and some drawbacks of those.

Just like tree model APIs, cursor model allows us to navigate and manipulate an XML document in any order we want by the basic four moving APIs. This is the difference from SAX-like API in which we have to access the elements in sequential order only, and have no way to modify the original XML document. However, unlike the tree model APIs, cursor model has no need to map all XML elements to some object on memory at one time. This is because all this model has to provide is the access to the ‘current’ element pointed by the cursor. This fact implies that XML cursor APIs have a potential to be more efficient in terms of both time and space than tree model APIs.

Moreover, limiting the access only to local elements around a cursor leads to an important generalization of the framework. Since there is no need to create node objects per each informative item in the XML document, it is more easier to create an XML view on non-XML data without converting whole underlying non-XML data to the XML tree form. Just by supplying a few operations like “move up”, “move next”, “get value”, and “set value” etc., many data structures can be treated by single implementation based on the cursor model APIs without distinguishing the actual data format. For example, we can read address data from an address book stored in a comma separated text (CSV) in the same way from an XML file using the cursor API.

Note that this generalization is not completely general. We can insert an additional element into a person element in XML address book without affecting the other person elements. But it is not possible to insert an additional field into a person data in CSV address book without affecting the others, because we have to keep the number and the order of fields to be same among all entries in one CSV file. So the same cursor operation 'insert' may behave differently depending on the underlying structure. Cursor based data processing is general to the extent that every cursor operations used in the program is common among all target data structure. In this example, read from addressbook and modification of existing fields is generally implementable for both XML and CSV.

Last of all, we claim that one of the advantages of cursor model is its suitability for XML processing in applicative programming languages as discussed in the following section.

2.3 Feasibility in applicative setting

A data structure is called applicative (or functional) if it does not rely on destructive update of the structure for its manipulation. This kind of data structure is useful not only in 'purely functional' programming languages but also in popular imperative languages as a common practice like Immutable Object [10] or as a basic component to assure the Exception Safety [11].

Standard way to represent an XML document in applicative setting is to use an algebraic tree:

```
(* simplified for ease of reading *)
type xml = PCDATA  of string
         | Element of (name * attribute list) * xml list
```

Figure 2.1: Type of XML tree

This representation, however, is awkward in some situations. Three similar operations "find a node with some condition and modify it", "find a node with some condition and remove it”, and “find a node with some condition and duplicate it” cannot share the implementations of “find a node” part efficiently, because the found node with type `xml` have no clue to the XML document tree which the node belongs to.

This problem may be solved by adding a pointer to parent to each node, but without the destructive update, we can not implement a pointer to parent in naive manner. Subtle methods has been developed in this area by Kiselyov [5], but they incur some overheads and limitations. After all, the problem seems to be inevitable if we hold the whole tree in above representation and use the sub node to denote each element.

Let us consider the cursor model instead of the tree model approach. The cursor model does not require to represent the whole document as a tree structure. What it require is only a ‘cursor’ which can go up and down in the tree. So the pointer to parent does not necessarily mean the cycle in the data structure. Therefore, it should be implementable even in purely applicative programming languages. In the next chapter, we describe the method to implement the cursor model in applicative manner in detail.

Chapter 3

Applicative Cursor on Trees

In this chapter, methods to implement an XML cursor are examined. First, we review the zipper data structure introduce by Huet [1]. This is a concise representation for the cursor on algebraic data types, but reveals some drawbacks when applied to a cursor on XML. Though several works to enhance the zipper was done [3, 4], their aim was to make the zipper generic among all algebraic datatypes, and not meant to solve those problems. We propose a new data structure named "Slit" to resolve the shortcomings. This data structure is an improved version of the zipper for a variadic tree.

3.1 The Zipper data structure

Huet proposed a data structure "Zipper" as an efficient and elegant representation for a tree together with a focused subtree. We briefly review this data structure, and discuss whether it is suitable as the cursor on an XML document tree.

