Encoding XQuery Using *System F*

by

Yun Xia

A thesis
presented to the University of Waterloo
in fulfillment of the
thesis requirement for the degree of
Master of Mathematics
in
Computer Science

Waterloo, Ontario, Canada, 2005

Abstract

Since the World Wide Web Consortium (W3C) has recommended XQuery as the standard XML query language, the interest in using existing relational technology to query the XML data has dramatically increased. The most significant challenge of the relational approach is how to fully support XQuery semantics in XQuery-to-SQL translation. To eliminate the implicit semantics of XQuery, an XQuery fragment must be defined with simple syntax and explicit semantics. XQ [6] is proposed as an XQuery fragment to express XML queries.

In this thesis, XQ is intensively investigated. It is encoded by *System F*, a second-order lambda calculus with a proven expressive power and a strong normalization property. Since XML data is defined as inductive data types, XML tree and XML forest, in *System F*, all basic XML operators in XQ have been successfully encoded. Also, the semantics of XQ are represented in *System F* where XQ's semantics environment is encoded by an *Environment* data type with the corresponding operators. The successful encoding of XQ by *System F* ensures the termination of XQ query evaluation.

Moreover, an extension of XQ by a new tree operator `Xtree` and a vertical Vfor clause is proposed in this thesis to express the *undefinable* XQ queries. It is demonstrated that this extension still allows XQ to retain its XQ-to-SQL translation property that ensures the polynomial evaluation time complexity, and its *System F* encodable property that ensures the termination of query evaluation.

Acknowledgments

I would like to acknowledge the support of several people who have made this thesis possible. First and foremost, I am very grateful to Dr. David Toman, my supervisor, for his great guidance and support during my graduate studies. His patience, valuable suggestions and comments have directly facilitated the completion of this thesis.

Also, I would like to thank Dr. Richard Trefler and Dr. Grant Weddell for reviewing my thesis and providing their valuable advice and comments.

I am particularly grateful to my husband, Xin, for his love and understanding, and to my son, Steven, for the happiness he has brought to my heart, and to my parents for their wisdom and encouragement.

Dedication

To Xin for his love

Contents

Chapter 1

Introduction

Since XML (Extensible Markup Language) has rapidly grown to be the universal format standard for exchanging data among disparate applications on the internet, it is crucial to develop a way to query and interact with XML documents.

1.1 XQuery

"A successful query language can enhance productivity and serves as an unifying influence in the growth of an industry" [9]. As a programming language, which depends on the tree structure of XML data, there are many desirable qualities that an XML query language should exhibit:

- Support the operations on the document order and the *axis* expressions, which are used to navigate the tree structure of XML data, and to retrieve the context of a particular document fragment;

- Provide the ability to express various combinations of multiple XML documents and construct new document based on the query result;

- Offer full compositionality so that the XML query operators can be compose with a full generality; for example, the result of one expression can be used as the input of another expression

As XML has grown in popularity, there has been significant research in the area of XML query languages. As a result, XQuery has been defined and recommended by W3C as the standard of XML query language to support XML applications, such as data processing, transformation, and querying tasks.

As a fully compositional and typed functional language, XQuery provides the ability to flexibly select, recombine, and restructure XML documents and fragments. XQuery is characterized by the following constituents:

- XPath expression traverses and extracts node sequences from document(s);

- FLOWR (for-let-order-where-return) expression binds selected node sequences (in order) to variables and expresses joins and filter conditions;

- Expressions that construct new XML documents or values from the binding variables returned by the FLOWR expressions;

- There is a large library of functions and operators.

To illustrate XQuery, bibliography data is taken from the XML Query Use Cases in Figure 1.1 [1].

```
<bib>
    <book year="1994">
       <title>TCP/IP Illustrated</title>
       <author><last>Stevens</last><first>W.</first></author>
        <publisher>Addison-Wesley</publisher>
        <price>65.95</price>
    </book>
    <book year="1992">
        <title>Adv. Programming in the Unix environment</title>
        <author><last>Stevens</last><first>W.</first></author>
        <publisher>Addison-Wesley</publisher>
        <price>65.95</price>
    </book>
    <book year="1999">
        <title>The Economics of Tech. for Digital TV</title>
        <editor>
               <last>Gerbarg</last><first>Darcy</first>
                <affiliation>CITI</affiliation>
        </editor>
        <publisher>Kluwer Academic Publishers</publisher>
    </book>
</bib>
```

Figure 1.1: Bibliography data

The document contains one `bib` element that contains three `book` elements. This document is well formed because each open element tag has a corresponding closing element tag and the elements are properly nested. A common querying task is to translate the structure and content of XML data, which requires the construction of XML values. Suppose our example is in a file named `bib.xml`, the following expression, utilizing element constructors and FLOWR expressions, queries the titles and years of all books published by Addison-Wesley after 1991 in alphabetical order:

```
<bib>{
    for $b in doc("bib.xml")//book
    where $b/publisher = "Addison-Wesley" and $b/@year > 1991
    order by $b/title
    return
        <book>
            { $b/@year }
            { $b/title }
        </book>
}</bib>
```

Figure 1.2: XQuery example for the XML document in Figure 1.1

In the query expression in Figure 1.2, the `for` clause generates a sequence of `book` elements and binds the variable `$b` to each element; the `where` clause filters the sequence by retaining the books that publisher is "Addison-Wesley" and the published year is later than 1991; the `order by` clause sorts the resulting book sequence; the inner `return` clause returns a sequence of new `book` elements, each containing an attribute `year`, and an element `title`, extracted from the surviving `book` elements in the ordered sequence; the results of the entire FLOWR expression are composed in an

XML element `bib`, whose elements are the sequence returned from the inner `return` clause.

1.2 Challenges

The XQuery language provides such a strong expressive power that XML queries are easy to write, but XQuery complex and redundant syntax prevents an efficient XQuery implementation. Moreover, XQuery has been proven to be Turing-complete, which means the termination of its query evaluation is undecidable. A well-defined subset (fragment) of the XQuery language with a simple syntax and ensuring the query evaluation termination is pivotal to efficient XQuery implementation and optimization. This XQuery *fragment* must describe XML query semantics precisely by defining a data model and a corresponding set of simple operations. Defining the ideal subset or fragment of XQuery has proven to be difficult. Many XQuery subset have been proposed, but have problems with one of these requirements. Currently, there is no universally accepted XQuery subset which holds strong expressive, and meanwhile, ensures its evaluation termination.

1.3 Related Work

There are two key approaches in designing an XQuery fragment: a set-based algebra or a iterator-oriented XQuery fragment. The first approach designs an algebra with operators such as `OP(a, b)`, and the latter approach follows the syntax of XQuery with a format

```
for $i in a, $j in b return (OP $i, $b)
```

for the operator, OP. Although the set-based algebra is easier to reason about algebraically, the iterator-oriented language is generally easier to match to the existing XQuery syntax and capture its semantics.

A proper XQuery fragment is essential for efficient XML query evaluation, but what is right for XML Query processing is still an open issue. There are several good candidates but most of them display shortcomings, such as not recognizing that XML is an ordered data structure, or not ensuring the termination of query evaluation.

The rest of this section discusses about several existing XQuery fragments or XML algebras proposed in XML query research area.

Formal Semantics. The XQuery Core is defined by the W3C XML Query Working Group as part of XQuery Formal Semantics document [5]. As a proper subset of XQuery, the XQuery Core is proposed in the implementation of an XQuery processor called Galax [4]. Here, the evaluation process starts with normalization which translates the XQuery expressions into the XQuery Core syntax. We illustrate the core mapping with a query for the bibliography data in Figure 1.1:
`doc("book.xml")//book[publisher = "Addison-Wesley"]`,
which asks for the books whose publisher is “Addison-Wesley”. This query expression is normalized into the XQuery Core in Figure 1.3.

As an iterator-oriented language, Formal Semantics uses `for` loops (Line 2, Line 4, and Line 7) to rewrite the path expressions to eliminate the im-

```
fs:node-sequence(
  for $fs:dot in
      fs:node-sequence(
        for $fs:dot in doc("bib.xml")
  return descendant-or-self::node())
return
      for $fs:dot in fs:node-sequence(child::book)
      return
      if (some $v1 in fn:data(child::publisher) satisfies
            fn:boolean(op:equal($v1,"Addison-Wesley")))
      then $fs:dot
else ())
```

Figure 1.3: XQuery Core expression for the query doc("book.xml")//book[publisher = "Addison-Wesley"]

plicit semantics of XPath. As a functional programming language, Formal Semantics designs a full semantic mapping for XQuery, and provides a set of logical optimization rules based on the presence of a schema and static type inference. Without type information, its rewriting is not feasible and makes further optimization difficult.

TAX. As a set-based algebra, TAX [8] is a tree-based algebra with a set of operations similar to the relational algebra operations. TAX algebra treats the XML document as a labelled order rooted tree, and decomposes a query into one or more pattern trees, shown in Fig 1.4. Each edge denotes a structural inclusion relationship between the elements, represented by the respective pattern tree nodes. The inclusion relationship can be specified to be either *pc* (parent-child relationship) or of *ad* (ancestor-descendent relationship). In a pattern tree, there is also a boolean predicate, applied to the nodes in the pattern tree that contains the constrain information for the nodes in the pattern tree. For example, the query asks for the books published before 1993 with the publisher "Addison-Wesley", whose XQuery expression and corresponding pattern tree are reflected by Figure 1.4.

```
doc(book.xml)//book[publisher = "Addison-Wesley" AND year < 1993]
```

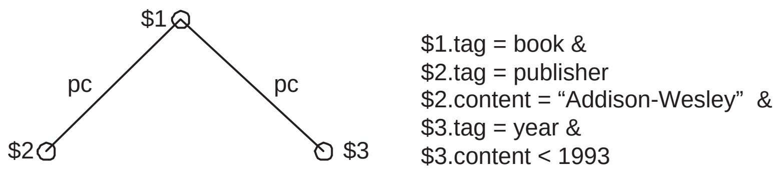

Figure 1.4: Pattern Tree for Query "doc(book.xml)//book[publisher = "Addison-Wesley" AND year < 1993]

The pattern tree expresses a projection operation, $\pi_{\mathcal{P},\ PL}(\mathcal{C})$, which takes the collection, $\mathcal{C}$ (XML document), as an input, and a pattern tree, $\mathcal{P}$, and projection list, PL, as parameters, where a projection list is a list of node labels, appearing in pattern $\mathcal{P}$, which is $1 in Figure 1.4. By using relational algebra as a guide, TAX defines a set of operators such as *selection*, *projection*, *production*, or *grouping*, that takes a collection trees as the input and produces a set of trees as the output. The query evaluation process involves a sequence of pattern tree matchings that breaks up the input tree into small pieces, then reassembles them by grouping and projection. Because of the semantic mismatching of FLWR and TAX, even TAX is efficient in a simple XPath query evaluation, the redundant pattern matching is unavoidable when dealing with complex XQuery queries with a nested structure. An extract effort has to be taken to combine query results from multiple tree pattern queries with extra pattern matching and extra joins.

XQ. DeHaan et al. [6] propose a dynamic interval method that combines the relational mapping implementation with presenting an XQuery fragment XQ[1]. The most significant challenge of the relational approach in XQuery evaluation is how to fully support the XQuery semantics in the XQuery-to-SQL translating process so that the document order is retained. Most relational methods fail to provide this support, and implement only simple XPath query evaluations. The dynamic interval method proposes a novel solution that is based on a XML-specific XQ query language, which captures most of the semantics of XQuery with a fairly simple syntax. As an iterator-oriented language, XQ models an XML database as an ordered forest whose elements are rooted, node-labelled ordered trees. A simple set of operations is defined to manipulate the XML forest. Moreover, XQ keeps the FLWR syntax, which is the most powerful and core part of XQuery, and explicitly expresses its semantics using an *environment* for the variables defined in let and for expressions with their resulting values. For example, the query for the bibliography in Figure 1.1, which returns the books whose publisher is "Addison-Wesley", is expressed by XQ in Figure 1.5.

The dynamic interval method describes the interval representation of XML data and its intermediate query results as relations, which seamlessly preserve the document order required by XQuery semantics. After the XQuery expressions are represented by XQ without losing its semantic meaning, a set of templates that translates the basic operations of XQ into single SQL state-

[1] [6] does not give the algebra a formal name. In the thesis, it is called XQ for convenience.

```
for $a = select('book', subtreedfs(doc(book.xml))) do
let $b = select('publisher', children($a)) in
      $c = select("Addison-Wesley", children($b))
where not empty $c
return $a
```

Figure 1.5: XQ for "doc(book.xml)//book[publisher = "Addison-Wesley"]

ment is proposed based on the interval information of the XML document. Therefore, the XML queries, with the XQ-to-SQL translation, can be evaluated by using an extended SQL engine to achieve the polynomial evaluation time complexity. Even the XQuery translation targets a relational implementation, but its XQ definition can also be used within the native XML database system. The present work in this thesis continues the work of [6], and focuses on the XQ language aspect part.

1.4 Contributions

In the thesis, the theoretical aspects of the XQ query language, including its computability and expressiveness, with respect to its XQ-to-SQL translation property, are investigated. First, XQ is encoded using *System F*. As a second order typed lambda calculus, *System F* has a considerable expressive power [7] and a *strong normalization* property. The successfully encoding of XQ in *System F* indicates that the termination of XP query evaluation, and the decidability of query equivalence. When the expressiveness of XQ are considered, a new tree operator `Xtree` and a vertical `Vfor` clause are proposed to express the *undefinable* XQ queries. In the extension of XQ, the intention is to retain its XQ-to-SQL translation property to achieve polynomial time complexity by utilizing relational query engine, and also its *System F* encodable property to ensure the termination of XQ query evaluation. The main contributions of the thesis are summarized as follows:

- represents XML data in *System F* as inductive data types, XML tree and XML forest, with the corresponding iterators;
- encodes the basic XML operators of XQ by *System F*, and represents the semantic equations of XQ by encoding the semantics environment using an *Environment* data type with a set of operations;
- analyzes the limitations of XQ, and proposes an extension of XQ with a vertical constructor `Xtree` and a Vfor-do clause;
- designs the corresponding XQ-to-SQL templates for the extension in order to utilize the relational engine to achieve a polynomial evaluation time complexity;
- encodes the extended XQ in *System F* to ensure the termination of XQ query evaluation

1.5 Thesis Organization

This chapter has introduced the framework of XQuery implementation by both native and relational approaches. The focus is on the challenges and research work for designing a suitable and simple XQuery fragment. The rest of the thesis is organized as follows. Chapter 2 reviews the background and techniques of the dynamic interval method, including the XQ syntax and semantics. A brief introduction to *System F*, which is a second-order lambda calculus with an expressive power and a strong formalization property, is provided in Chapter 3. Simple data types, such as integer and list, are illustrated to describe the translation schema. Chapter 4 defines the XML document as inductive types in *System F* and represents them as lambda abstractions. The basic XML operators of XQ are encoded in *System F*, and the semantic equations of XQ are represented in *System F* by encoding the semantics environment with a set of operations of an *Environment* data type. Chapter 5 states the expressive limitations of XQ, and proposes an extension with a vertical tree constructor `Xtree` and a new vertical Vfor clause, which is also XQ-to-SQL translatable and *System F* encodable. Chapter 6 summarizes the thesis and suggests future work.

Chapter 2

Dynamic Interval Method

In the thesis, XQ, an XQuery fragment proposed for the dynamic interval method [6] is examined. The dynamic interval method aims to implement XML queries in a relational approach. Its first step is to define the XQuery fragment, XQ, to eliminate redundant syntax and implicit semantics of XQuery. Not only does XQ capture the core features of XQuery with a set of basic XML operators, but also it supports successful XQ-to-SQL translation. Although XQ supports the relational-based XML queries, it can also be used in the native XML query processors. In this chapter, a brief overview of the design concept, assumptions, and techniques, on which the thesis works build, is presented.

2.1 XQ: An XQuery Fragment

As a query language, XQ models an XML database as an ordered forest whose elements are rooted, labelled and ordered trees; also a simple set of operators is presented to manipulate the forest.

2.1.1 XML Data Model and Operations

Definition 1 (XML Forest) An XML document can be described as an ordered forest XForest, and is defined inductively as

$$XForest \overset{\text{def}}{=} [\,] \;\mid\; [\texttt{<s>}\ XForest\ \texttt{</s>}] \;\mid XForest\ \texttt{@}\ XForest$$

where s is a string, [ ] denotes an empty forest, [<s> *XForest* </s>] signifies a forest containing a single tree element with a root node labelled s and an ordered forest XForest as its children, and *XForest* @ *XForest* indicates a new forest concatenated by two forests.

The XML forest describes an XML document without distinguishing between node identity and node types (element, attribute, and text). However, such features can be easily added by additional encoding conventions that relate either to node labelling or to subtree patterns: the text leaf node with CDATA text can be encoded by labelling the node with “text:”; an element eleName is encoded by using the label “<eleName>”. An attribute of an element can be treated as a subtree of this element, and is labelled “attribute:”, where as the attribute value is encoded as subtree of its attribute node and labelled as “text:”. For example, the XML document in Figure 1.1 can be encoded with node type information to distinguish the element, attribute, and text nodes. Figure 2.1 depicts part of the converted XML data.