3.1.1 Zipper for a list and a binary tree

Normally, a tree data structure is represented as a node which recursively points child trees. The 'root' node is used in order to refer to the whole tree.

```
type int_tree = Leaf of int
              | Node of int_tree * int_tree
```

Figure 3.1: Type of binary trees

In the zipper style, a tree is represented as a pair of a subtree and its one-hole context. The elegant solution to hold the context of a subtree is shown in Figure 3.3. It is a kind of pointer-reversal technique.

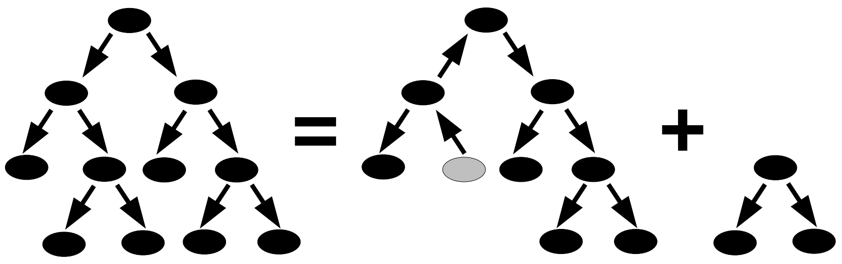

Figure 3.2: Holding a tree as a pair of subtree and its context

```
type it_subtree = int_tree
type it_context = NoPath
                | PathL of it_context *   int_tree
                | PathR of   int_tree * it_context
type it_zipper  = it_context * it_subtree
```

Figure 3.3: Type of zipper for binary trees

There are many possible ways to divide one tree to a subtree and its context. This degree of freedom allows us to "focus" on one part of the whole tree. Operations on the focus such as removing the subtree or replacing with a new subtree can be very efficiently implemented. Operations to move the focus locally in the tree is also simple. See Figure 3.4 and Figure 3.5.

```
(* type: it_zipper -> it_zipper *)
let left = function
  | PathR(t1,p), t2 -> PathL(p,t2), t1
  | _               -> failwith "cannot go left"
let right = function
  | PathL(p,t2), t1 -> PathR(t1,p), t2
  | _               -> failwith "cannot go right"
let up = function
  | PathL(p,t2), t1
  | PathR(t1,p), t2 -> p, Node(t1,t2)
  | _               -> failwith "cannot go up"
let down_left = function
  | p, Node(t1,t2)  -> PathL(p,t2), t1
  | _               -> failwith "cannot go down"
let down_right = function
  | p, Node(t1,t2)  -> PathR(t1,p), t2
  | _               -> failwith "cannot go down"
```

Figure 3.4: Navigation operations on a zipper for a binary tree

```
let remove = function
  | PathL(p,t2), _ -> p, t2
  | PathR(t1,p), _ -> p, t1
  | _              -> failwith "cannot remove the root node"

(* int_tree -> it_zipper -> it_zipper *)
let replace_subtree tr = function
  | p, _ -> p, tr
```

Figure 3.5: Editing operations on a zipper for a binary tree

Zipper approach can be applied for a list data structure as well as for a tree. The zipper for a list is rather simple, in that the context for a sublist becomes a list again.

```
type 'a list    = [] | (::) of 'a * 'a list
type 'a list_sublist = 'a list
type 'a list_context = 'a list
type 'a list_zipper  = 'a list_context * 'a list_sublist

(* moving operations *)
let lz_prev = fun (c::prevs, nexts) -> (prevs, c::nexts)
let lz_next = fun (prevs, c::nexts) -> (c::prevs, nexts)

(* editing operations *)
let lz_get_elem    = fun (prevs, c::nexts) ->  c
let lz_set_elem c  = fun (prevs, _::nexts) -> (prevs, c::nexts)
let lz_remove_elem = fun (prevs, _::nexts) -> (prevs, nexts)
let lz_insert_after  c = fun (prevs,nexts) -> (prevs, c::nexts)
let lz_insert_before c = fun (prevs,nexts) -> (c::prevs, nexts)

(* convert a list_zipper and a list *)
let lz_of_list = fun lst -> ([], lst)
let list_of_lz = fun (prevs, nexts)->List.rev_append prevs nexts
```