Since XML data is described as an ordered forest, *XForest*, XQ provides a set of basic operations to manipulate the forest data model so that it can navigate the structure of *XForest* data, retrieving the context of particular

```
<bib>
    <book>
      <attribute:year> <text:1994> </attribute:year>
      <title> <text:TCP/IP Illustrated/></title>
      <author>
         <last> <text:Stevens/> </last>
         <first> <text:W./> </first>
      </author>
      <publisher> <text:Addison-Wesley/> </publisher>
      <price> <text:65.95/> </price>
    </book>
    ...
</bib>
```

Figure 2.1: Converted bibliography data with encoded node information

XForest fragments, and joining multiple XForest forests to construct a new *XForest* document.

Constructors. *XForest* data (XML forest) can be empty (no tree element). Also, it can be constructed from a concatenation of two XML forests, or from an element constructor that adds a labelled root to an XML forest. The constructors and corresponding types [1]are described as follows:

empty constructor [ ] : *XForest*, constructs an XML forest without elements;

element constructor $Xnode : String \rightarrow XForest \rightarrow XForest$, which takes a string as a root label and an XML forest as children to

[1]In the thesis, curried functions is used instead of uncurried functions presented in [6]. The curried function f of type $int \rightarrow int$ means that the function application $f\ a$ with an integer argument a returns a result of integer.

construct an XML forest with a single tree node[2];

concatenation `@` [3]: *XForest* → *XForest* → *XForest*, takes two XML forests as arguments to construct a new XML forest by appending the second argument to the first one, keeping the inner order of each argument

Horizontal Operators. XQ defines a set of horizontal operators that manipulate the *XForest* data (XML forest) at the top element (tree) level without recursively traversal into the structure of the trees.

head : *XForest* → *XForest*, returns the first element tree from a forest. Strictly, the return value should be an XML tree instead of an XML forest. As a result, a new type *XTree* (XML Tree), in Chapter 4, is added;

tail : *XForest* → *XForest*, takes an XML forest, and returns a new forest without the first element. The return forest still keeps the original order of the input forest;

reverse : *XForest* → *XForest*, reverses the element order in an XML forest and keeps the original structure of each tree element unchanged;

select : *String* → *XForest* → *XForest*, takes a string, s, and an XML forest as arguments, and returns sub-forests whose tree elements

[2]In the thesis, the element constructor is defined to be a tree constructor that takes a labelled node and a forest as arguments and returns a tree

[3]In the thesis, another constructor, concat, which takes a tree and a forest to construct a new forest, is adopted instead of concatenation

have the root node labelled s. For example,

```
select('bib', doc(bib.xml))
```

returns the element trees whose root labels "`bib`" exists; otherwise, an empty forest is returned;

distinct : *XForest* $\rightarrow$ *XForest*, filters out duplicate elements in an XML forest. Although `distinct` is defined as a horizontal operator, it still needs an auxiliary boolean function equal that recursively compares two trees' structural relationships;

sort : *XForest* $\rightarrow$ *XForest*, takes an XML forest and returns a sorted forest. The sort operator also needs an auxiliary boolean function less to recursively compare two trees' structural order

Vertical Operators. In order to navigate the structure of each tree element of the forest, XQ provides vertical operations such as children and subtreedfs to locate the desired sub-element nodes.

roots : *XForest* $\rightarrow$ *XForest*, returns all the tree elements of a forest at the top level;

children : *XForest* $\rightarrow$ *XForest*, acting as path axis "/", selects children nodes and returns them in the original (document) order;

subtreedfs : *XForest* $\rightarrow$ *XForest*, acting as path axis "//" (descendent-or-self), selects self and sub-elements, and returns them in an DFS (document) order as an XML forest

Boolean Operators. XQ provides two binary boolean operators, `equal` and `less`, to compare the structural relation of trees. [4]

2.1.2 Syntax of XQ

With denoting XML data as an XML forest, dynamic interval method describes the XML query in a simple syntax, with which explicit or implicit semantics of an XQuery expression can still be captured.

Definition 2 (XQ) An XML query expression can be expressed by the following BNF rules:

$$\begin{array}{rcl} e & ::= & x \\ & | & \mathsf{XF_n}(e_1,\ \ldots,\ e_n) \\ & | & \mathsf{let}\ x = e_1 \mathsf{in}\ e_2 \\ & | & \mathsf{where}\ \varphi\ \mathsf{return}\ e \\ & | & \mathsf{for}\ x\ \in e\ \mathsf{do}\ e' \end{array}$$

where e, e_1, ..., and e_n are query expressions, $\mathsf{XF_n}$ is a basic operator defined in the previous section, x is a variable, and φ is a boolean condition.

2.1.3 Semantics of XQ

The semantics of XML queries are expressed by denotational semantics style equations. The denotational semantics introduces the concept of an *Environment*, which maps each free variable of an expression to its value. Since the behaviour of a query evaluation is a computation from an initial state to

[4][6] denotes equal and less as XML forest operators. Since we intend to compare two trees' structural relationship, they are denoted as tree operators in Chapter 4.

a final state through a sequence of intermediate states (results), the evaluation denotes as a *transformation* function $[\![-]\!]$, where $E' = [\![e]\!]E$ means the new state E', an instance of *Environment*, is transformed from state E by the computation of the expression e. Therefore, the semantics of FLWR-like expressions is described as semantic equations as follows:

SEM EQ: Variables

$$[\![x]\!]E \overset{\text{def}}{=} E\ (x)$$

The role of variable x is to obtain the value (XML forest) of variable x from *Environment* E without changing E;

SEM EQ: Let-Assignment

$$[\![\mathsf{let}\ x = e\mathsf{in}\ e'\ ]\!]E \quad \overset{\text{def}}{=} \quad [\![e']\!](E[x = ([\![e]\!]E)])$$

Expression e' is evaluated in E', which is a new *Environment* extended from E by binding the variable x to the result of e;

SEM EQ: Where-Return

$$[\![\mathtt{where}\ \varphi\ \mathsf{return}\ e]\!]E \quad \overset{\text{def}}{=} \quad \mathsf{if}\ ([\![\varphi]\!]E)\ \mathsf{then}\ ([\![e]\!]E)\ \mathsf{else}\ []$$

where φ is a boolean condition. If condition φ holds, expression e is evaluated in *Environment* E;

SEM EQ: For-Do

$$\begin{aligned}[\![\mathsf{for}\ x\ \in\ e\ \mathsf{do}\ e']\!]E \quad \overset{\text{def}}{=} \quad & [\![e']\!](E[x := v_1])\ @ \ldots @\ [\![e']\!]E([x := v_k]) \\ & \text{where } [v_1,\ \ldots,\ v_k] = [\![e]\!]E\end{aligned}$$

The expression $\mathsf{for}\ x\ \in e\ \mathsf{do}\ e'$ first evaluates expression e in *Environment* E and obtains a list of results $[v_1,\ \ldots,\ v_k]$; and then uses assignment operation "$x\in$" to transform *Environment* E into a list of new *Environments* $[E_1,\ \ldots,\ E_k]$. Finally expression e' is computed in these new environments, and the results are concatenated.

SEM EQ: $\mathtt{XF_n}$

$$[\![\mathtt{XF_n}(e_1,\ldots,e_k)]\!]E \ \stackrel{\text{def}}{=}\ \mathtt{XF_n}\ ([\![e_1]\!]E,\ldots,[\![e_k]\!]E)$$

where $\mathtt{XF_n}$ is an XML operator defined in Section 2.1.1, such as `select`, `subtreesdfs`, with k arguments. The k arguments are evaluated simultaneously in *Environment* E, and the results are fed to the function $\mathtt{XF_n}$.

2.1.4 XQ Translation of XQuery

As an XQuery fragment, XQ captures all the detailed intricate nuances presented in the XQuery's FLWR and XPath expressions. Its syntax allows an arbitrary composition of the basic function invocations, local variable definitions, filtering by boolean conditions, and iteration over tree elements in XML forests [6]. Some XQueries are used for the bibliography data in Figure1.1 to illustrate the XQ translation.

Example 2.1.1 List the books published by Addison-Wesley, including their year and title.
XQuery expression:

```
<bib>
 {
  for $b in doc("bib.xml")/bib/book
  where $b/publisher = "Addison-Wesley"
  return
    <book year="{ $b/@year }">
     { $b/title }
    </book>
 }
</bib>
```

corresponding XQ expression:

```
for $b in select(book, children(select(bib,
                                       children(doc(bib.xml))))) do
where not empty (select(Addison-Wesley, children(publisher,
                                                 children($b))))
return
  let $3 =
    let $1 = children($b)
    in (let $2 = Xnode(year, (select(year, children($1)))
             @ Xnode(title, (select(title, $1)))
        in Xnode(book, $2))
  in Xnode(bib, $3)
```

Example 2.1.2 Create a flat list of all the title-author pairs with each pair enclosed in a "result" element.
XQuery expression:

```
<results>
  {for $b in doc(bib.xml)/bib/book,
        $t in $b/title,
        $a in $b/author
    return
        <result>
            {$t}
            {$a}
        </result>}
</results>
```

corresponding XQ expression:

```
let $2 = for $b in select(book, children(select(bib,
                   children(doc(bib.xml)))))
         do let $1 = children($b) in
            Xnode(result, (select(title, $1) @ select(author, $1)))
in Xnode(results, $2)
```

2.2 Dynamic Interval Relational Method

2.2.1 Interval Encoding

In order to manipulate XML documents in relational database, an interval-encoding method is proposed to capture enough information about XML data by encoding an XML document as triples (s, l, r), where s is the string of the node name (node s), l is the left endpoint of node s, and r is the right endpoint of node s. The way to generate the interval encoding is to perform a depth-first traversal of the XML tree by using a counter to mark the l value of the node when it is visited first time, and to mark its r value when the node is visited the last time. The interval encoding relation of bibliography XML data in Figure 1.1 is shown in Figure 2.2.

2.2.2 Dynamic Intervals

To evaluate an FLWR expression by a fixed relational query, a *dynamic* interval encoding is presented to represent the sequence of environments generated within an query evaluation process. A detailed definition of dynamic interval can be found in [6].

Example 2.2.1 Figure 2.3 shows the query results for the path expression,

s	l	r	s	l	r	s	l	r
bib	0	79	book	25	47	book	52	78
book	1	24	year	26	29	year	53	56
year	2	5	1992	27	28	1999	54	55
1994	3	4	title	30	33	title	57	60
title	6	9	Adv. Prog.	31	32	The Economics of	58	59
TCP/IP Illu.	7	8	author	34	43	editor	61	47
author	10	19	last	35	38	last	62	65
last	11	14	Stevens	36	37	Gerbarg	63	64
Stevens	12	13	first	39	42	first	66	69
first	15	18	W.	40	41	Darcy	67	68
W.	16	17	publisher	44	47	affiliation	70	73
author	10	19	Addison-Wesley	45	46	CITI	71	72
price	20	23	price	48	51	publisher	74	77
65.95	21	22	65.95	49	50	Kluwer Academic	75	76

Figure 2.2: encoded Bibliography data

T_{title} = `doc(bib.cml)/bib/book/title`, with the index relation I, representing the initial environment with $i = 0$.

2.2.3 XQ-to-SQL Translation

Based on the semantics of XQ expressions, SQL translation templates, including the templates for basic operations and those for FLWR patterns, are adopted to translate an XQ-syntax query expression into a relational query statement by composing SQL templates from the sub-query fragments. The detailed XQ-to-SQL translation can be found in [6].

Proposition 1 For each XQ expression, there is a corresponding SQL query statement, such that all XQ queries can be implemented in the relational

I
i
0

T_{title}		
s	l	r
title	6	9
TCP/IP Illustrate	7	8
title	30	33
Advanced Programming in the Unix environment	31	32
title	57	60
The Economics of Technology and Content for Digital TV	58	59

Figure 2.3: Titles in an initial environment

query engine.

By defining a simple XQuery fragment, XQ, to capture the core features of XQuery, and a set of XQ-to-SQL translation templates, the dynamic intervals method successfully evaluates the XML queries using the relational query engine, which ensures the evaluation polynomial time complexity.

Chapter 3

System F

In the thesis, *System F*, a second order lambda calculus, is adopted to represent XQ presented in [6]. In this chapter, we introduce *System F*, whose generality and strong expressive power supports faithful encoding of arbitrary composition of XQ expressions. To understand *System F*, *lambda calculus*, which underlies all functional languages and the procedural facilities of many more general languages, is introduced.

3.1 Lambda Calculus

Originally developed and investigated in the 1930's by logicians such as Church, Curry, and Rosser, lambda calculus has influenced the design of programming languages. Many problems of language design and implementation, especially those concerning procedure mechanisms or type structures, can be posed and investigated more easily by the lambda calculus than in more complex languages. For example, for the long and repetitive expression,

$$(5 \times 4 \times 3 \times 2 \times 1) + (7 \times 6 \times 5 \times 4 \times 3 \times 2 \times 1) - (3 \times 2 \times 1)$$

we rewrite it to be $\texttt{factorial}\ 5 + \texttt{factorial}\ 7 - \texttt{factorial}\ 3$, where

$$\texttt{factorial}\ n = \textsf{if}\ n = 0\ \textsf{then}\ 1\ \textsf{else}\ n \times\ (\texttt{factorial}\ n - 1)$$[1]

For each nonnegative number n, instantiating the function `factorial` with the argument n yields the factorial of n as a result. If "$\lambda\, n.\ \ldots$" is an abbreviation for "the function that, for each n, yields ...", the definition of `factorial` is restated as:

$$\texttt{factorial} = \lambda\, n.\textsf{if}\ n = 0\ \textsf{then}\ n \times (\texttt{factorial}\ n - 1)$$

where the subexpression, $\lambda\, n. \ldots$, is called an *abstraction* or *lambda expression*. The *function application* `factorial` 0 refers to "the result when the argument variable n in the function body ($\lambda\, n.\textsf{if}\ n = 0\ \textsf{then}\ \ldots$) is replaces by 0"; that is, 1.

In lambda calculus, a function is defined in the *abstraction* form, which does not specify the name of the function. The capability of denoting functions without giving them names makes abstractions particular useful for describing functions that are arguments for other "higher order" functions.

3.1.1 Syntax

The syntax of lambda calculus is comprised of three types of expressions (*terms*). A variable x by itself is a term; an abstraction of a variable x from

[1] Here the *curried form* "`factorial` n" is used instead of the *uncurried form* "`factorial` (n)" to deal with the higher order function that can treat functions as arguments.

a term t_1, written $\lambda\, x.t_1$, is a term; and the application of a term t_1 to another term t_2, written $t_1\ t_2$, is a term. These ways of forming terms are summarized in the following abstract syntax:

$$\begin{array}{llll} t & ::= & x & \textit{variable} \\ & | & \lambda\, x.t_1 & \textit{abstraction} \\ & | & t_1\ t_2 & \textit{application} \end{array}$$

It is assumed that the application is left associative, that is,

$$t_1\ t_2\ t_3 \ldots\ t_n \equiv (\ldots((t_1\ t_2)\ t_3)\ldots t_n)$$

and in $\lambda\, x.t$, the subterm t extends the first stop symbol or to the end of the enclosing phrase. For example,

$$\lambda\, x.(\lambda\, y.xyx)x \equiv \lambda\, x.((\lambda\, y.((xy)x))x)$$

Definition 3 (Bound and Free Variable) In the term $\lambda\, x.t$, the occurrence of x is a binder with scope t. An occurrence of variable x is said to be *bound*, when it occurs in body t of an abstraction $\lambda\, x.t$. An occurrence of x is *free* if it appears in a position where it is not bound by an enclosing abstraction of x.

Example 3.1.1 The occurrence of x in xy and $\lambda\, y.xy$ are free, whereas the ones in $\lambda\, x.x$ and $\lambda\, z.\,\lambda\, x.\,\lambda\, y.x(y\ z)$ are bound. In $(\lambda\, x.x)\ x$, the first occurrence of x is bound and the second one is free.

The set of free variables in term t, denoted as $FV(t)$, is defined as

$$\begin{array}{rcl} FV(x) & = & \{x\} \\ FV(t_1 t_2) & = & FV(t_1) \cup FV(t_2) \\ FV(\lambda\, x.t) & = & FV(t) - \{x\} \end{array}$$

and term t is said to be *closed*, if and only if $FV(t) = \emptyset$.

Definition 4 (α-conversion) α-conversion is the operation of replacing an occurrence of a bound variable by

$$\lambda\, x.t \equiv_\alpha \lambda\, y.t[x \mapsto y]$$

where $y \notin \; FV(t)$

Terms t and t' are α-equivalent with $t \equiv_\alpha t'$, if t' results from t by a series of changes of the bound variables. The α-conversion states that the renaming should preserve the semantic meaning.

3.1.2 Operational Semantics (Reduction)

The computation in lambda calculus is a sequence of applications of function to its arguments (which themselves are functions). Each step in the computation is a term rewriting that substitutes the arguments for bound variables in the abstraction's body.

Definition 5 (Substitution) For the arbitrary terms t_1, t_2 and variable x, $t_1[x \mapsto t_2]$ is defined as the result of substituting t_2 for each free occurrence of x in t_1.

Definition 6 (Redex and β-reduction) A term matching the form

$$(\lambda\, x.t_1)\; t_2$$

is called *redex* (a reducible expression), and the operation of rewriting a redex,

$$(\lambda\, x.t_1)\; t_2 \rightsquigarrow t_1[x \mapsto t_2]$$

is called a *β-reduction.*

According to the definitions, simple β-reductions are shown as follows:

$$\begin{aligned}(\lambda\, x.x)\ y &\ \leadsto\ x[x \mapsto y] \leadsto y \\ (\lambda\, x.x\ (\lambda\, x.x))\ (uv) &\ \leadsto\ x\ (\lambda\, x.x)[x \mapsto uv] \leadsto uv\ (\lambda\, x.x)\end{aligned}$$

If t' is obtained from t by zero or more β-reductions (or just by renaming), then t *reduces to* t', and is written as $t \leadsto^* t'$. In lambda calculus, the reduction sequence is a type of execution, and the obvious computation termination is defined by the reduction sequence reaching a term that contains no redexes.

Definition 7 (Normal Form) A term t is in a *normal form* if it does not have a redex as a subexpression. Moreover, if t is reduced to the normal form t', then t' is called a normal form of term t.

The following are examples of reduction sequences, terminating in normal forms:

$$\begin{aligned}\underline{(\lambda\, x.y)(\lambda\, z.z)} &\ \leadsto\ y \\ \underline{(\lambda\, x.x)(\lambda\, z.z)} &\ \leadsto\ \lambda\, z.z \\ \underline{(\lambda\, x.xx)(\lambda\, z.z)} &\ \leadsto\ \underline{(\lambda\, z.z)(\lambda\, z.z)} \leadsto (\lambda\, z.z) \\ \underline{(\lambda\, x.(\lambda\, y.yx)z)}(t\ \ v) &\ \leadsto\ \underline{(\lambda\, x.zx)(t\ \ v)} \leadsto z\ \ (t\ \ v) \\ \underline{(\lambda\, x.(\lambda\, y.yx)z)(t\ \ v)} &\ \leadsto\ \underline{(\lambda\, y.y(t\ \ v))\ \ z} \leadsto z\ \ (t\ \ v)\end{aligned}$$

Any redexes can be reduced at any time. At each step, some redexes (denoted by underlining) are chosen. The last two reduction sequences begin with the same expression and move in different directions (by reducing different redexes), but eventually return to the same normal form. The following theorem confirms that this is a general phenomenon for lambda calculus.

$$\begin{tikzcd}
& t \arrow[dl] \arrow[dr] & \\
t_0 \arrow[dr] & & t_1 \arrow[dl] \\
& t' &
\end{tikzcd}$$

Theorem 1 (Church-Rosser (Confluence) Property) If $t \rightsquigarrow^* t_0$ and $t \rightsquigarrow^* t_1$, then there is a term t' such that $t_0 \rightsquigarrow^* t'$ and $t_1 \rightsquigarrow^* t'$ [2].

This property states the uniqueness of the normal form, independent of the existence of normal form. As a special case, since normal forms can be reduced only by renaming, and if t_0 and t_1 are normal forms, then both must be α-equivalent to t', and therefore, to each other.

Theorem 2 If a term t exists a normal form n, then in α-equivalence, the normal form, n, is unique.

Unfortunately, untyped lambda calculus terms may not have a normal form is false. In the following examples of β-reductions:

$$\begin{aligned}
(\lambda\, x.xx)(\lambda\, x.xx) &\rightsquigarrow (\lambda\, x.xx)(\lambda\, x.xx) \rightsquigarrow \ldots \\
(\lambda\, x.f(x\ x))(\lambda\, x.f(x\ x)) &\rightsquigarrow f((\lambda\, x.f(x\ x))(\lambda\, x.f(x\ x))) \rightsquigarrow \ldots
\end{aligned}$$

the β-reduction sequences never terminate. For each initial term, there are no other reduction sequences, so that these initial terms have no normal form. Meanwhile, there are cases where both terminating and nonterminating sequences begin with the same term by different redexes. For example,

$$\begin{array}{lcl} \underline{(\lambda\, y.c)((\lambda\, x.xxx)(\lambda\, x.xxx))} & \rightsquigarrow & c \\ (\lambda\, y.c)\underline{((\lambda\, x.xxx)(\lambda\, x.xxx))} & \rightsquigarrow & (\lambda\, y.c)\underline{((\lambda\, x.xxx)(\lambda\, x.xxx))} \rightsquigarrow \ldots \end{array}$$

For untyped lambda calculus, the existence of normal form is *undecidable* [7], which is immediate corollary of the *undecidability* of the *β-equivalence* of two lambda terms.

In summary, two important facts of untyped lambda calculus should be stated:

- any reduction sequence terminating in a normal form produces the same result. Computationally, it means that every expression describes an unique computation if it exists.
- not all reduction sequences end in a normal form even choosing the reduction strategy; the existence of normal form is undecidable.

So far, untyped lambda calculus, which is a solid core of functional programming languages, has been examined. Lambda calculus is known to be computationally equivalent in power to many other models for computation (including Turing machines); that is, any calculation that can be accomplished in any of these other models can be expressed in the lambda calculus, and vice versa [11]. But without type constraints, lambda calculus causes problems: the termination of the evaluation without type constraints is undecidable.

Due to the limitation of untyped lambda calculus, researchers have been investigating type systems for lambda calculus. In simply typed-lambda calculus, type constraints are imposed on terms to prevent self-application, and to ensure its strong normalization property. But simple typed-lambda calculus has very little expressive power [7], and it is too weak to represent the XQ data types and operations. However, *System F*, a second-order lambda calculus, guarantees the termination of all computations, while retaining sufficient expressive power to encode XQ.

3.2 Introduction to System F

System F was first discovered by Girard [7], in 1971 as a system for constructive proofs in PA_2, and by Reynolds [11] in 1974 in the area of programming language. It is a extension of simple untyped lambda with treating types as variables and allowing new forms of *type abstractions* and *type applications*. Let us start with the ordinary untyped lambda calculus and add the capability to pass types as parameters. In untyped lambda calculus, the lambda *abstraction*, $(\lambda\, x.t)$, is used to abstract variables out of terms, and *application* is used to supply values for the abstracted parts without considering type constraints.

Example 3.2.1 The abstraction $\lambda f.\, \lambda\, x.f(f\ x)$ [10], in untyped lambda calculus, denotes the function `double`, which accepts a function as an argument, and yields the composition of this function to itself. To extend this definition with type variable T, we rewrite it as:

$$\lambda f^{T\to T}.\, \lambda\, x^{T}.f(f\ x)$$

denoting the function `double` for type T. By abstracting on type variable T

with the symbol “Λ”, a polymorphic function `double` is defined as

$$\Lambda T.\, \lambda f^{T \to T}.\, \lambda x^{T}. f(f\ x)$$

that can be applied to any type to obtain the corresponding “double” function for that type.

3.2.1 Syntax of System F

After extending lambda calculus with type variables, types become type expressions [7]:

- type variables X, Y, Z, $\ldots$ are types;
- if U and V are types, then function type $U \to V$ is a type, which denotes the set of functions that return a value of type V when applied to a value of type U;
- if V is a type and X is a type variable, then the polymorphic type $\forall X. V$ is a type, which denotes the type of polymorphic function that, when applied to a type X, yields a result of type V

Therefore, there are five ways to form *terms*:

- a *variable* x of type T is a term;
- an *abstraction* $\lambda x^{U}.v$ is a term of type $U \to V$, where x is a variable of type U and v is of type V;
- an *application* $t\ u$ is a term of type V, where t is of type $U \to V$, and u is type of U;

Type $::= X$	*type variable*
$\mid U \rightarrow V$	*function type*
$\mid \forall X. V$	*polymorphic type*
Term $::= x$	*variable*
$\mid \lambda x^{U}.v$	*abstraction*
$\mid t\ u$	*application*
$\mid \Lambda X.v$	*type abstraction*
$\mid t\ [U]$	*type application*

Figure 3.1: Syntax definition of *System F*

- a *type abstraction* $\Lambda X.v$ is a term of type $\forall X. V$, where v is a term of type V;
- a *type application* $t\ [U]$ is a term of type $V[X \mapsto U]$, where t is a term of type $\forall X. V$, and U is a type

The syntax definition of *System F* is presented in Figure 3.1. As well as the usual conversion, the $\rightarrow$ associates to the right, where

$$T_1 \rightarrow T_2 \rightarrow \cdots \rightarrow\ T_n \rightarrow \alpha \quad \text{stands for}(T_1 \rightarrow (T_2 \rightarrow \cdots \rightarrow\ (T_n \rightarrow \alpha)))$$

and application associates to the left, where

$$f\ t_1\ t_2 \ldots t_n \quad \text{stands for} \quad (\ldots((f\ t_1)\ t_2)\ldots)\ t_n$$

The β reduction "$\rightsquigarrow$" is extended to denote a new rule regarding the type applications.

Definition 8 (β-reduction)

$$(\lambda\, x.t_1)\ t_2 \rightsquigarrow t_1[x \mapsto t_2] \tag{3.1}$$

$$(\Lambda X.t)[U] \rightsquigarrow t[X \mapsto U] \tag{3.2}$$

where t_1, t_2 are terms, x is a term variable, and X is a type variable.

During an evaluation, the first rule denotes the usual *β-reduction*; that is, when a function, in the abstraction form of $\lambda\, x.t_1$, is applied to term t_2, the pair forms a *redex*, and term t_2 will substitute for all the occurrences of the bound variable x in t_1. The second rule states that, when a function, which is a type abstraction, $\Lambda X.t$, is applied to a type value U, the pair is also a *redex* and type U will substitute for the occurrences of type variable X in term t.

Example 3.2.2 The double function $\Lambda\, T.\, \lambda\, f^{T\rightarrow T}.\, \lambda\, x^T.f\ (f\ x)$
for type `int` becomes

$$\begin{aligned} & (\Lambda\, T.\, \lambda\, f^{T\rightarrow T}.\, \lambda\, x^T.f(f\ x))\ [\texttt{int}] \\ \rightsquigarrow\ & (\lambda f^{T\rightarrow T}.\, \lambda\, x^T.f(f\ x))[T \mapsto \texttt{int}] \\ \rightsquigarrow\ & \lambda f^{\texttt{int}\rightarrow\texttt{int}}.\, \lambda\, x^{\texttt{int}}.f(f\ x) \end{aligned}$$

and for the complex function type `real` $\rightarrow$ `real`, becomes

$$\begin{aligned} & (\Lambda\, T.\, \lambda\, f^{T\rightarrow T}.\, \lambda\, x^T.f(f\ x))\ [\texttt{real} \rightarrow \texttt{real}] \\ \rightsquigarrow\ & (\lambda f^{T\rightarrow T}.\, \lambda\, x^T.f(f\ x))[T \mapsto (\texttt{real} \rightarrow \texttt{real})] \\ \rightsquigarrow\ & \lambda f^{(\texttt{real}\rightarrow\texttt{real})\rightarrow(\texttt{real}\rightarrow\texttt{real})}.\, \lambda\, x^{\texttt{real}\rightarrow\texttt{real}}.f(f\ x) \end{aligned}$$

3.2.2 Why System F?

It was mentioned in Section 3.1.2 that in lambda calculus, untyped terms can not guarantee to have a normal form; that is, the termination of computation is undecidable. But when we considering types in the lambda calculus, the things have changed. It is the case that the typed-terms always possess normal forms [3]. This property also holds for *System F* [7].

Theorem 3 (Strong Normalization Theorem) All the well-typed terms of *System F* have normal forms. Moreover, all the terms are strongly normalizable, and there are no infinite reduction sequences, regardless of the evaluation strategy.

For languages in which each expression describes a terminating computation, such as simple typed lambda calculus, there must be computable functions that cannot be expressed [3]. Indeed we are used to taking this fact as evidence that such language are uninteresting for practical computation. Yet *System F* is such a language, in which one can express “almost everything” that one might actually want to compute. Indeed, Girard has shown that *System F* provides a surprising degree of expressiveness for the computations for a variety of data types [7].

3.3 Expressive Power of System F

In this section, it is demonstrated that a variety of primitive data types such as boolean, number, and list can be encoded in *System F*. These encodings are interesting for two reasons. First, they exemplify type abstraction and application. Secondly, they demonstrate that *System F* is, computationally, a very rich language, in the sense that the pure system can express a large range of data and control structures. This means that, when we later design a full-scale language with *System F* as its core, we can add these features as primitives (for efficiency, and so that we can equip them with a more convenient concrete syntax) without disturbing the fundamental properties of the core language.

3.3.1 Inductive Types

The expression power of System F allows the representation of basic types such as integers, lists, and trees by defining the inductive types.

Definition 9 (Inductive Type) An *inductive type* Θ is a data type that can be constructed by a set of constructors $f_1,\ f_2,\ \ldots,\ f_k$, with corresponding types:

$$\begin{array}{lll} f_1 & : T_1 & \text{with } T_1 = T_{11} \to \cdots \to T_{1n_1} \to \Theta \\ f_2 & : T_2 & \text{with } T_2 = T_{21} \to \cdots \to T_{2n_2} \to \Theta \\ \vdots & & \\ f_k & : T_k & \text{with } T_k = T_{k1} \to \cdots \to T_{1n_k} \to \Theta \end{array}$$

where Θ *occurs positively* in any T_{ij}, and each term of Θ can be represented by constructor f_i uniquely [7].

Definition 10 (Representation of the Inductive Type) An inductive type, Θ, which is described by the constructors $f_1,\ f_2, \ldots,\ f_k$ in Definition 9, is represented by

$$\begin{aligned}\Theta \;=\;& \forall X.\,(T_{11} \to \cdots \to T_{1n_1} \to \Theta)[\Theta \mapsto X]\\ & \to (T_{21} \to \cdots \to T_{2n_2} \to \Theta)[\Theta \mapsto X]\\ & \qquad\vdots\\ & \to (T_{k1} \to \cdots \to T_{kn_k} \to \Theta)[\Theta \mapsto X]\\ & \to X\end{aligned}$$

where $T_i[\Theta \mapsto X]$ denotes T_i, where all the occurrences of Θ are replaced by a type variable X.

The most interesting part of *System F* is its expressive power to define the common types. Now that the general schema of the representation of type and the corresponding iterator function have been introduced, some examples are provided.

3.3.2 Representation of Simple Types

In this section, the general schema, defined in Section 3.3.1, is applied to describe the common types such as boolean, product type, natural number, and list.

Example 3.3.1 (Boolean) The boolean type, Bool, has two functions: `True` and `False`, 0-ary functions of type Bool. According the Definition 9, the boolean type is defined as

$$\mathsf{Bool} \overset{\text{def}}{=} \forall X.X \to X \to X$$

and the two 0-ary constructors are

$$\mathsf{True} \overset{\text{def}}{=} \Lambda X.\,\lambda\, x^X.\,\lambda\, y^X.x$$
$$\mathsf{False} \overset{\text{def}}{=} \Lambda X.\,\lambda\, x^X.\,\lambda\, y^X.y$$

For e, e' of type U, and b of type Bool, the condition expression can be defined as an iterator function:

$$\mathsf{if}\ b\ \mathsf{then}\ e\ \mathsf{else}\ e' \overset{\text{def}}{=} b\ [U]\ e\ e'$$

and the following reductions prescribe the right behaviour:

$$\begin{aligned}
\mathsf{if}\ \underline{\mathtt{True}}\ \mathsf{then}\ e\ \mathsf{else}\ e' &\overset{\text{def}}{=} (\underline{\Lambda X.\,\lambda\, x^X.\,\lambda\, y^X.x})[U]\ e\ e' \\
&\rightsquigarrow (\lambda\, x^U.\,\lambda\, y^U.x)\ e\ e' \\
&\rightsquigarrow (\lambda\, y^U.e)\ e' \\
&\rightsquigarrow e \\
\mathsf{if}\ \underline{\mathtt{False}}\ \mathsf{then}\ e\ \mathsf{else}\ e' &\overset{\text{def}}{=} (\underline{\Lambda X.\,\lambda\, x^X.\,\lambda\, y^X.y})[U]\ e\ e' \\
&\rightsquigarrow (\lambda\, x^U.\,\lambda\, y^U.y)\ e\ e' \\
&\rightsquigarrow (\lambda\, y^U.y)\ e' \\
&\rightsquigarrow e'
\end{aligned}$$

Moreover,

$$\begin{aligned}
\mathtt{not} &\overset{\text{def}}{=} \lambda\, a^{\mathtt{Bool}}.\Lambda X.\,\lambda\, x^X.\,\lambda\, y^X.b\ [X]\ y\ x \\
\mathtt{and} &\overset{\text{def}}{=} \lambda\, a^{\mathtt{Bool}}.\,\lambda\, b^{\mathtt{Bool}}.\Lambda X.\,\lambda\, x^X.\,\lambda\, y^X.a\ [X]\ (b\ [X]\ x\ y)y
\end{aligned}$$

where the boolean function `not` has the type $\mathtt{Bool} \rightarrow \mathtt{Bool}$, and the function `and` has type $\mathtt{Bool} \rightarrow \mathtt{Bool} \rightarrow \mathtt{Bool}$.