Figure 3.6: Operations on a zipper for a list

3.1.2 Zipper for a labeled variadic arity tree

Clearly the "focus" stated in the last subsection works as a cursor on tree or a list structure. To use the focus as a cursor for an XML tree, we need a zipper for the tree which can have arbitrary number of child trees and have a label (the attributes and the name of the XML element) of a node. A code for the "subtree and its context" representation is shown in Figure 3.7. Now, the context has four members - the context of the parent, the label of the parent, the list of preceding siblings, and the list of following siblings.

```
type tree = Leaf of item
          | Node of label * tree list
type tr_subtree = tree
type tr_context = NoPath
                | Path of label
                       * tree list * tr_context * tree list
type tr_zipper  = tr_context * tr_subtree
```

Figure 3.7: Type of a zipper for a labeled variadic arity tree

```
(* type: tr_zipper -> tr_zipper *)
(* case of pattern match failure omitted *)
let prev = fun  (* move the focus to the previous sibling *)
  Path(label,l::ls,p,rs), t -> Path(label,ls,p,t::rs), l
let next = fun  (* move the focus to the next sibling *)
  Path(label,ls,p,r::rs), t -> Path(label,t::ls,p,rs), r
let down = fun  (* down to the first child node *)
  p, Node(label,c::cs)      -> Path(label,[],p,cs), c
```

Figure 3.8: Navigation primitives of a zipper for a tree

Most operations of this zipper are straightforward, but a few have difficulty with their implementation. The operation up is not a constant time operation anymore (even if its amortized cost is constant). It takes linear time with respect to the number of child nodes to reconstruct a node from left siblings, current tree, and right siblings:

```
(* List.rev_append takes linear time *)
let up = fun
  Path(label,ls,p,rs), t ->
        p, Node(label, List.rev_append (c::ls) rs)
```

Figure 3.9: `up` operation on a zipper for a tree

Another problem is that 'remove the focused subtree' operation fails to be an intuitive operation. Figure 3.10 shows one possible implementation of this function. After the removal we choose the next focus to move right, if possible, otherwise left, and up in case of an empty children list.

This complicated rule is an consequence of the fact that the zipper have to focus on some subtree. It cannot focus to an empty children list. This problem also arises in the case of `insert` operation, which is the dual of `remove`. Since it is impossible to go down to an empty children list by the zipper, similar complicated rule takes place in the case of subtree insertion to a node which may or may not have empty children.

```
let remove = function
  | Path(label,t::ls,p,rs), _ -> Path(label,ls,p,rs), t
  | Path(label,ls,p,t::rs), _ -> Path(label,ls,p,rs), t
  | Path(label,[],p,[]),    _ -> p, Node(label,[])
```

Figure 3.10: `remove` operation on a zipper for a tree

3.1.3 Conclusion

As we have seen, the zipper needs to be improved in several ways. The problems of current zipper are summarized as follows:

- Treatment of an empty children list
- Non-constant time `up` operation

In the next section, we will introduce a new data structure and show how to solve these two problems.

3.2 The Slit data structure

3.2.1 Slit basics

The key concept of our "Slit" data structure is simple. We focus on the gap between one subtree and another instead of focusing on a subtree itself. In other word, we replace the zipper style representation of a tree:

$$\texttt{left\_siblings} * \texttt{current\_subtree} * \texttt{right\_siblings} * \texttt{parent}$$

with another form:

$$\texttt{left\_siblings} * \texttt{right\_siblings} * \texttt{parent}$$

The relationship of these two style resembles that of overwrite mode and insert mode in text editors. In the slit style, a cursor resides between two tree nodes as if an insert-mode cursor of text editors exists between two characters, while the zipper style cursor always selects one node as if a overwrite-mode cursor of text editors selects one character. The name "Slit" is derived from its appearance while digging into a tree.

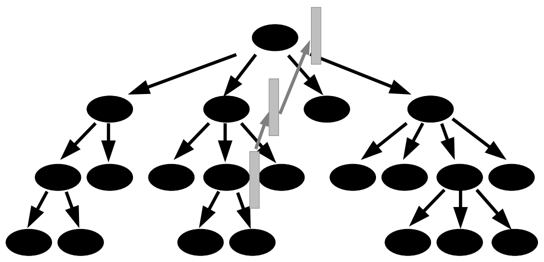

Figure 3.11: Slit for a tree

Straightforward representation of this data type slit is like this:

```
(* slit = root | left_siblings * right_siblings *)
type slit = NoPath
          | Path of tree list * tree list
```

Figure 3.12: Type of slit for a labeled variadic arity tree (first try)

Carefully seeing, you may notice that this representation can be more simplified. The `tree list * tree list` part turns out to be a zipper type for `tree list`. This is not surprising since what that part indicates is the position in the child node list. Additionally, the `NoPath | Path of ...` structure is representable by the standard list structure. Applying these changes, the slit type now becomes much shorter:

```
type slit = (tree list_zipper) list
```

Figure 3.13: Type of slit for a labeled variadic arity tree (revised)

A value which belongs to this type acts as a cursor in a tree. Since this cursor is a composition of list_zipper and list data strucuture, the operations on the cursor is represented as a composition of the operations on a list_zipper and a list as well. We see them in detail in the following subsections.

3.2.2 Cursor navigation operations

Moving a cursor to another gap between its sibling nodes boils down to the operations to move a focus of the zipper for the sibling node list. More specifically, "move the cursor to the next sibling gap in the tree" corresponds to "move the cursor to the next element in the siblings list". Thus, `prev` operation is just an application of the `lz_prev` operation to its current siblings list zipper. So is `next` operation.

Operation for moving the cursor up to parent position and down to its child position has to deal with the tree structure directly, but not so complicated.

```
(* case of pattern match failure omitted *)
let prev = fun hd::tl -> (lz_prev hd)::tl
let next = fun hd::tl -> (lz_next hd)::tl
let down = fun hd::tl ->
  match (lz_get_elem hd) with Node(_, children) ->
    (lz_of_list children)::hd::tl
let up = fun hd::th::tl ->
  match (lz_get_elem th) with Node(label, _) ->
    (lz_set_elem (Node(label, list_of_lz hd)) th)::tl
```

Figure 3.14: Navigation operations for slit (1)

To emphasize that all these `prev` and `next` operations do is a forwarding to `lz_prev` and `lz_next`, we can factor out this "application to current siblings list zipper" as a single function. See Figure 3.15.

```
let apply_to_head lz_op = fun hd::tl -> (lz_op hd)::tl
let prev = apply_to_head lz_prev
let next = apply_to_head lz_next
```

Figure 3.15: Navigation operations for slit (2)

The problem of empty child list is resolved by this representation. The case moving the cursor down to the child position of a node with no children is naturally treated in `down` function by setting both left and right siblings to empty list `[]`,

At this point, the problem of non-constant `up` still remains. The modification for this operation is discussed later.

3.2.3 Editing operations

Unlike zipper, the slit has no direct concept of "current element". So we define the "set or remove the current element" operations to affect on the next sibling element of the cursor.