Example 3.3.2 (Integer) The integer type, Int, has two constructors: one is the zero function, `zero`, with type Int; the other is the successor function `succ` of type $\mathsf{Int} \to \mathsf{Int}$. Therefore, type Int is defined as

$$\mathsf{Int} \overset{\text{def}}{=} \forall X.X \to (X \to X) \to X$$

and we have the integer numbers as

$$\begin{array}{rcl} 0 & \overset{\text{def}}{=} & \Lambda X.\,\lambda x^X.\,\lambda f^{X\to X}.x \\ 1 & \overset{\text{def}}{=} & \Lambda X.\,\lambda x^X.\,\lambda f^{X\to X}.f\ x \\ 2 & \overset{\text{def}}{=} & \Lambda X.\,\lambda x^X.\,\lambda f^{X\to X}.f(f\ x) \\ \vdots & & \\ n & \overset{\text{def}}{=} & \Lambda X.\,\lambda x^X.\,\lambda f^{X\to X}.f^n x \end{array}$$

where $f^n x$ denotes $f(f(f \ldots (f\ x)))$ with n occurrences of f. The two basic constructors are defined as

$$\begin{array}{rcl} \mathtt{zero} & \overset{\text{def}}{=} & \Lambda X.\,\lambda x^X.\,\lambda f^{X\to X}.x \\ \mathtt{succ}\ t & \overset{\text{def}}{=} & \Lambda X.\,\lambda x^X.\,\lambda f^{X\to X}.f(t\ [X]\ x\ f) \end{array}$$

Clearly, $\mathtt{zero} = 0$ and $\mathtt{succ}\ n = n + 1$.

Also, there is an iterator function, $\mathtt{It}_{\mathtt{int}}$, with type $\mathsf{Int} \to U$,

$$\mathtt{It}_{\mathtt{Int}}\ t = t\ [U]\ g_1\ g_2$$

where $g_1 : U$ and $g_2 : \mathsf{Int} \to U^2$, and now

$$\begin{array}{rcl}
\mathtt{It}_{\mathtt{Int}}\ \mathtt{zero} & \overset{\text{def}}{=} & \mathtt{zero}\ [U]\ g_1\ g_2 \\
& \overset{\text{def}}{=} & (\Lambda X.\,\lambda\, x^X.\,\lambda f^{X\to X}.x)[U]\ g_1\ g_2 \\
& \rightsquigarrow & (\lambda\, x^U.\,\lambda f^{U\to U}.x)\ g_1\ g_2 \\
& \rightsquigarrow & (\lambda f^{U\to U}.g_1)\ g_2 \\
& \rightsquigarrow & g_1
\end{array}$$

$$\begin{array}{rcl}
\mathtt{It}_{\mathtt{Int}}\ \mathtt{succ}\ t & \overset{\text{def}}{=} & (\mathtt{succ}t)\ [U]\ g_1\ g_2 \\
& \overset{\text{def}}{=} & (\Lambda X.\,\lambda\, x^X.\,\lambda f^{X\to X}.f(t\ [X]\ x\ f))[U]\ g_1\ g_2 \\
& \rightsquigarrow & (\lambda\, x^U.\,\lambda f^{U\to U}.f\ (t\ [U]\ x\ f))\ g_1\ g_2 \\
& \rightsquigarrow & (\lambda f^{U\to U}.f\ (t\ [U]\ g_1\ f))\ g_2 \\
& \rightsquigarrow & g_2(t\ [U]\ g_1\ \ g_2) \\
& = & g_2(\mathtt{It}_{\mathtt{Int}}\ t)
\end{array}$$

Based on the iterator, $\mathtt{It}_{\mathtt{Int}}$, in order to define a function $h : \mathsf{Int} \to U$, it is necessary to decide the corresponding $g_1 : U$ and $g_2 : \mathsf{Int} \to U$, considering the results, when function h is inductively applied to the constructor of type Int:

$$h\ 0 \;=\; g_1 \tag{3.3}$$

$$h\ (\mathtt{succ}\ n) \;=\; g_2(h\ n) \tag{3.4}$$

Then h is defined by using g_1 and g_2:

$$h\ n \overset{\text{def}}{=} n\ [U]g_1\ g_2 \qquad \text{that is} \qquad h\ \overset{\text{def}}{=} \lambda\, n^{\mathsf{Int}}.n\ [U]\ g_1\ g_2$$

[2] ":" means "has the type of"

The *addition* function, $\texttt{add} : \mathsf{Int} \rightarrow \mathsf{Int} \rightarrow \mathsf{Int}$, which adds two integer numbers, with the case:

$$\begin{array}{rcl} \texttt{add}\ m\ \texttt{zero} & = & m \\ \texttt{add}\ m\ (\texttt{succ}\ n) & = & \texttt{succ}\ (\texttt{add}\ m\ n) \end{array}$$

and matches the format of Equation 3.3 and 3.4, is defined as:

$$\texttt{add}\ m\ n \overset{\text{def}}{=} n\ [\mathsf{Int}]\ m\ \texttt{succ}$$

or more abstractly by

$$\texttt{add} \overset{\text{def}}{=} \lambda\, m^{\mathsf{Int}}.\, \lambda\, n^{\mathsf{Int}}.n\ [\mathsf{Int}]\ m\ \texttt{succ}$$

Example 3.3.3 (Product Type($\times$)) The expressive power of *System F* not only allows us to represent the primitive type, but also provides the capability to describe the various type constructors. In this thesis, the *product type*, $U \times V$, is demonstrated, allowing the construction of the pair (a, b), where a has the type U and b has the type V. The only constructor function for the product type is the pair function,

$$f : U \rightarrow V \rightarrow U \times V$$

which takes two arguments of type U and V, and returns a pair of type $U \times V$. Therefore, by Definition 9, a *product type* is signified as

$$U \times V \overset{\text{def}}{=} \forall X.(U \rightarrow V \rightarrow X) \rightarrow X$$

and, given $a : U$, $b : V$, the pair (a, b) is defined as

$$(a, b) \overset{\text{def}}{=} \Lambda X.\, \lambda f^{U \rightarrow V \rightarrow X}.f\ \ a\ \ b$$

Given a pair (a, b), the project functions are defined as

$$\begin{aligned}(a,b).0 &\overset{\text{def}}{=} (a,b)\ [U]\ (\lambda\, x^U.\,\lambda\, y^V.x)\\ (a,b).1 &\overset{\text{def}}{=} (a,b)\ [V]\ (\lambda\, x^U.\,\lambda\, y^V.y)\end{aligned}$$

The project function $(a, b).0$ is calculated by the β-reduction,

$$\begin{aligned}(a,b).0 &\overset{\text{def}}{=} (\Lambda X.\,\lambda f^{U\to V\to X}.f\ \ a\ \ b)\ [U]\ (\lambda\, x^U.\,\lambda\, y^V.x)\\ &\rightsquigarrow (\lambda f^{U\to V\to U}.f\ \ a\ \ b)\ (\lambda\, x^U.\,\lambda\, y^V.x)\\ &\rightsquigarrow (\lambda\, x^U.\,\lambda\, y^V.x)\ \ a\ \ b\\ &\rightsquigarrow (\lambda\, y^V.a)\ \ b\\ &\rightsquigarrow a\end{aligned}$$

which obtains the first element of the pair (a, b), and $(a, b).1$ returns the second element, since

$$\begin{aligned}(a,b).1 &\overset{\text{def}}{=} (\Lambda X.\,\lambda f^{U\to V\to X}.f\ \ a\ \ b)\ [V]\ (\lambda\, x^U.\,\lambda\, y^V.y)\\ &\rightsquigarrow (\lambda f^{U\to V\to V}.f\ \ a\ \ b)\ (\lambda\, x^U.\,\lambda\, y^V.y)\\ &\rightsquigarrow (\lambda\, x^U.\,\lambda\, y^V.y)\ \ a\ \ b\\ &\rightsquigarrow (\lambda\, y^V.y)\ \ b\\ &\rightsquigarrow b\end{aligned}$$

Example 3.3.4 (List) A list is a finite sequence of elements. Typical lists are $[2, 5, 6]$ and [“good”, “better”, “best”]. The empty list, [ ], has no elements. For a given list, the order of elements is significant, and elements may appear more than once. For instance, the following lists are different:

$$[3,5,9] \qquad [3,5,9,9] \qquad [3,9,5]$$

Each element in a list must have the same type. If the element u_0, u_1, $\ldots$, u_n are elements of type U, the list $[u_0, u_1, \ldots, u_n]$ has a type U list. The empty list, $[\,]$, has a polymorphic type α list, which is regarded as a list with any type of elements.

A list of type U list is constructed by the following two functions:

- $\texttt{nil} : U$ list, a synonym of the empty list, $[\,]$;
- construct function, $\texttt{cons} : U \to U\ \mathsf{list} \to U\ \mathsf{list}$, returns a new list by inserting an element of type U in front of an existing list of type U list

Therefore, each list is either `nil`, if empty, or has the form $\texttt{cons}\ u\ l$, where u is the *head* of type U and l is the *tail* of type U list. For the given elements u_0, u_1, $\ldots$, u_n, the list $[u_0, u_1, \ldots, u_n]$ is constructed as follows:

$$\begin{aligned}
\texttt{nil} &= [\,] \\
\texttt{cons}\ u_n\ \texttt{nil} &= [u_n] \\
\texttt{cons}\ u_{n-1}\ [u_n] &= [u_{n-1}, u_n] \\
&\vdots \\
\texttt{cons}\ u_0\ [u_1, \ldots, u_n] &= [u_0, u_1, \ldots, u_{n-1}, u_n]
\end{aligned}$$

With the schema of Definition 9, the type list U is defined as:

$$U\ \mathsf{list} \overset{\text{def}}{=} \forall X.X \to (U \to X \to X) \to U$$

and the individual list $[u_0, u_1, \ldots, u_n]$ is

$$\begin{gathered}
\texttt{cons}\ u_0\ (\texttt{cons}\ u_1\ \ldots\ (\texttt{cons}\ u_n\ \texttt{nil})\ldots) \overset{\text{def}}{=} \\
\Lambda X.\, \lambda\, x^X.\, \lambda f^{U \to X \to X}.f\ \ u_0\ (f\ \ u_1 \ldots (f\ \ u_n\ x))
\end{gathered}$$

In addition, two constructor functions are denoted as:

$$\begin{aligned}
\texttt{nil} &\overset{\text{def}}{=} \Lambda X.\, \lambda x^{U}.\, \lambda f^{U\to X\to X}.x \\
\texttt{cons}\ t\ ts &\overset{\text{def}}{=} \Lambda X.\, \lambda x^{U}.\, \lambda f^{U\to X\to X}.f\ t\ (ts\ [X]\ x\ f)
\end{aligned}$$

where t is an element of type U, and ts is a list of type $\mathsf{list}\ U$.

Definition 11 (Iterator on List) The iterator function on lists $\texttt{It}_{\texttt{list}} : U\ \mathsf{list} \to W$ takes an argument of type $U\ \mathsf{list}$ and returns the result of type W, and is denoted as

$$\texttt{It}_{\texttt{list}}\ ts \overset{\text{def}}{=} ts\ [W]\ g_1\ g_2 \tag{3.5}$$

where $g_1 : W$ and $g_2 : U \to W \to W$, and $\texttt{It}_{\texttt{list}}$ satisfies,

$$\begin{aligned}
\texttt{It}_{\texttt{list}}\,\texttt{nil} &\overset{\text{def}}{=} \texttt{nil}[W]\ g_1\ g_2 \\
&\overset{\text{def}}{=} (\Lambda X.\, \lambda x^{U}.\, \lambda f^{U\to X\to X}.x)[W]g_1\ g_2 \\
&\rightsquigarrow (\lambda x^{U}.\, \lambda f^{U\to W\to W}.x)\ g_1\ g_2 \\
&\rightsquigarrow (\lambda f^{U\to W\to W}.g_1)\ g_2 \\
&\rightsquigarrow g_1
\end{aligned}$$

and

$$\begin{aligned}
\texttt{It}_{\texttt{list}}\ (\texttt{cons}\ t\ ts) &\overset{\text{def}}{=} (\texttt{cons}\ t\ ts)\ [W]\ g_1\ g_2 \\
&\overset{\text{def}}{=} (\Lambda X.\, \lambda x^{U}.\, \lambda f^{U\to X\to X}.f\ t\ (ts\ [X]\ x\ f))\ [W]g_1\ g_2 \\
&\rightsquigarrow (\lambda x^{U}.\, \lambda f^{U\to W\to W}.f\ t\ (ts\ [W]\ x\ f))\ g_1\ g_2 \\
&\rightsquigarrow (\lambda f^{U\to W\to W}.f\ t\ (ts\ [W]\ g_1\ f))\ g_2 \\
&\rightsquigarrow g_2\ t\ (ts\ [W]\ g_1\ g_2) \\
&= g_2\ t\ (\texttt{It}_{\texttt{list}}\ ts)
\end{aligned}$$

The iterator definition provides a schema to define a function $f : U\ \mathsf{list} \to W$ by determining the corresponding $g_1 : W$ and $g_2 : U\ \mathsf{list} \to W \to W$, which satisfy

$$f\ \mathtt{nil} = g_1 \quad \text{and} \quad f\ (\mathtt{cons}\ t\ ts) = g_2\ t\ (f\ ts) \tag{3.6}$$

such that function f is defined by Definition 11 as

$$f\ ts \overset{\text{def}}{=} ts\ [W]\ g_1\ g_2 \quad \text{that is,} \quad f \overset{\text{def}}{=} \lambda\, ts^{U\ \mathsf{list}}.ts\ [W]\ g_1\ g_2 \tag{3.7}$$

The *length* function $\mathtt{length} : U\ \mathsf{list} \to \mathsf{Int}$ calculates the number of elements in a given list, and it is known that

$$\begin{aligned} \mathtt{length}\ \mathtt{nil} &= 0 \\ \mathtt{length}\ (\mathtt{cons}\ t\ ts) &= 1 + (\mathtt{length}\ ts) = (\lambda\, t^{U}.\, \lambda\, c^{\mathsf{Int}}.1 + c)\ t\ (\mathtt{length}\ ts) \end{aligned}$$

that is, $g_1 = 0$ with type Int and $g_2 = \lambda\, t^{U}.\, \lambda\, c^{\mathsf{Int}}.1 + c$ with type $U \to \mathsf{Int} \to \mathsf{Int}$.

Therefore, the function `length` is defined using Equation 3.7 such that,

$$\mathtt{length} \overset{\text{def}}{=} \lambda\, ts^{U\ \mathsf{list}}.ts\ [\mathsf{Int}]\ 0\ (\lambda\, t^{U}.\, \lambda\, c^{\mathsf{Int}}.1 + c)$$

where type variable W is replaced by Int.

Now that the expressive power of *System F* has been illustrated. In fact, all the functions, which are *System F* representable, have the common property:

Proposition 2 The functions representable in *System F* are exactly those which are *provably total* in PA_2[3].

[3] A detained definition and proof can be found in [7]

In this chapter, the expressive power of *System F* has been demonstrated. The definition schema allows the representation of XQ and its basic operations in *System F*, as detailed in the following chapter.

Chapter 4

Encoding XQ in *System F*

Now that the expressive power of *System F* has been illustrated by simple data types, *System F* will presented as the core of XQ, to improve the generality and power of XQ. In this chapter, *System F* is adopted to represent the XML data model that has been defined as an ordered forest whose elements are rooted labelled trees [6]. By the representation of XML tree and forest data types in *System F*, all the basic operators of XQ are encoded by iterators defined on the XML tree and forest structures. Also a general XQ syntax is proposed in*System F* format, and the semantics are described by defining an *Environment* data type in *System F* with the corresponding operators.

4.1 XML data in *System F*

The dynamic interval method presents an XML document as an ordered forest whose elements are rooted labelled trees, shown in Definition 1. Without a tree definition, the XML forest constructors are not primitive, which means that the empty node([ ]), single node forest (`<s> XF </s>`), and concatenation constructor (`@`) are not independent. For example, a single node forest can be constructed by using an empty forest and a tree; the concatenation of two forests can also be represented recursively in a more primitive way with a new constructor **cons**, which will be described in Section 4.2.2. In order to precisely describe XML document, XML forest and XML tree are considered separately.

Definition 12 (XML Tree)

$$\mathit{XTree} \stackrel{\text{def}}{=} \texttt{Xnode}\ \mathit{String}\ \mathit{XForest}$$

where an XML tree is constructed a *String* s as its root and an XML forest ts as its children.

In this chapter, `Xnode` s ts is used, instead of `<s>`ts `</s>`, to express an XML tree whose root is string s. An XML forest is described as a list of ordered XML trees.

Definition 13 (XML Forest)

$$\mathit{XForest} \stackrel{\text{def}}{=} \texttt{nil}\ \mid\ \texttt{Cons}\ \mathit{XTree}\ \mathit{XForest}$$

where `nil` denotes an empty forest without elements, and `Cons` signifies the construction of an XML forest by putting an XML tree t in front of an existing XML forest ts, written as $ts' = \texttt{Cons}\ t\ ts$.

Simultaneous inductions occur in the *XTree* and *XForest* definitions: building an object of one type involves the object of another. For example, constructor `Cons` builds tree t by employing a forest, ts; `Xnode` takes tree t and forest *ts* to build a new forest, *ts'*. In order to solve this problem, a new new type, *Xtf*, which is a combination of type *XTree* and type *XForest*, containing constructors of both types, to solve the simultaneous construction inherently.

To describe type *Xtf*, a general type, called Sum type, is introduced.

Definition 14 (Sum Type) If U and V are types, the Sum of U and V, denoted as $U + V$, is defined as

$$U + V \overset{\text{def}}{=} (U \to \alpha) \to (V \to \alpha) \ \to \alpha$$[1]

Sum type denotes an *alternative* construction, where different values are paired with *tags*, with which the values can be processed by branch. For example, $\iota^1 u$ is a term of u of type U paired with tag 1. If t has type U and $f_1,\ f_2$ are functions, then $(f_1, f_2)\ \iota^1 t$ results in applying function f_1 to t.