There is no need to penetrate into the tree data structure for local editing operations required by the cursor model. So all of those are implemented in terms of `apply_to_head` and list_zipper operations.

```
let get_elem        = fun hd::tl -> lz_get_elem hd
let set_elem e      = apply_to_head (lz_set_elem e)
let insert_after  e = apply_to_head (lz_insert_after e)
let insert_before e = apply_to_head (lz_insert_before e)
let remove_elem     = apply_to_head (lz_remove_elem)
```

Figure 3.16: Editing operations for slit

3.2.4 Read only slit

Sometimes, only the reading capability is required for tree manipulation. For such cases, the slit allows us to write a very efficient `up` operation with no extra change to the datatype representation itself.

```
let ro_up = fun hd::tl -> tl  (* List.tl *)
```

Figure 3.17: 'up' operation for read only slit

If no modification was done, moving `up` after `down` causes no effects. And as shown in Figure 3.14, what the `down` operation does is to concatenate a new zipper at the head of the context list. So all we have to do in `up` operation is to remove the head and retain the tail of the context list.

3.2.5 Slit with dirty flags

Even in the case that perfectly read only framework is not acceptable, the optimization technique for read only case can be still partially applied. We add so-called 'dirty flags' to the previous slit data structure.

```
type slit_df = (tree list_zipper * bool) list
```

Figure 3.18: Type of slit with dirty flags

If this flag is true, it shows that somewhere in the current siblings list was modified. The new up operation switches its behavior according to this flag. If it is true, normal non-constant up operation is done. Otherwise, fast ro_up operation is able to be safely applied.

```
let up = fun (hd1,d1)::(hd2,d2)::tl ->
  if d1 then
    match (lz_get_elem hd2) with Node(label, _) ->
      (lz_set_elem (Node(label, list_of_lz hd1)) hd2, true)::tl
  else
    (hd2,d2)::tl
```

Figure 3.19: 'up' of slit with dirty flags

Other operations than up is essentially same as the older version without dirty flags, but it must keep the flag consistent. The strategy for the dirty flag manipulation is simple. Cursor navigation operations do not change the flag, and modifying operations set the flag on.

```
let prev = function (hd,m)::tl -> fun (lz_prev hd, m)::tl
let next = function (hd,m)::tl -> fun (lz_next hd, m)::tl
let down = function (hd,m)::tl ->
  match (lz_get_elem hd) with Node(_, children) ->
    (lz_of_list children,false)::(hd,m)::tl

let get_elem =
  function (hd,m)::tl -> lz_get_elem hd
let set_elem e =
  function (hd,m)::tl -> (lz_set_elem e hd,true)::tl
let insert_after  e =
  function (hd,m)::tl -> (lz_insert_after e hd,true)::tl
let insert_before e =
  function (hd,m)::tl -> (lz_insert_before e hd,true)::tl
let remove_elem =
  function (hd,m)::tl -> (lz_remove_elem hd,true)::tl
```

Figure 3.20: Operations for slit with dirty flags

By introducing this dirty flag check, `up` becomes a constant time operation from linear time one when it is possible. In many use cases like "find an element with specified id and remove it", most elements won't be modified. As a result, most `up` operation is done with the dirty flag off and is performed by the efficient version.

3.2.6 Conclusion

Here's the summary of the features of slit with dirty flags compared to zipper:

- More simple representation achieved through the data structure which is a composition of two simple structures
- Capability to deal with the node with empty child list intuitively.

- When XML modifying operations are rare and cursor moving operations are frequent, the `up` works efficiently.

Chapter 4

Implementation of XML Cursor Framework

4.1 Signature

We start with the definition of the general interface of XML cursor. The data types dealt with this framework is shown in Figure 4.1. Note that there is no recursive `xData list` term which represents child nodes. This is because the cursor model should hide the underlying tree structure and treat them through *cursor moving* operations. In this way, we can implement the cursor API not-only on XML document but also on non-XML data structure.

```
type xName      = string
type xAttribute = xName * string
type xData      = Data of string
                | Node of xName * xAttribute list
```

Figure 4.1: Data types for cursor framework

Currently, our implementation does not support XML namespace. But it is easy to extend this framework to support namespace by changing the type `xName` to hold qualified name. The implementation does not include the comment node and the processing instruction node in its data model.