For $u : U$, $v : V$, two constructors for the Sum type are defined as follows:

$$\begin{aligned} \iota^1\ u &\overset{\text{def}}{=} \lambda\, x^{U\to\alpha}.\ \lambda\, y^{V\to\alpha}.\ x\ u \\ \iota^2\ v &\overset{\text{def}}{=} \lambda\, x^{U\to\alpha}.\ \lambda\, y^{V\to\alpha}.\ y\ v \end{aligned}$$

For functions $f_1 : U \to R$, $f_2 : V \to S$, and t of type $U + V$,

$$(f_1, f_2)\ t \overset{\text{def}}{=} t[R + S]\ f_1\ f_2$$

and for terms $u : U$ and $v : V$, $(f_1, f_2)\ t$ satisfies

$$(f_1, f_2)\ \iota^1\ u = f_1\ u \tag{4.1}$$

$$(f_1, f_2)\ \iota^2\ v = f_2\ v \tag{4.2}$$

[1]The primitive types α and β (denoted by Greek letter) are employed to implicitly express the type polymorphism instead of using the *universal type* expression $\forall\,\alpha.T$, which binds the occurrences of α in T. For example, $(U \to \alpha) \to (V \to \alpha) \to \alpha \equiv \forall\,\alpha.(U \to \alpha) \to (V \to \alpha) \to \alpha$.

where

$$\begin{aligned}(f_1, f_2)\ \iota^1\ u &\overset{\text{def}}{=} (\lambda\, x^{U\to\alpha}.\ \lambda\, y^{V\to\alpha}.\ x\ u)[R+S]\ f_1\ f_2\\ &\rightsquigarrow (\lambda\, x^{U\to R+S}.\ \lambda\, y^{V\to R+S}.\ x\ u)\ f_1\ f_2\\ &\rightsquigarrow (\lambda\, y^{V\to R+S}.\ f_1\ u)\ f_2\\ &\rightsquigarrow f_1\ u\end{aligned}$$

and similarly, $(f_1, f_2)\ \iota^2\ v \rightsquigarrow^* f_2\ v$

The problem of the simultaneous induction of types *XTree* and *XForest* can be solved by using the concept of Sum. A new type *Xtf*, which is a Sum type that combines types *XTree* and *XForest*, is built. When a function is recursively applied to an argument, its action depends on the element type it visits. For example, when the boolean operation `equal` compares the equality of two trees, at the top level, `equal` is a tree operator, comparing two root labels. Then `equal` becomes a forest operator recursively applied to the root's children. For the elements of the children forests, `equal` behaves as a tree operator, and so on.

Definition 15 (*Xtf* Type) *Xtf* is defined as an Sum of *XTree* and *XForest*, which is represented as follows:

$$Xtf \overset{\text{def}}{=} XTree + XForest$$

and as a polymorphic type in *System F* denoted as:

$$Xtf \overset{\text{def}}{=} \alpha \to (String \to \beta \to \alpha) \to \beta \to (\alpha \to \beta \to \beta) \to (\alpha + \beta)$$

The new type *Xtf* has three constructors, which are inherited from *XTree* and *XForest*:

- the function, `Xnode`: *String* $\rightarrow \beta \rightarrow \alpha$, builds a new tree with *String* s and forest ts in `XNode` s ts;
- the empty forest, `nil` : β
- the function, `Cons`: $\alpha \rightarrow \beta \rightarrow \beta$, constructs a forest by inserting tree t before forest ts with `Cons` t ts

With the general scheme of *System F*, the constructors of type *Xtf* are treated as certain *abstractions*.

Definition 16 (Constructors for Type *Xtf*)

$$\begin{aligned}
\texttt{Xnode}\ s\ ts &\stackrel{\text{def}}{=} \lambda\, x_1^{String\rightarrow\beta\rightarrow\alpha} . \lambda\, y_1^{\beta} .\ \lambda\, y_2^{\alpha\rightarrow\beta\rightarrow\beta}. \\
&\quad x_1\ s\ (ts\ [\beta]\ x_1\ y_1\ y_2) && (4.3)\\
\texttt{nil} &\stackrel{\text{def}}{=} \lambda\, x_1^{String\rightarrow\beta\rightarrow\alpha} . \lambda\, y_1^{\beta} .\ \lambda\, y_2^{\alpha\rightarrow\beta\rightarrow\beta}.\ \ y_1 && (4.4)\\
\texttt{Cons}\ t\ ts &\stackrel{\text{def}}{=} \lambda\, x_1^{String\rightarrow\beta\rightarrow\alpha} . \lambda\, y_1^{\beta} .\ \lambda\, y_2^{\alpha\rightarrow\beta\rightarrow\beta} . \\
&\quad y_2\ (t\ [\alpha]\ x_1\ y_1\ y_2)(ts\ [\beta]\ x_1\ y_1\ y_2) && (4.5)
\end{aligned}$$

where (4.3) is a constructor from *XTree*, (4.4) and (4.4) are inherited from *XForest*.

With type *Xtf* and its constructors, an individual XML tree and forest can be represented by a function in *abstraction* form, shown in Example 4.1.1.

Example 4.1.1 An XML tree in Fig 4.1 is represented as

$$\begin{aligned}
t &\stackrel{\text{def}}{=} \lambda\, x_1^{String\rightarrow\beta\rightarrow\alpha} .\ \lambda\, y_1^{\beta} .\ \lambda\, y_2^{\alpha\rightarrow\beta\rightarrow\beta}. \\
&\quad x_1\ a\ (y_2\ (x_1\ b\ (y_2\ (x_1\ c\ y_1)\ y_1))\ (y_2\ (x_1\ d\ y_1)\ y_1))
\end{aligned}$$

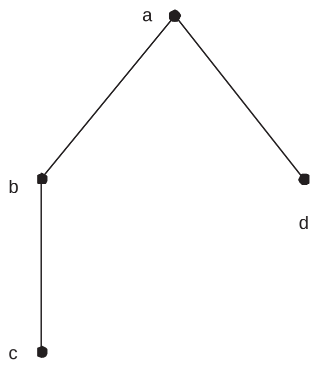

Figure 4.1: An XML tree

we recognize that, with replacing x_1 by `Xnode`, y_2 by `Cons`, and y_1 by `nil`, t becomes

$$\texttt{Xnode}\ a\ (\texttt{Cons}\ (\texttt{Xnode}\ b\ (\texttt{Cons}\ (\texttt{Xnode}\ c\ \texttt{nil})\ \texttt{nil}))$$
$$(\texttt{Cons}\ (\texttt{Xnode}\ d\ \texttt{nil})\ \texttt{nil}))$$

which is obtained by reducing t [*XTree*] `Xnode nil Cons`.

As an Sum type, *Xtf* combines the properties of both *XTree* and *XForest*. Therefore, *Xtf*'s induction function, *iterator* `It`, can also manipulates both *Xtree* and *XForest* data.

Definition 17 (Iterator for type *Xtf*) Given object v of type V, functions f_T : *String* $\rightarrow V \rightarrow U$ and $f_{XF} : U \rightarrow V \rightarrow V$, with the definition *Xtf* $=$ *XTree* $+$ *XFroest*, for an object m of type *Xtf*, its iterator, `It`, is defined as

$$\texttt{It}\ m \overset{\text{def}}{=} m\ [U + V]\ f_T\ v\ f_{XF}$$

where the resulting type of `It` depends on its arguments. For object t of type *XTree*,

$$\texttt{It}\ t^{2} \overset{\text{def}}{=} t\ [U]\ f_T\ v\ f_{XF}$$

with the result type U. For object *ts* of type *XForest*,

$$\texttt{It}\ ts \overset{\text{def}}{=} ts\ [V]\ f_T\ v\ f_{XF}$$

resulting in type V.

Proposition 3 Given an object $v : V$, functions $f_T : String \to V \to U$, and $f_{XF} : U \to V \to V$, the iterator `It` satisfies

$$\begin{aligned} \texttt{It}\ (\texttt{Xnode}\ s\ ts) &= f_T\ s\ (\texttt{It}\ ts) && (4.6)\\ \texttt{It nil} &= v && (4.7)\\ \texttt{It}\ (\texttt{Cons}\ t\ ts) &= f_{XF}\ (\texttt{It}\ t)\ (\texttt{It}\ ts) && (4.8) \end{aligned}$$

Proof: From the definition of iterator `It`, for an object m of type *Xtf*, `It` is expressed as:

$$\texttt{It}\ m \overset{\text{def}}{=} m\ [U+V]\ f_T\ v\ f_{XF}$$

[2]Here, the sum symbol, ι, is omitted for convenience.

since

$$\begin{array}{rcl}
\texttt{It}\ (\texttt{Xnode}\ s\ ts) & \overset{\text{def}}{=} & (\texttt{Xnode}\ s\ ts)\ [U]\ f_T\ v\ f_{XF} \\
& \leadsto & (\lambda\, x_1^{String\to\beta\to\alpha}.\ \lambda\, y_1^{\beta}\ .\ \lambda\, y_2^{\alpha\to\beta\to\beta}\ .\ x_1 \\
& & s\ (ts\ [\beta]\ x_1\ y_1\ y_2))\ [U]\ f_T\ v\ f_{XF} \\
& \leadsto & (\lambda\, x_1^{String\to\beta\to U}.\ \lambda\, y_1^{\beta}\ .\ \lambda\, y_2^{U\to\beta\to\beta}\ .\ x_1 \\
& & s\ (ts\ [\beta]\ x_1\ y_1\ y_2))\ f_T\ v\ f_{XF} \\
& \leadsto & (\lambda\, y_1^{\beta}.\ \lambda\, y_2^{U\to\beta\to\beta}\ .\ f_T\ s\ (ts\ [\beta]\ f_T\ y_1\ y_2))\ v\ f_{XF} \\
& \leadsto & (\lambda\, y_2^{U\to V\to V}.\ f_T\ s\ (ts\ [V]\ f_T\ v\ y_2))\ f_{XF} \\
& \leadsto & f_T\ s\ (ts\ [V]\ f_T\ v\ f_{XF}) \\
& \leadsto & f_T\ s\ (\texttt{It}\ ts)
\end{array}$$

The following can be proved similarly:

$$\begin{array}{rcl}
\texttt{It nil} & = & v \\
\texttt{It}\ (\texttt{Cons}\ t\ ts) & = & f_{XF}\ (\texttt{It}\ t)\ (\texttt{It}\ ts)
\end{array}$$

□

With iterator `It`, all the basic XML operators in XQ are represented in *System F*.

4.2 Encoding Basic XML Operators

4.2.1 General Functions

Two general functions, `select_first` : $\alpha \to \beta \to \alpha$ and `select_second` : $\alpha \to \beta \to \beta$, where α and β are *primitive* types that can be instantiated to concrete types, are introduced. The first function returns the first argument, and the second function returns the second argument as follows:

$$\texttt{select\_first} = \lambda\, x^{\alpha}.\, \lambda\, y^{\beta}.\ x$$
$$\texttt{select\_second} = \lambda\, x^{\alpha}.\, \lambda\, y^{\beta}.\ y$$

4.2.2 Horizontal Operators

Simple horizontal operators, `head`, `tail`, `concat`, `reverse` and `select`, are defined. These operators deal only with elements at the XML forest top level (horizontal), and never recur in the tree structure (vertical) of each element. In this case, iterator `It` is simplified to a horizontal iterator $\texttt{It}_h$ as

$$\begin{aligned}
\texttt{It}_{\texttt{h}}\ (\texttt{Xnode}\ s\ ts) &= \texttt{Xnode}\ s\ ts && (4.9)\\
&= \mathsf{let}\ f_1 = \lambda\, s^{String}.\, \lambda\, b^{V}.\texttt{Xnode}\ s\ ts\ \texttt{in}\ f_1\ s\ (\texttt{It}_{\texttt{h}}\ ts) &&\\
\texttt{It}_{\texttt{h}}\ \texttt{nil} &= c_2 && (4.10)\\
\texttt{It}_{\texttt{h}}\ (\texttt{Cons}\ t\ ts) &= f_2\ t\ (\texttt{It}_{\texttt{h}}\ ts) && (4.11)\\
&= f_2\ (\texttt{It}_{\texttt{h}}\ t)\ (\texttt{It}_{\texttt{h}}\ ts) &&
\end{aligned}$$

When $\texttt{It}_h$ traversals along the structure of object m of type Xtf, for m an XML tree, $\texttt{It}_h$ returns m without recurring through m's tree structure. For m an XML forest, $\texttt{Cons}\ t\ ts$, function $f_2 : Xtf \to V \to V$ takes two arguments: the first element tree t and the result returned by $\texttt{It}_h$ recursively applied to the remaining forest ts. Equations (4.9-4.11) are rewritten (the second line of each equation) to fit the form of the standard iterator Equations (4.6-4.8). For object m of type Xtf, function $\texttt{It}_{\texttt{h}}$ is defined as

$$\texttt{It} \overset{\text{def}}{=} \lambda\, m^{Xtf}.\ m\ [V]\ (\lambda\, s^{String}.\, \lambda\, b^{V}.\ \texttt{Xnode}\ s\ ts)\ c_2\ f_2 \qquad (4.12)$$

With defining the corresponding functions c_2, f_2 and proper types V, a specified horizontal operator is expressed by $\texttt{It}_\texttt{h}$ Equation (4.12).

head This operator obtains the first tree element from an XML forest, and has type $\texttt{head} : \; \mathit{Xtf} \rightarrow \mathit{Xtf}$:

$$\begin{aligned} \texttt{head}\,\texttt{nil} &= \texttt{default} \\ \texttt{head}\;(\texttt{Cons}\;t\;\mathit{ts}) &= t \\ &= (\lambda\, t^{\mathit{Xtf}}.\,\lambda\, z^{\mathit{Xtf}}.\; t)\; t\;(\texttt{head}\;\mathit{ts}) \end{aligned}$$

In the case of empty forest, `nil`, `head` returns a *unit* type value `default` meaning *nothing*, where type *unit* is consistent with any other type. Using Equation (4.12) with V being *Xtf*, `head` is defined as

$$\begin{aligned} \texttt{head} \;\; &\overset{\text{def}}{=} \;\; \lambda\, m^{\mathit{Xtf}}.m\;(\lambda\, s^{\mathit{String}}.\,\lambda\, b^{\mathit{Xtf}}.\texttt{Xnode}\;s\;\mathit{ts})\;\texttt{default} \\ &\qquad (\lambda\, t^{\mathit{Xtf}}.\,\lambda\, z^{\mathit{Xtf}}.\; t) \end{aligned}$$

tail This operator returns all but the first tree element from an XML forest with type $\texttt{tail} : \; \mathit{Xtf} \rightarrow \mathit{Xtf}$. In the case of empty forest, `nil`, `tail` returns only `nil`.

$$\begin{aligned} \texttt{tail}\,\texttt{nil} &= \texttt{nil} \\ \texttt{tail}\;(\texttt{Cons}\;t\;\mathit{ts}) &= \mathit{ts} \end{aligned}$$

The second equation does not match the form of Equation (4.11). This case is treated by a new approach.