Signature of operations we define in this framework are shown in Figure 4.2. It consists of operations to move the cursor and to edit the indicated value, as stated in chapter 2.

```
module type CURSOR =
  sig
    (* the type used as an cursor *)
    type cursor

    (* move operations *)
    val prev    : cursor -> cursor
    val next    : cursor -> cursor
    val up      : cursor -> cursor
    val down    : cursor -> cursor

    (* edit operations *)
    val get     : cursor -> xData
    val set     : xData  -> cursor -> cursor
    val insertA : xData  -> cursor -> cursor
    val insertB : xData  -> cursor -> cursor
    val remove  : cursor -> cursor
  end
```

Figure 4.2: Required operations to act as an cursor

Currently, all errors such as “could not go up” are reported by throwing exception.

4.2 Cursor for XML data

It is straightforward to implement the operations of `CURSOR` by using the slit in chapter 3. The `slit` type shown in Figure 3.13 is used as the `cursor` type. Every operation of `CURSOR` corresponds to the counterpart of the type `slit`.

In addition, we implemented two functions for input and output of XML file.

The function `input_xml_cursor` loads an XML document from specified channel and returns a cursor placed at the root position of XML tree. The function `output_xml_cursor` saves the tree indicated by the cursor to specified channel as an XML document. The document tree from the root is saved whereever the cursor is located. Current implementation uses Yaxpo library [15] as the parser of XML texts.

```
val  input_xml_cursor :  in_channel -> cursor
val output_xml_cursor : out_channel -> cursor -> unit
```

Figure 4.3: XML IO operations

4.3 Cursor for non-XML data

The concrete instance of cursor model is not limited to XML documents. If we could define the operations listed in Figure 4.2 for some data structure appropriately, we can manipulate the data structure through cursor API. Hereby a single algorithm written in the cursor API becomes generic so that it can manipulate on multiple kind of data structures.

In this section, we show two examples of the implementation of cursor model operations for non-XML data structures.

4.3.1 Example 1: Cursor for a user defined data structure

Suppose a list of user defined record type which holds an address information of a person. We can define a cursor on this `addressbook` data structure.

```
type person = { name : string; address : string; age : int; }
type addressbook = person list
```

Figure 4.4: Sample user defined data structure

There are more than one possible way to define a cursor on this structure, but the most natural one should be the one shown in Figure 4.5.

```
module CursorForAddressbookType
  struct
    type cursor_lv2 = person list_zipper
    type cursor_lv3 = NameField | AddrField | AgeField | EndMark
    type cursor = LV1 of bool  (* left or right of root? *)
                | LV2 of cursor_lv2
                | LV3 of cursor_lv2 * cursor_lv3
                | LV4 of cursor_lv2 * cursor_lv3 * bool
                  (* bool = left or right of the string? *)
  end
```

Figure 4.5: Cursor for an user defined data structure

Cursor resides in the gap between two substructures as well as the representation of slit for trees. It is straightforward to implement most operations required to be a model of cursor API. But since we cannot insert a new field to a record typed `person` in strongly typed language like OCaml, `insertA` and `insertB` operation for `LV3` cursor must raise an exception. Similar limitation applies to other operations, too.

4.3.2 Example 2: Cursor for a table

Another example is to define a cursor on a abstract data type (ADT), namely a cursor on a data type for which only the operations to be performed on the data are specified, without concern for how the data and the operations are implemented. Here is the sample ADT, 2-dimensional table:

```
module Table :
  sig
    type t
    val entry_num  : t -> int
    val field_num  : t -> int
    val field_name : int -> t -> string
    val get        : int -> int -> t -> string
    val set        : string -> int -> int -> t -> t
    val insert     : int -> t -> t
    val remove     : int -> t -> t
  end
```

Figure 4.6: Table ADT

All manipulation of the table is done through two indices *entry* and *field*, and actual implementation are hidden. Cursor on this ADT consists of the table itself and the index where we focus on. Similar limitation as previous record example also arises when we implement `insertA` operation and so on.

```
type cursor = LV1 of Table.t * bool
            | LV2 of Table.t * int
            | LV3 of Table.t * int * int
            | LV4 of Table.t * int * int * bool
```

Figure 4.7: Cursor for a table

Chapter 5

Example

An example application of our framework is shown in this chapter.

5.1 XML editor shell

We developed a command line XML editor which resembles XSH [16], using our cursor API and its implementation. This editor is a shell-like application which supports:

- The concept of "current working node". User can navigate within an XML document tree by changing the current node using the cursor operations `up`, `down`, etc.
- Modification and display of the current node.
- Deletion and insertion of nodes.
- UNIX-shell-like listing command, `ls` and `pwd`.

All these manipulation commands are implemented through our cursor API. This means that in fact, our *XML*-editor can also handle *non-XML* data structures with cursor API. Example of such data structures are give in chapter 4.

This editor is incomplete at the present moment. Several useful features should be added in the future. We are considering to imlement the commands listed below, and it would not be so hard because of the nature of cursor model.

- Filesystem-like navigation by `cd` command based on XPath notation.
- *Cut-and-paste* of XML nodes.

Sample usage of this editor shell is shown in Appendix A.

Chapter 6

Conclusion and Future Work

We have introduced a purely applicative data structure "Slit" which is an improvement of the zipper data structure for a variadic arity labeled tree. The slit data structure was suitable to represent a cursor on XML documents. By using the slit, we have implemented an XML processing framework.Many extension of our framework should be considered:

- DTD-aware or schema-aware XML manipulation. Current implementation treats XML documents in completely un-typed manner, so the user cannot guarantee the validness of generated result from this framework. It is useful if every edit operation on XML tree automatically keeps the validness with respect to specified schema. This extension may solve the problem that we can not define a general implementation among XML and CSV, stated in chapter 2. An XML tree can have an exact correspondence to table-like structure under restriction by appropriate schemas.

- Multiple cursors on one data structures. Sometimes we want to place two or more cursors in one data structures. This allows us to write a code which copies, moves, or swaps elements in same XML document easily. Of course, it is impossible to naively create two writable cursors which run on one same document keeping the consistency with the principle of referential transparency. But it may be possible to maintain a cursor pair (`cursor * cursor`) for same document, by restricting all operations to affect always on cursor *pair*.

- Cursor for data structures with sharing such as dags. Apart from XML, it is an interesting question if we can define a cursor on dags or graphs efficiently. Cursor based on a dag will have more than one `up` destination. In this style of cursor API, we can implement a cursor view for 2-dimensional table with overlapped hierarchy, i.e. a cursor which can go down in both column-row order and row-column order.

References

[1] Gérard Huet. Functional Pearl: The Zipper, *Journal of Functional Programming*, 7(5):549-554. 1997.

[2] Gérard Huet. Linear Contexts and the Sharing Functor: Techniques for Symbolic Computation. 2002.

[3] Ralf Hinze and Johan Jeuring. Functional Pearl: Weaving a Web, *Journal of Functional Programming*, 11(6):681-689. 2001.

[4] Ralf Hinze and Johan Jeuring. Generic Haskell: applications. 2003.

[5] Oleg Kiselyov. On parent pointers in SXML trees. 2003.