Let $m : \mathit{Xtf}$, $h : \mathit{Xtf} \to \mathit{Xtf} \times \mathit{Xtf}$, and $h\ m = (m,\ \texttt{tail}\ m)$, then

$$\begin{aligned}
h\,\texttt{nil} &= (\texttt{nil}, \texttt{tail}\,\texttt{nil}) \\
&= (\texttt{nil},\ \texttt{nil}) \\
h\ (\texttt{Cons}\ t\ \mathit{ts}) &= (\texttt{Cons}\ t\ \mathit{ts},\ \texttt{tail}\ (\texttt{Cons}\ t\ \mathit{ts})) \\
&= (\texttt{Cons}\ t\ \mathit{ts},\ \mathit{ts}) \\
&= (\texttt{Cons}\ t\ \mathit{ts},\ \texttt{select\_first}\ \mathit{ts}\ (\texttt{tail}\ \mathit{ts})) \\
&= \textsf{let}\ f_2 = \lambda\, t^{\mathit{Xtf}}.\ \lambda(k,\ z)^{\mathit{Xtf} \times \mathit{Xtf}}.\ (\texttt{Cons}\ t\ k, \\
&\qquad\qquad \texttt{select\_first}\ k\ z) \\
&\quad\ \texttt{in}\ f_2\ t\ (h\ \mathit{ts})
\end{aligned}$$

which fits the forms of Equations (4.10-4.11) with V as $\mathit{Xtf} \times \mathit{Xtf}$. With $\texttt{tail}\ (\texttt{Xnode}\ s\ \mathit{ts}) = \texttt{Xnode}\ s\ \mathit{ts}$, h has

$$\begin{aligned}
h\ \texttt{Xnode}s\ \mathit{ts} &= (\texttt{Xnode}\ s\ \mathit{ts},\ \texttt{Xnode}\ s\ \mathit{ts}) \\
&= \lambda\, s^{\mathit{String}}.\, \lambda(k, z)^{\mathit{Xtf} \times \mathit{Xtf}}.(\texttt{Xnode}\ s\ k, \\
&\qquad\qquad \texttt{Xnode}\ s\ k)\ s\ (\mathit{ts}, \texttt{tail}\ \mathit{ts}) \\
&= \textsf{let}\ f_1 = \lambda\, s^{\mathit{String}}.\ \lambda(k,\ z)^{\mathit{Xtf} \times \mathit{Xtf}}.\ (\texttt{Xnode}\ s\ k, \\
&\qquad\qquad \texttt{Xnode}\ s\ k) \\
&\quad\ \texttt{in}\ f_1\ s\ (h\ \mathit{ts})
\end{aligned}$$

By applying Equation (4.12), the following is defined:

$$\begin{aligned}
h \ \overset{\text{def}}{=}\ & \lambda\, m^{\mathit{Xtf}}.\ m\ [\mathit{Xtf} \times \mathit{Xtf}] \\
& \quad \textsf{let}\ f_1 = \lambda\, s^{\mathit{String}}.\, \lambda(k,\ z)^{\mathit{Xtf} \times \mathit{Xtf}}.(\texttt{Xnode}\ s\ k,\ \texttt{Xnode}\ s\ k) \\
& \quad\quad\ f_2 = \lambda\, t^{\mathit{Xtf}}.\, \lambda(k,\ z)^{\mathit{Xtf} \times \mathit{Xtf}}.(\texttt{Cons}\ t\ k, \texttt{select\_first}\ k\ z) \\
& \quad \textsf{in}\ f_1\ (\texttt{nil},\ \texttt{nil})\ f_2
\end{aligned}$$

and `tail` is the second element in the pair of type $[\mathit{Xtf} \times \mathit{Xtf}]$. Therefore,

$$\begin{aligned}
\texttt{tail} \quad \overset{\text{def}}{=} \quad & \lambda\, m^{\mathit{Xtf}}.\ (\ m\ [\mathit{Xtf} \times \mathit{Xtf}] \\
& \mathsf{let}\ f_1 = \lambda\, s^{\mathit{String}}.\ \lambda(k,\ z)^{\mathit{Xtf}\times\mathit{Xtf}}.\ (\texttt{Xnode}\ s\ k,\ \texttt{Xnode}\ s\ k) \\
& \quad\ \ f_2 = \lambda\, t^{\mathit{Xtf}}.\ \lambda(k,\ z)^{\mathit{Xtf}\times\mathit{Xtf}}.(\texttt{Cons}\ t\ k, \texttt{select\_first}\ k\ z) \\
& \mathsf{in}\ f_1\ (\texttt{nil},\ \texttt{nil})\ f_2).1
\end{aligned}$$

concat (@) For two XML forest ts_1 and ts_2, `concat` ts_1 ts_2[3] concatenates t_1 and t_2 with type of `concat` : $\mathit{Xtf} \rightarrow \mathit{Xtf} \rightarrow \mathit{Xtf}$, represented by

$$\begin{aligned}
\texttt{concat}\ \texttt{nil}\ y &= y \\
\texttt{concat}\ (\texttt{Cons}\ t\ ts)\ y &= \texttt{Cons}\ t\ (\texttt{concat}\ ts\ y)
\end{aligned}$$

These equations can be written more abstractly as

$$\begin{aligned}
\texttt{concat}\ \texttt{nil} &= \lambda\, y^{\mathit{Xtf}}.\ y \\
\texttt{concat}\ (\texttt{Cons}\ t\ ts) &= \texttt{Cons}\ t\ (\texttt{concat}\ ts)
\end{aligned}$$

which matches the forms of Equations(4.10-4.11) with type V as $\mathit{Xtf} \rightarrow \mathit{Xtf}$ so that the following can be defined:

$$\begin{aligned}
\texttt{concat} \quad \overset{\text{def}}{=} \quad & \lambda\, m^{\mathit{Xtf}}.\ m\ [\mathit{Xtf} \rightarrow \mathit{Xtf}]\ (\lambda\, s^{\mathit{String}}.\ \lambda\, b^{\mathit{Xtf}}.\ \texttt{Xnode}\ s\ ts) \\
& (\lambda\, y^{\mathit{Xtf}}.\ y)\ \texttt{Cons}
\end{aligned}$$

reverse This operator reverses an XML forest on the top level with type `reverse` : $\mathit{Xtf} \rightarrow \mathit{Xtf}$. For example, `reverse` $[t_1,\ t_2,\ t_3]$ returns

[3]Its infix form is @, represented by ts_1 @ ts_2.

$[t_3, t_2, t_1]$.

In order to define `reverse`, a subsidiary function, `rappend` : $\mathit{Xtf} \rightarrow \mathit{Xtf} \rightarrow \mathit{Xtf}$, is presented, such that

$$\texttt{rappend}\ [a, b, c]\ [3, 2, 1] = [1, 2, 3, a, b, c]$$

If the first argument is `nil`, `rappend` *reverses* the second argument so that

$$\begin{aligned} \texttt{rappend}\ y\ \texttt{nil} &= y \\ \texttt{rappend}\ y\ (\texttt{Cons}\ t\ ts) &= \texttt{Cons}\ t\ (\texttt{rappend}\ y\ ts) \end{aligned}$$

With Equation (4.12) and type V as Xtf, function `rappend` is denoted as

$$\begin{aligned} \texttt{rappend}\ y &\overset{\text{def}}{=} \lambda\, m^{\mathit{Xtf}}.\ m\ [\mathit{Xtf}]\ (\lambda\, s^{\mathit{String}}.\ \lambda\, b^{\mathit{Xtf}}.\ \texttt{Xnode}\ s\ ts)\ y \\ &\qquad (\lambda\, t^{\mathit{Xtf}}.\ \lambda\, z^{\mathit{Xtf}}.\ \texttt{Cons}\ t\ z) \\ \text{or} & \\ \texttt{rappend} &\overset{\text{def}}{=} \lambda\, y^{\mathit{Xtf}}.\lambda\, m^{\mathit{Xtf}}.m\ [\mathit{Xtf}]\ (\lambda\, s^{\mathit{String}}.\lambda\, b^{\mathit{Xtf}}.\texttt{Xnode}\ s\ ts)\ y \\ &\qquad (\lambda\, t.\ \lambda\, z.\ \texttt{Cons}\ t\ z) \end{aligned}$$

With `rappend`, `reverse` is expressed as follows:

$$\begin{aligned} \texttt{reverse}\ \texttt{nil} &= \texttt{nil} \\ \texttt{reverse}\ (\texttt{Cons}\ t\ ts) &= \texttt{rappend}\ (\texttt{Cons}\ t\ \texttt{nil})\ (\texttt{reverse}\ ts) \\ &= (\lambda\, t^{\mathit{Xtf}}.\ \texttt{rappend}\ \texttt{Cons}\ t\ \texttt{nil})\ t\ (\texttt{reverse}\ ts) \end{aligned}$$

which fit the recursive forms of Equations (4.10-4.11). By Equation

(4.12),

$$\begin{aligned}\texttt{reverse} \;&\overset{\text{def}}{=}\; \lambda\, m^{Xtf}.\; m\;[Xtf]\;(\lambda\, s^{String}.\;\lambda\, b^{Xtf}.\,\texttt{Xnode}\; s\;\, ts)\;\texttt{nil}\\ &\qquad(\lambda\, t^{Xtf}.\,\texttt{rappend Cons}\; t\;\texttt{nil})\end{aligned}$$

select For XML forest *ts*, and string *s*, `select` : *String* → *Xtf* → *Xtf* extracts the tree elements whose roots are labelled *s*.

First a function `rootLabel`: *Xtf* → *String* is defined to extract the roots label as follows:

$$\begin{aligned}\texttt{rootLabel}\;(\texttt{Xnode}\; s\;\; ts) \;&=\; s\\ &=\; \texttt{select\_first}\; s\;(\texttt{rootLabel}\; ts)\\ \texttt{rootLabel}\;\texttt{nil} \;&=\; \texttt{default}\\ \texttt{rootLabel}\;(\texttt{Cons}\; t\;\; ts) \;&=\; (\lambda\, x^{String}.\,\lambda\, y^{String}.\,\texttt{default})\\ &\quad\;\; (\texttt{rootLabel}\; t)\;(\texttt{rootLabel}\; ts)\end{aligned}$$

When `rootLabel` applies to an XML forest, it returns `default` which means *nothing*. The function `rootLabel` is then signified as

$$\begin{aligned}\texttt{rootLabel} \;&\overset{\text{def}}{=}\; \lambda\, m^{Xtf}.\; m\;[String]\;\texttt{select\_first}\;\texttt{default}\\ &\qquad(\lambda\, x^{String}.\,\lambda\, y^{String}.\,\texttt{default})\end{aligned}$$

With function `rootLabel`, `select` is described as

$$\begin{aligned}
\texttt{select}\ s\ \texttt{nil} &= \texttt{nil} \\
\texttt{select}\ s\ (\texttt{Cons}\ t\ ts) &= \textsf{if}\ s == (\texttt{rootLabel}\ t)\ \textsf{then} \\
&\quad \texttt{Cons}\ t\ (\texttt{select}\ s\ ts)\ \textsf{else}\ \texttt{select}\ s\ ts \\
&= \textsf{if}\ s == (\texttt{rootLabel}\ t)\ \textsf{then} \\
&\qquad \textsf{let}\ f = \lambda\, z^{Xtf}.\ \texttt{Cons}\ t\ z \\
&\qquad \textsf{in}\ f\ \ t\ (\texttt{select}\ s\ ts) \\
&\quad \textsf{else}\ \texttt{select\_second}\ t\ (\texttt{select}\ s\ ts)
\end{aligned}$$

where "==" is a boolean operator comparing strings equality, and can also be represented in *System F*: when a string is represented as a list of characters, the operator "==" is defined as a polymorphic function on list. Evidently, the previous equations fit (4.10-4.11) with type V as Xtf. Therefore, `select` is defined as

$$\begin{aligned}
\texttt{select} \overset{\text{def}}{=}\ & \lambda\, s^{String}.\ \lambda\, m^{Xtf}.\ m\ [Xtf] \\
& \textsf{let}\ f_1 = \lambda\, s^{String}.\ \lambda\, b^{Xtf}.\texttt{Xnode}\ s\ ts \\
& \quad f_2 = \lambda\, t^{Xtf}.(\textsf{if}\ s == (\texttt{rootLabel}\ t)\ \textsf{then} \\
& \qquad\quad \lambda\, z^{Xtf}.\texttt{Cons}\ t\ z\ \textsf{else}\ \lambda\, z^{Xtf}.\texttt{select\_second}\ t\ z) \\
& \textsf{in}\ f_1\ \texttt{nil}\ f_2
\end{aligned}$$

4.2.3 Vertical Operators

Vertical operators are considered to be the operators that recursively deal with the tree structure of each tree in an XML forest. For example, performing a depth-first traversal for tree t attains a forest, including t's subtrees, in a DFS order. Now we show how vertical operators `roots`, `children`, `subtreesdfs` are represented by polymorphic functions in *System F*.

roots The operator `roots` : $\mathit{Xtf} \rightarrow \mathit{Xtf}$ returns all the root nodes, indicating one node for each tree, and a list of root nodes for a forest,

$$\begin{aligned}
\texttt{roots}\ (\texttt{Xnode}\ s\ ts) &= \texttt{Xnode}\ s\ \texttt{nil} \\
&= \mathsf{let}\ f_1 = \lambda\, s^{\mathit{String}}.\ \lambda\, b^{\mathit{Xtf}}.\ \texttt{Xnode}\ s\ \texttt{nil}\ \mathsf{in} \\
&\quad\ f_1\ s\ (\texttt{roots}\ ts) \\
\texttt{roots}\ \texttt{nil} &= \texttt{nil} \\
\texttt{roots}\ (\texttt{Cons}\ t\ ts) &= \texttt{Cons}\ (\texttt{roots}\ t)\ (\texttt{roots}\ ts)
\end{aligned}$$

fits Equations (4.6-4.8) with U and V as type Xtf, and `roots` is defined as

$$\begin{aligned}
\texttt{roots} \ \overset{\text{def}}{=}\ & \lambda\, m^{\mathit{Xtf}}.\ m\ [\mathit{Xtf}]\ (\lambda\, s^{\mathit{String}}.\ \lambda\, b^{\mathit{Xtf}}.\ \texttt{Xnode}\ s\ \texttt{nil}) \\
& \texttt{nil}\ \texttt{Cons}
\end{aligned}$$

children For XML forest m, `children` : $\mathit{Xtf} \rightarrow \mathit{Xtf}$ recursively obtains children $c_i : \mathit{Xtf}$ for each tree element t_i, and concatenates all c_i, and is expressed as

$$\begin{aligned}
\texttt{children}\ (\texttt{Xnode}\ s\ ts) &= ts \\
\texttt{children}\ \texttt{nil} &= \texttt{nil} \\
\texttt{children}\ (\texttt{Cons}\ t\ ts) &= \texttt{concat}\ (\texttt{children}\ t)\ (\texttt{children}\ ts)
\end{aligned}$$

where the first equation does not match the form of Equation (4.6), but can be treated by the approach used for the `tail` function.

Let $g : \mathit{Xtf} \rightarrow \mathit{Xtf} \times \mathit{Xtf}$, such that $g\ m = (m,\ \texttt{children}\ m)$, then

$$\begin{aligned}
g\ (\texttt{Xnode}\ s\ ts) &= (\texttt{Xnode}\ s\ ts,\ ts) \\
&= \lambda\, s^{\mathit{String}}.\ \lambda(k,\ z)^{\mathit{Xtf}\times\mathit{Xtf}}.\ (\texttt{Xnode}\ s\ k,\ k)\ s\ (g\ ts) \\
&= \textsf{let}\ f_T = \lambda\, s^{\mathit{String}}.\ \lambda(k,\ z)^{\mathit{Xtf}\times\mathit{Xtf}}.\ (\mathit{Xnode}\ s\ k,\ k) \\
&\quad\ \textsf{in}\ f_T\ s\ (g\ ts) \\
g\ \texttt{nil} &= (\texttt{nil}, \texttt{children}\ \texttt{nil}) \\
&= (\texttt{nil},\ \texttt{nil}) \\
g\ (\texttt{Cons}\ t\ ts) &= (\texttt{Cons}\ t\ ts,\ \texttt{children}\ (\texttt{Cons}\ t\ ts)) \\
&= (\texttt{Cons}\ t\ ts,\ \ \texttt{concat}\ (\texttt{children}\ t)\ (\texttt{children}\ ts)) \\
&= \lambda(k_1,\ z_1)^{\mathit{Xtf}\ \times\mathit{Xtf}}.\ \lambda(k_2,\ z_2)^{\mathit{Xtf}\ \times\mathit{Xtf}}. \\
&\quad\ (\texttt{Cons}\ k_1\ k_2,\ \texttt{concat}\ z_1\ z_2)\ (g\ t)\ (g\ ts) \\
&= \textsf{let}\ f_{XF} = \lambda(k_1,\ z_1)^{\mathit{Xtf}\ \times\mathit{Xtf}}.\ \lambda(k_2,\ z_2)^{\mathit{Xtf}\ \times\mathit{Xtf}}. \\
&\quad\quad\ (\texttt{Cons}\ k_1\ k_2,\ \texttt{concat}\ z_1\ z_2) \\
&\quad\ \textsf{in}\ f_{XF}\ (g\ t)\ (g\ ts)
\end{aligned}$$

which fit the forms of Equation (4.6-4.8), and function g is defined as follows:

$$\begin{aligned}
g \overset{\text{def}}{=}\ & \lambda\, m^{\mathit{Xtf}}.\ m\ [\mathit{Xtf} \times \mathit{Xtf}] \\
&\quad \textsf{let}\ f_T = \lambda\, s^{\mathit{String}}.\ \lambda(k,\ z)^{\mathit{Xtf}\times\mathit{Xtf}}.\ (\texttt{Xnode}\ s\ k,\ k) \\
&\quad\quad f_{XF} = \lambda(k_1,\ z_1)^{\mathit{Xtf}\ \times\mathit{Xtf}}.\ \lambda(k_2,\ z_2)^{\mathit{Xtf}\ \times\mathit{Xtf}}. \\
&\quad\quad\quad (\texttt{Cons}\ k_1\ k_2,\ \texttt{concat}\ z_1\ z_2) \\
&\quad \textsf{in}\ f_T\ (\texttt{nil},\ \texttt{nil})\ f_{XF}
\end{aligned}$$

and `children` is the second element in the pair of type $[\mathit{Xtf} \times \mathit{Xtf}]$, expressed as

$$
\begin{aligned}
\texttt{children} \overset{\text{def}}{=} \ & \mathsf{let}\ g = \lambda\, m^{\mathit{Xtf}}.\ m\ [\mathit{Xtf} \times \mathit{Xtf}] \\
& \quad \mathsf{let}\ f_T = \lambda\, s^{\mathit{String}}.\ \lambda(k,\ z)^{\mathit{Xtf} \times \mathit{Xtf}}.\ (\texttt{Xnode}\ s\ k,\ k) \\
& \qquad f_{XF} = \lambda(k_1,\ z_1)^{\mathit{Xtf}\ \times \mathit{Xtf}}.\ \lambda(k_2,\ z_2)^{\mathit{Xtf}\ \times \mathit{Xtf}}. \\
& \qquad\qquad (\texttt{Cons}\ k_1\ k_2,\ \texttt{concat}\ z_1\ z_2) \\
& \quad \mathsf{in}\ f_T\ (\texttt{nil},\ \texttt{nil})\ f_{XF} \\
& \mathsf{in}\ g.1
\end{aligned}
$$

subtreesdfs For object m of type *Xtf*, and if m is an XML tree, `subtreesdfs` : $\mathit{Xtf} \to \mathit{Xtf}$ performs a depth-first traversal and returns all the subtrees of m, including itself; if m is an XML forest, `subtreesdfs` traverses each tree element t_i of m, obtains subtrees for each t_i, and concatenates them, such that:

$$
\begin{aligned}
\texttt{subtreesdfs}\ (\texttt{Xnode}\ s\ ts) &= \texttt{Cons}\ (\texttt{Xnode}\ s\ ts)\ (\texttt{subtreesdfs}\ ts) \\
\texttt{subtreesdfs}\ \texttt{nil} &= \texttt{nil} \\
\texttt{subtreesdfs}\ (\texttt{Cons}\ t\ ts) &= \texttt{concat}\ (\texttt{subtreesdfs}\ t) \\
& \quad\ (\texttt{subtreesdfs}\ ts)
\end{aligned}
$$

where the second equation does not fit iterator Equation(4.6). The approach in `children` is adopted by defining $g : \mathit{Xtf} \to \mathit{Xtf} \times \mathit{Xtf}$ such that $g\ m = (m,\ \texttt{subtreesdfs}\ m)$. For $\texttt{Xnode}\ s\ ts$,

$$\begin{aligned}
g\ (\texttt{Xnode}\ s\ ts) &= (\texttt{Xnode}\ s\ ts,\ \texttt{Cons}\ (\texttt{Xnode}\ s\ ts)\ (\texttt{subtreesdfs}\ ts)) \\
&= \lambda(k,\ z)^{Xtf \times Xtf}.\ (\texttt{Xnode}\ s\ k,\ \texttt{Cons}\ (\texttt{Xnode}\ s\ k)\ z) \\
&\quad (ts,\ \texttt{subtreesdfs}\ ts) \\
&= \mathsf{let}\ f_T = \lambda\, s^{String}.\ \lambda(k,\ z)^{Xtf \times Xtf}. \\
&\qquad\qquad (Xnode\ s\ k,\ \texttt{Cons}\ (\texttt{Xnode}\ s\ ts)\ z) \\
&\quad \mathsf{in}\ f_T\ s\ (g\ ts)
\end{aligned}$$

Then we define `subtreesdfs` is

$$\begin{aligned}
\texttt{subtreesdfs} \overset{\text{def}}{=}\ & \mathsf{let}\ g = \lambda\, m^{Xtf}.\ m\ [Xtf \times Xtf] \\
& \quad \mathsf{let}\ f_T = \lambda\, s^{String}.\ \lambda(k,\ z)^{Xtf \times Xtf}. \\
& \qquad\qquad (Xnode\ s\ k,\ \texttt{Cons}\ (\texttt{Xnode}\ s\ ts)\ z) \\
& \qquad f_{XF} = \lambda(k_1,\ z_1)^{Xtf\ \times Xtf}.\ \lambda(k_2,\ z_2)^{Xtf\ \times Xtf}. \\
& \qquad\qquad (\texttt{Cons}\ k_1\ k_2,\ \texttt{concat}\ z_1\ z_2) \\
& \quad \mathsf{in}\ f_T\ (\texttt{nil},\ \texttt{nil})\ f_{XF} \\
& \mathsf{in}\ g.1
\end{aligned}$$

4.2.4 Boolean Operators

XQ defines two structural boolean operators, `equal` and `less`, that define structural relationships of the forests. Also, operator `empty` is introduced to check the emptiness of a forest.

empty This operator tests the emptiness of an XML forest [4] with `empty` :

[4]When `empty` is applied to an XML tree, it is assumed that False is returned.

$\mathit{Xtf} \rightarrow \texttt{Bool}$, and represented as

$$\begin{aligned}
\texttt{empty}\ (\texttt{Xnode}\ s\ \mathit{ts}) &= \mathsf{False} \\
&= (\lambda\, a^{\mathit{String}}.\, \lambda\, b^{\mathit{Bool}}.\, \mathsf{False})\ s\ (\texttt{empty}\ \mathit{ts}) \\
\texttt{empty}\ \texttt{nil} &= \mathsf{True} \\
\texttt{empty}\ (\texttt{Cons}\ t\ \mathit{ts}) &= \mathsf{False} \\
&= (\lambda\, x^{\mathit{Bool}}.\ \lambda\, y^{\mathit{Bool}}.\, \mathsf{False})\ (\texttt{empty}\ t)\ (\texttt{empty}\ \mathit{ts})
\end{aligned}$$

For object $m : \mathit{Xtf}$, functions $f_T : \mathit{String} \rightarrow \mathit{Bool} \rightarrow \mathit{Bool}$, and $f_{XF} : \mathit{Bool} \rightarrow \mathit{Bool} \rightarrow \mathit{Bool}$, the boolean condition `empty` is defined by Equations(4.6-4.8) as follows:

$$\begin{aligned}
\texttt{empty} \overset{\text{def}}{=}\ & \lambda\, m^{\mathit{Xtf}}.\ m\ [\mathit{Bool}] \\
& \mathsf{let}\ f_T = \lambda\, x^{\mathit{String}}.\ \lambda\, y^{\mathit{Bool}}.\, \mathsf{False} \\
& \quad f_{XF} = \lambda\, x^{\mathit{Bool}}.\ \lambda\, y^{\mathit{Bool}}.\, \mathsf{False} \\
& \mathsf{in}\ f_T\ \mathsf{True}\ f_{XF}
\end{aligned}$$

equal The operator, `equal`, tests whether two objects of type *Xtf* are structurally equal.

Definition 18 (Structural Equality) Two objects ts_1 and ts_2 of type *Xtf* are structurally equal, if and only if they have the same constructor at the top level, and all the substructures are inductively equal.

Thus: ts_1 and ts_2 are *structurally equal* if and only if one of the following conditions holds:

- ts_1 and ts_2 are trees with the same root label, and their children nodes are recursively *equal*;

- ts_1 and ts_2 are `nil`;
- ts_1 and ts_2 are forests with constructor `Cons`, and their tree elements are recursively *structural equals*

The operator, `equal`, recursively tests two forests with type `equal` : $Xtf \rightarrow Xtf \rightarrow$ `Bool`. The definition of `equal` needs two functions that have been defined: `rootLabel` : $Xtf \rightarrow String$, which extracts the root label of a tree; and `empty` : $Xtf \rightarrow Bool$, which checks the emptiness of a forest. Operator `equal` can be expressed by using `rootLabel` and `empty` as follows:[5]:

$$\begin{aligned}
\texttt{equal}\ (\texttt{Xnode}\ s\ ts)\ m &= (s == \texttt{rootLabel m})\ \texttt{and} \\
&\quad (\texttt{equal}\ ts\ (\texttt{children}\ m)) \\
\texttt{equal}\ \texttt{nil}\ m &= \texttt{empty}\ m \\
\texttt{equal}\ (\texttt{Cons}\ t\ ts)\ m &= (\texttt{equal}\ t\ (\texttt{head}\ m))\ \texttt{and}\ (\texttt{equal}\ ts\ (\texttt{tail}\ m))
\end{aligned}$$

These equations are then written more abstractly as

$$\begin{aligned}
\texttt{equal}\ (\texttt{Xnode}\ s\ ts) &= \lambda\, m^{Xtf}.\ (s == \texttt{rootLabel m})\ \texttt{and} \\
&\qquad (\texttt{equal}\ ts\ (\texttt{children}\ m)) \\
&= \mathsf{let}\ f_T = \lambda\, s^{String}.\ \lambda\, z^{Xtf \rightarrow Bool}.\ \lambda\ m^{Xtf}. \\
&\quad (s == \texttt{rootLabel}\ m)\ \texttt{and}\ (z\ (\texttt{children}\ m)) \\
&\quad \mathsf{in}\ f_T\ s\ (\texttt{equal}\ ts) \\
\texttt{equal}\ \texttt{nil} &= \texttt{empty}
\end{aligned}$$

[5]Boolean operators `and` and `or` have been represented in *System F*[7].

$$\begin{aligned}
\texttt{equal}\ (\texttt{Cons}\ t\ ts) &= \lambda\, m^{Xtf}.\ (\texttt{equal}\ t\ (\texttt{head}\ m))\ \texttt{and} \\
&\qquad\qquad (\texttt{equal}\ ts\ (\texttt{tail}\ m)) \\
&= \textsf{let}\ f_{XF} = \lambda\, x^{Xtf \to Bool}.\ \lambda\ y^{Xtf \to Bool}.\ \lambda\, m^{Xtf}. \\
&\qquad (x\ (\texttt{head}\ m))\ \texttt{and}\ (y\ (\texttt{tail}\ m)) \\
&\quad\ \textsf{in}\ f_{XF}\ (\texttt{equal}\ t)\ (\texttt{equal}\ ts)
\end{aligned}$$

which matches the forms of Equations(4.6-4.8) with

$$\begin{aligned}
f_T &: String \to (Xtf \to Bool) \to Xtf \to Bool \\
f_{XF} &: (Xtf \to Bool) \to (Xtf \to Bool) \to Xtf \to Bool
\end{aligned}$$

leading to

$$\begin{aligned}
\texttt{equal} \ \overset{\text{def}}{=}\ & \lambda\, n^{Xtf}.\ n\ [Xtf \to Bool] \\
& \textsf{let}\ f_T = \lambda\, s^{String}.\ \lambda\, z^{Xtf \to Bool}.\ \lambda\ m^{Xtf}. \\
& \qquad\qquad (s == \texttt{rootLabel m})\ \texttt{and}\ (z\ (\texttt{children}\ m)) \\
& \quad\ f_{XF} = \lambda\, x^{Xtf \to Bool}.\ \lambda\ y^{Xtf \to Bool}.\ \lambda\, m^{Xtf}. \\
& \qquad\qquad (x\ (\texttt{head}\ m))\ \texttt{and}\ (y\ (\texttt{tail}\ m)) \\
& \textsf{in}\ f_T\ \texttt{empty}\ f_{XF}
\end{aligned}$$

less The boolean operator `less` examines the structural ordering of XML trees or forests.

Definition 19 (Structurally Less) For two objects m and n of type *Xtf*, m is structurally less than n, if and only if one of the following conditions recursively holds:

- m and n are trees, and one of the following conditions exclusively holds:
  - their root nodes satisfy the *less*[6] condition: `rootLabel` m is a string that is literally less than `rootLabel` n;
  - `children` m is structurally *less* than `children` n
- m and n are forests, and one of the following conditions exclusively holds,
  - m is `nil`, and n has at least one tree element;
  - their first element satisfies the structurally *less* ordering; that is, `head` m is `less` than `head` n;
  - their remaining elements satisfy the structurally *less* ordering, that is, `tail` m is `less` than `tail` n

From the definition of structurally less, for two objects of type Xtf, m and n, the function `less` is expressed in *System F* by

$$\begin{aligned}
\texttt{less}\ (\texttt{Xnode}\ s\ ts)\ n &= (s < \texttt{rootLabel}\ n)\ \texttt{or}\ (\texttt{less}\ ts\ (\texttt{children}\ n)) \\
\texttt{less}\ \texttt{nil}\ n &= \texttt{not}\ (\texttt{empty}\ n) \\
\texttt{less}\ (\texttt{Cons}\ t\ ts)\ n &= (\texttt{less}\ t\ (\texttt{head}\ n))\ \texttt{or}\ (\texttt{less}\ ts\ (\texttt{tail}\ n))
\end{aligned}$$

[6]The boolean operator *less*,"<", for strings can also be defined in *System F*.

where `not`, `or` are boolean operators defined in *System F*. The previous functions are rewritten as

$$\begin{aligned}
\texttt{less (Xnode } s\ ts) &= \lambda\, n^{Xtf}.\ (s < \texttt{rootLabel n})\ \texttt{or}\ (\texttt{less}\ ts\ (\texttt{children}\ n))\\
&= \mathsf{let}\ f_T = \lambda\, s^{String}.\ \lambda\, z^{Xtf\rightarrow Bool}.\ \lambda\ \ n^{Xtf}.\\
&\qquad (s < (\texttt{rootLabel n}))\ \texttt{or}\ (z\ (\texttt{children}\ n))\\
&\quad\ \mathsf{in}\ f_T\ \ s\ (\texttt{less}\ ts)
\end{aligned}$$

$$\begin{aligned}
\texttt{less nil} &= \texttt{not empty}\\
\texttt{less (Cons } t\ ts) &= \lambda\, n^{Xtf}.\ (\texttt{less}\ t\ (\texttt{head}\ n))\ \texttt{or}\ (\texttt{less}\ ts\ (\texttt{tail}\ n))\\
&= \mathsf{let}\ f_{XF} = \lambda\, x^{Xtf\rightarrow Bool}.\ \lambda\ \ y^{Xtf\rightarrow Bool}.\ \lambda\, n^{Xtf}.\\
&\qquad (x\ (\texttt{head}\ n))\ \texttt{or}\ (y\ (\texttt{tail}\ n))\\
&\quad\ \mathsf{in}\ f_{XF}\ \ (\mathsf{less}\ t)\ (\texttt{less}\ ts)
\end{aligned}$$

By using Equation(4.12), `less` is defined as follows:

$$\begin{aligned}
\texttt{less}\ &\overset{\text{def}}{=}\ \lambda\, m^{Xtf}.\ m\ [Xtf \rightarrow Bool]\\
&\qquad \mathsf{let}\ f_T = \lambda\, s^{String}.\ \lambda\, z^{Xtf\rightarrow Bool}.\ \lambda\ \ n^{Xtf}.\ (s < \texttt{rootLabel n})\ \texttt{or}\\
&\qquad\qquad\qquad (z\ (\texttt{children}\ n))\\
&\qquad\quad\ f_{XF} = \lambda\, x^{Xtf\rightarrow Bool}.\ \lambda\ \ y^{Xtf\rightarrow Bool}.\ \lambda\, n^{Xtf}.\ (x\ (\texttt{head}\ n))\ \texttt{or}\\
&\qquad\qquad\qquad (y\ (\texttt{tail}\ n))\\
&\qquad \mathsf{in}\ f_T\ \ (\texttt{not empty})\ f_{XF}
\end{aligned}$$

4.2.5 Application Operators

Since XML trees, forests, and their basic operators are represented by *System F* polymorphic functions, the focus is now on some complex functions such as `sort`: $\mathit{Xtf} \rightarrow (\mathit{Xtf} \rightarrow \mathit{Xtf} \rightarrow \mathit{Bool}) \rightarrow \mathit{Xtf}$, which sorts an XML forest using a tree structural boolean condition, and `distinct` : $\mathit{Xtf} \rightarrow \mathit{Xtf}$, which filters out all the duplicate tree elements from an XML forest.

sort For object ms of type Xtf and tree boolean condition f, `sort` $ms\ f$ reorders ms by using tree structural order function f, where `sort` recursively selects current tree element m in forest ms, compares m with the remaining elements of ms, and then inserts m into an appropriate position.

In order to express `sort`, a subsidiary function is required,

$$\texttt{insert} :\ \mathit{Xtf} \rightarrow (\mathit{Xtf} \rightarrow \mathit{Xtf} \rightarrow \mathit{Bool}) \rightarrow \mathit{Xtf} \rightarrow \mathit{Xtf}$$

where `insert` $m\ f\ ts$ inserts tree element m into an appropriate position of forest ts by tree boolean condition f. By using `insert`, `sort` is described as:

$$\begin{aligned}
\texttt{sort}\ f\ \texttt{nil} &= \texttt{nil} \\
\texttt{sort}\ f\ (\texttt{Cons}\ t\ ts) &= \texttt{insert}\ t\ (\texttt{sort}\ f\ ts) \\
&= \mathsf{let}\ f_{XF} = \lambda\, k^{\mathit{Xtf}}.\ \lambda\, z^{\mathit{Xtf}}.\ \texttt{insert}\ k\ z \\
&\quad\ \mathsf{in}\ f_{XF}\ (\texttt{sort}\ f\ t)\ (\texttt{sort}\ f\ ts)
\end{aligned}$$

`sort` is defined as a forest operator, which returns its second argument back when applied to a tree. These equations match the forms of the horizontal iteration Equations(4.10-4.11) with $f_{XF} : \mathit{Xtf} \rightarrow \mathit{Xtf} \rightarrow \mathit{Xtf}$.