[6] Extensible Markup Language 1.0 Second Edition, W3C Recommendation 6 October 2000. `http://www.w3.org/TR/REC-xml`

[7] Document Object Model Technical Reports.
`http://www.w3.org/DOM/DOMTR`

[8] Simple API for XML. `http://www.saxproject.org/`

[9] Dare Obasanjo. A Survey of APIs and Techniques for Processing XML. 2003.
`http://www.xml.com/pub/a/2003/07/09/xmlapis.html`

[10] Brian Goetz. Java theory and practice: To mutate or not to mutate? *IBM developerWorks Java technology.* 2003.
`http://www-106.ibm.com/developerworks/java/library/j-jtp02183.html`

[11] Herb Sutter. Exception safety issues and techniques, *Exceptional C++: 47 Engineering Puzzles, Programming Problems, and Solutions* 1999.

[12] Xavier Leroy. The Objective Caml system. `http://caml.inria.fr/ocaml/`

[13] Microsoft .NET Framework. `http://msdn.microsoft.com/netframework/`

[14] The Apache XML Project. XML Beans 1.0.
`http://xml.apache.org/xmlbeans/`

[15] Mike Lin. Yaxpo: Yet Another XML Parser for O'Caml.
`http://mikelin.mit.edu/yaxpo/`

[16] Petr Pajas. XSH - XML Editing Shell.
`http://xsh.sourceforge.net/`

Appendix A

Example of XML Shell Usage

A.1 Input XML

```
<Addressbook>
  <Person><Name>Jane</Name>
          <Address>Hongo1-2-3</Address>
          <Birthday>1982/03/06</Birthday></Person>
  <Person><Name>George</Name>
          <Address>Kashiwa4-5-6</Address>
          <Birthday>1999/12/31</Birthday></Person>
  <Person><Name>Michael</Name>
          <Address>Komaba7-8-9</Address>
          <Birthday>1977/04/01</Birthday></Person>
  <Person><Name>Susan</Name>
          <Address>Asano0-1-2</Address>
          <Birthday>1929/08/22</Birthday></Person>
  <Person><Name>David</Name>
          <Address>Sakasai3-4-5</Address>
          <Birthday>1958/04/06</Birthday></Person>
</Addressbook>
```

A.2 Execution Sample

This example demonstrates a manipulation of *address book* file. The same command sequence executes the same modification to both an XML address book and a CSV address book.

```
% editor test.xml
*** welcome to xml_mode ***
>>> pwd
/[Addressbook]
>>> down
>>> ls
[Name]
[Address]
[Birthday]
>>> down
>>> next
>>> down
>>> set Kashiwa4-5-8
>>> up
>>> up
>>> next
>>> next
>>> dig
/[Addressbook]/[Person]
  [Name]
    Michael
  [Address]
    Komaba7-8-9
  [Birthday]
    1977/04/01
>>> remove
>>> exit
```

A.3 Output XML

```
<Addressbook>
  <Person><Name>Jane</Name>
          <Address>Kashiwa4-5-8</Address>
          <Birthday>1982/03/06</Birthday></Person>
  <Person><Name>George</Name>
          <Address>Kashiwa4-5-6</Address>
          <Birthday>1999/12/31</Birthday></Person>
  <Person><Name>Susan</Name>
          <Address>Asano0-1-2</Address>
          <Birthday>1929/08/22</Birthday></Person>
  <Person><Name>David</Name>
          <Address>Sakasai3-4-5</Address>
          <Birthday>1958/04/06</Birthday></Person>
</Addressbook>
```

Reformatting was done by hand for ease of reading. Actual output does not contain any line breaks and indentation spaces.

A.4 Input CSV

```
Name,Address,Birthday
Jane,Hongo1-2-3,1982/03/06
George,Kashiwa4-5-6,1999/12/31
Michael,Komaba7-8-9,1977/04/01
Susan,Asano0-1-2,1929/08/22
David,Sakasai3-4-5,1958/04/06
```

A.5 Output CSV

```
Name,Address,Birthday
Jane,Kashiwa4-5-8,1982/03/06
George,Kashiwa4-5-6,1999/12/31
Susan,Asano0-1-2,1929/08/22
David,Sakasai3-4-5,1958/04/06
```