`sort` is defined by Equation (4.12) as follows:

$$\begin{aligned}\texttt{sort} \overset{\text{def}}{=}\ & \lambda f^{Xtf \to Xtf \to Bool}.\ \lambda m^{Xtf}.\ m\ [Xtf] \\ & \textsf{let}\ f_T = \lambda s^{String}. \lambda z^{Xtf}. \texttt{Xnode}\ s\ ts \\ & \quad f_{XF} = \lambda k^{Xtf}.\ \lambda z^{Xtf}. \texttt{insert}\ t\ z \\ & \textsf{in}\ f_T\ \texttt{nil}\ f_{XF}\end{aligned}$$

The remaining task is to program the function

$$\texttt{insert}:\ Xtf \to (Xtf \to Xtf \to Bool) \to Xtf \to Xtf$$

where `insert` $m\ f\ n$ inserts tree m into forest n when boolean condition f is satisfied. `insert` behaves like a horizontal operator on forests, and is expressed as:

$$\begin{aligned}\texttt{insert}\ m\ f\ \texttt{nil} &= \texttt{Cons}\ m\ \texttt{nil} \\ \texttt{insert}\ m\ f\ (\texttt{Cons}\ t\ ts) &= \textsf{if}\ f\ m\ t\ \textsf{then}\ \texttt{Cons}\ m\ (\texttt{Cons}\ t\ ts) \\ & \quad\ \textsf{else}\ \texttt{Cons}\ t\ (\texttt{insert}\ m\ f\ ts)\end{aligned}$$

where the second equation does not match (4.11), and the same approach holds for the definition of `tail`. Let

$$g\ :\ Xtf \to (Xtf \to Xtf \to Bool) \to Xtf \to (Xtf \times Xtf)$$

such that $g\ m\ f\ ts = (ts,\ \texttt{insert}\ m\ f\ ts)$,

$$\begin{array}{rcl}
g\ m\ f\ \texttt{nil} & = & (\texttt{nil}, \texttt{Cons}\ m\ \texttt{nil}) \\
g\ m\ f\ (\texttt{Cons}\ t\ ts) & = & (\texttt{Cons}\ t\ ts,\ \textsf{if}\ f\ m\ t\ \textsf{then}\ \texttt{Cons}\ m\ (\texttt{Cons}\ t\ ts) \\
 & & \qquad\qquad \textsf{else}\ \texttt{Cons}\ t\ (\texttt{insert}\ m\ f\ ts)) \\
 & = & \lambda(k,\ z)^{Xtf \times Xtf}.(\texttt{Cons}\ t\ k,\ \textsf{if}\ f\ m\ t\ \textsf{then}\ \texttt{Cons}\ m \\
 & & \quad (\texttt{Cons}\ t\ k)\ \textsf{else}\ \texttt{Cons}\ t\ z)\ (ts,\ \texttt{insert}\ m\ f\ ts)) \\
 & = & \textsf{let}\ f_{XF} = \lambda\, x^{Xtf \times Xtf}.\ \lambda(k,\ z)^{Xtf \times Xtf}.\ (\texttt{Cons}\ t\ k, \\
 & & \quad \textsf{if}\ f\ m\ t\ \textsf{then}\ \texttt{Cons}\ m\ (\texttt{Cons}\ t\ k)\ \textsf{else}\ \texttt{Cons}\ t\ z) \\
 & & \textsf{in}\ f_{XF}\ (g\ m\ f\ t)\ (g\ m\ f\ ts)
\end{array}$$

These fit Equations (4.10-4.11). By employing (4.12), the following is defined:

$$\begin{array}{rcl}
g & \overset{\text{def}}{=} & \lambda\, m^{Xtf}.\ \lambda\, f^{Xtf \to Xtf \to Bool}.\ \lambda\, n^{Xtf}.\ n\ [Xtf \times Xtf] \\
 & & \quad \textsf{let}\ f_T = \lambda\, s^{String}.\ \lambda(k,\ z)^{Xtf \times Xtf}.\ (\texttt{Xnode}\ s\ k,\ \texttt{Xnode}\ s\ k) \\
 & & \qquad f_{XF} = \lambda\, x^{Xtf \times Xtf}.\ \lambda(k,\ z)^{Xtf \times Xtf}.(\texttt{Cons}\ t\ k, \\
 & & \qquad\qquad \textsf{if}\ f\ m\ t\ \textsf{then}\ \texttt{Cons}\ m\ (\texttt{Cons}\ t\ k)\ \textsf{else}\ \texttt{Cons}\ t\ z) \\
 & & \quad \textsf{in}\ f_T\ (\texttt{nil}, \texttt{Cons}\ m\ \texttt{nil})\ f_{XF}
\end{array}$$

and therefore, `insert` is the second element in the pair of type $[Xtf \times$

$Xtf]$, and

$$\begin{aligned}
\texttt{insert} \overset{\text{def}}{=} \ & \mathsf{let}\ g = \lambda\, m^{Xtf}.\ \lambda f^{Xtf\to Xtf\to Bool}.\ \lambda\, n^{Xtf}.\ n\ [Xtf \times Xtf] \\
& \quad \mathsf{let}\ f_T = \lambda\, s^{String}.\, \lambda(k,\ z)^{Xtf\times \texttt{Xtf}}.(\texttt{Xnode}\ s\ k,\ \texttt{Xnode}\ s\ k) \\
& \quad\quad f_{XF} = \lambda\, x^{Xtf\times Xtf}.\, \lambda(k,\ z)^{Xtf\times Xtf}.(\texttt{Cons}\ t\ k, \\
& \quad\quad\quad \mathsf{if}\ f\ m\ t\ \mathsf{then}\ \texttt{Cons}\ m\ (\texttt{Cons}\ t\ k)\ \mathsf{else}\ \texttt{Cons}\ t\ z) \\
& \quad \mathsf{in}\ f_T\ (\texttt{nil}, \texttt{Cons}\ m\ \texttt{nil})\ f_{XF} \\
& \mathsf{in}\ g.1
\end{aligned}$$

distinct For XML forest ts, `distinct` ts eliminates all the duplicate tree elements with $Xtf \to Xtf$. Given a function, `filter` : $(Xtf \to Xtf \to Bool) \to Xtf \to Xtf \to Xtf$, where `filter` $f\ m\ ts$ eliminates all tree elements t that satisfy boolean condition $f\ m\ t$, the horizontal function `distinct` is then expressed by `filter` so that

$$\begin{aligned}
\texttt{distinct nil} &= \texttt{nil} \\
\texttt{distinct}\ (\texttt{Cons}\ t\ ts) &= \texttt{Cons}\ t\ (\texttt{distinct}\ (\texttt{filter equal}\ t)\ ts) \\
&= \mathsf{let}\ \texttt{shift} = \lambda\, h^{\alpha\to\beta}.\, \lambda\, g^{\beta\to\gamma}.\, \lambda\, x^{\alpha}.\ g\ h\ x\ \mathsf{in} \\
&\quad\ \texttt{Cons}\ t\ \texttt{shift}\ (\texttt{filter equal}\ t)\ \texttt{distinct}\ ts \\
&= \mathsf{let}\ \texttt{shift} = \lambda\, h^{\alpha\to\beta}.\, \lambda\, g^{\beta\to\gamma}.\, \lambda\, x^{\alpha}.g\ h\ x\ \mathsf{in} \\
&\quad\ \mathsf{let}\ f^{Xf} = \lambda\, k^{Xtf}.\ \lambda\, z^{Xtf}. \\
&\quad\quad\quad \texttt{Cons}\ k\ \texttt{shift}\ (\texttt{filter equal}\ k)\ z \\
&\quad\ \mathsf{in}\ f_{XF}\ (\texttt{distinct}\ t)\ (\texttt{distinct}\ ts)
\end{aligned}$$

When `distinct` is applied to tree t, `distinct` returns t. Also, a general purpose function `shift` is needed to switch the order of the arguments such that

$$\begin{aligned}
&\texttt{shift}\ (\texttt{filter}\ \texttt{equal}\ t)\ \texttt{distinct}\ \mathit{ts}\\
&\quad=\ (\lambda\, h^{\alpha\to\beta}.\ \lambda\, g^{\beta\to\gamma}.\ \lambda\, x^{\alpha}.\ g\ h\ x)\ (\texttt{filter}\ \texttt{equal}\ t)\ \texttt{distinct}\ \mathit{ts}\\
&\quad\leadsto\ \texttt{distinct}\ (\texttt{filter}\ \texttt{equal}\ t)\ \mathit{ts}
\end{aligned}$$

With `switch`, `distinct` is signified as

$$\begin{aligned}
\texttt{distinct}\ \overset{\text{def}}{=}\ & \lambda\, m^{\mathit{Xtf}}.\ m\ [\mathit{Xtf}]\\
& \mathsf{let}\ f_T = \lambda\, s^{\mathit{String}}.\lambda\, z^{\mathit{Xtf}}.\ \texttt{Xnode}\ s\ \mathit{ts}\\
& \quad f_{XF} = \mathsf{let}\ \texttt{shift} = \lambda\, h^{\alpha\to\beta}.\ \lambda\, g^{\beta\to\gamma}.\ \lambda\, x^{\alpha}.\ g\ h\ x\ \mathsf{in}\\
& \quad\quad \lambda\, k^{\mathit{Xtf}}.\ \lambda\, z^{\mathit{Xtf}}.\ \texttt{Cons}\ k\ \texttt{shift}\ (\texttt{filter}\ \texttt{equal}\ k)\ z\\
& \mathsf{in}\ f_T\ \texttt{nil}\ f_{XF}
\end{aligned}$$

The definition of `distinct` still requires operator `filter` : $(\mathit{Xtf} \to \mathit{Xtf} \to \mathit{Bool}) \to \mathit{Xtf} \to \mathit{Xtf} \to \mathit{Xtf}$, where `filter` $f\ m\ \mathit{ts}$ eliminates the tree elements that satisfy boolean condition $f\ m\ t$. Operator `distinct` is then described as,

$$\begin{aligned}
\texttt{filter}\ f\ b\ \texttt{nil} &= \texttt{nil}\\
\texttt{filter}\ f\ b\ (\texttt{Cons}\ t\ \mathit{ts}) &= \mathsf{if}\ f\ b\ t\ \mathsf{then}\ \texttt{filter}\ f\ b\ \mathit{ts}\\
&\quad\ \mathsf{else}\ \texttt{Cons}\ t\ (\texttt{filter}\ f\ b\ \mathit{ts})\\
&\quad\ \mathsf{let}\ f_{XF} = \lambda\, k^{\mathit{Xtf}}.\ \lambda\, z^{\mathit{Xtf}}.\ \mathsf{if}\ f\ b\ k\ \mathsf{then}\ z\\
&\quad\quad\quad\quad \mathsf{else}\ \texttt{Cons}\ k\ z\\
&\quad\ \mathsf{in}\ f_{XF}\ (\texttt{filter}\ f\ b\ t)\ (\texttt{filter}\ f\ b\ \mathit{ts})
\end{aligned}$$

As a horizontal forest operator, `filter` returns only the second argument, if it is a tree. The previous equations match (4.10-4.11), and are defined by Equation (4.12) with type $f_{XF} : \mathit{Xtf} \to \mathit{Xtf} \to \mathit{Xtf}$ such that

$$\begin{aligned}
\texttt{filter} \overset{\text{def}}{=}\ & \lambda f^{\mathit{Xtf} \to \mathit{Xtf} \to \mathit{Bool}}.\ \lambda b^{\mathit{Xtf}}.\ \lambda m^{\mathit{Xtf}}.\ m\ [\mathit{Xtf}] \\
& \mathsf{let}\ f_T = \lambda s^{\mathit{String}}.\lambda z^{\mathit{Xtf}}.\ \texttt{Xnode}\ s\ ts \\
& \quad f_{XF} = \lambda k^{\mathit{Xtf}}.\ \lambda z^{\mathit{Xtf}}.\ \mathsf{if}\ f\ b\ k\ \mathsf{then}\ z \\
& \qquad\qquad \mathsf{else}\ \texttt{Cons}\ k\ z \\
& \mathsf{in}\ f_T\ \texttt{nil}\ f_{XF}
\end{aligned}$$

4.3 Translation of XQ to *System F*

After the XML tree and forest are represented in *System F*, the syntax definition of XQ expressions is rewritten into more general form, where the FLWR keyword allows the arbitrary composition of query expressions; and also, all the basic operators on XML data are presented in *System F*. Encoded in *System F*, a complex XML query expression can be expressed as a sequence of function applications without any special purpose operators. The syntax of XQ in *System F* is defined in Figure 4.2, where $\mathsf{XF_n}$ expresses the basic XML operators defined as polymorphic function *abstractions*, x is a variable, φ is a boolean expression, U is a type, and $e_{1_{SF}}$, $e_{2_{SF}}$, …, $e_{k_{SF}}$ denote XQ expressions encoded in *System F*. For example,the `head` is represented in *System F* as

$$\begin{aligned}
\texttt{head}\ m \overset{\text{def}}{=}\ & m\ [\mathit{Xtf}]\ (\lambda s^{\mathit{String}}.\lambda b^{\mathit{Xtf}}.\texttt{Xnode}\ s\ ts)\ \texttt{default} \\
& \qquad (\lambda t^{\mathit{Xtf}}.\lambda z^{\mathit{Xtf}}.\ t)
\end{aligned}$$

$$
\begin{array}{rcl}
\mathit{XQexp} & ::= & x \\
& | & \mathsf{let}\ x = e_{SF}\ \mathsf{in}\ e'_{SF} \\
& | & \mathsf{for}\ x\ \mathsf{in}\ e_{1_{SF}}\ \mathsf{do}\ e_{2_{SF}} \\
& | & \mathsf{where}\ \varphi\ \mathsf{return}\ e_{SF} \\
& | & \mathsf{XF_n}\ e_{1_{SF}}\ \ldots e_{k_{SF}} \\
\mathit{Boolexp} & ::= & \mathsf{True}\ |\ \mathsf{False} \\
& | & \texttt{empty}\ e_{SF} \\
& | & \texttt{less}\ e_{SF}\ e'_{SF} \\
& | & \texttt{equal}\ e_{SF}\ e'_{SF} \\
\mathsf{XF_n}\ e_{1_{SF}}\ e_{2_{SF}}\ \ldots\ e_{k_{SF}} & ::= & e_{SF}\ [U]\ e_T\ e_{nil}\ e_{XF}
\end{array}
$$

Figure 4.2: XQ in *System F*

where "[*Xtf*]" denotes the resulting type of `head`, and the following three terms, e_T, e_{nil}, and e_{XF}, correspond iterators inductively applied to `Xnode`, `nil` and `Cons`.

4.4 Semantic Analysis

Since XQ has been encoded in *System F*, the semantics of XQ query expressions are described as semantic equations, and encoded in *System F*.

4.4.1 Environment

To understand the semantics of XQ expressions, it is defined *semantic* equations that map XQ expressions to the meanings that these phrases denote. Clearly, XQ expressions have corresponding values which are XML forests, similar to those of boolean expressions that have boolean values True or False. However, the *meanings* or *denotations* of XQ expressions are much more complex than the values, because the values of XQ expressions depend on the values of the variables of the XQ expressions. More abstractly, the values depend on an *environment*, *Env*, which maps each variable to its corresponding value.

Definition 20 (Environments) An environment, *Env*, denotes the mapping of each variable with its value, and is described as a list of paired variables and values with the following definitions:

$$\begin{aligned} \mathit{Env} &\overset{\text{def}}{=} (\mathit{Var} \times \mathit{Xtf})\ \texttt{list} \\ \mathit{Var} \times \mathit{Xtf} &\overset{\text{def}}{=} (\mathit{Var} \to \mathit{Xtf} \to \mathit{Pair}) \to \mathit{Pair} \end{aligned}$$

For $v : \mathit{Var}$, $ts : \mathit{Xtf}$, pair $(v,\ ts) : \mathit{Var} \times \mathit{Xtf}$ is expressed by the following polymorphic function:

$$(v,\ ts) \overset{\text{def}}{=} \lambda f^{\mathit{Var} \to \mathit{Xtf} \to \mathit{Pair}}.\ f\ \ v\ \ ts$$

and the project operations are

$$\begin{aligned} (v,\ ts).0 &= (v,\ ts)\ [\mathit{Var}]\ (\lambda x^{\mathit{Var}}.\ \lambda y^{\mathit{Xtf}}.\ x) \\ (v,\ ts).1 &= (v,\ ts)\ [\mathit{Xtf}]\ (\lambda x^{\mathit{Var}}.\ \lambda y^{\mathit{Xtf}}.\ y) \end{aligned}$$

where .0 returns the first element, variable v, in the pair, and .1 receives value ts, shown as

$$\begin{aligned}
(v,\ ts).0 &\overset{\text{def}}{=} (v,\ ts)\ [\mathit{Var}]\ (\lambda\, x^{\mathit{Var}}.\ \lambda\, y^{\mathit{Xtf}}.\ x) \\
&= (\lambda\, f^{\mathit{Var}\to\mathit{Xtf}\to\mathit{Pair}}.\ f\ \ v\ \ ts)\ [\mathit{Var}]\ (\lambda\, x^{\mathit{Var}}.\ \lambda\, y^{\mathit{Xtf}}.\ x) \\
&\rightsquigarrow (\lambda\, f^{\mathit{Var}\to\mathit{Xtf}\to\mathit{Var}}.\ f\ \ v\ \ ts)\ (\lambda\, x^{\mathit{Var}}.\ \lambda\, y^{\mathit{Xtf}}.\ x) \\
&\rightsquigarrow (\lambda\, x^{\mathit{Var}}.\ \lambda\, y^{\mathit{Xtf}}.\ x)\ (\lambda\, x^{\mathit{Var}}.\ \lambda\, y^{\mathit{Xtf}}.\ x)\ v\ \ ts \\
&\rightsquigarrow (\lambda\, y^{\mathit{Xtf}}.\ v)\ \ ts \\
&\rightsquigarrow v
\end{aligned}$$

and similarly,

$$(v,\ ts).1 \rightsquigarrow^{*} ts$$

4.4.2 Operations of the Environment

The *environment* describes the mapping of variable v to its corresponding value ts. In order to operate the values stored in the environment, two operations must be defined to manipulate the environment.

put Operation `put` $v\ ts\ env$ mapping value ts to variable v in environment env. If v already exists and stores value ts', ts' is replaced by new value ts; otherwise, a new mapping $(v,\ ts)$ is added to env. The operation `put` : $\mathit{Var} \to\ \mathit{Xtf} \to \mathit{Env} \to \mathit{Env}$ is expressed as

$$\begin{aligned}
\texttt{put}\ v\ ts\ \texttt{nil} &= \texttt{Cons}\ (v,\ ts)\ \texttt{nil} \\
\texttt{put}\ v\ ts\ (\texttt{Cons}\ a\ as) &= \textsf{if}\ v == a.0\ \textsf{then}\ \texttt{Cons}\ (v,\ ts)\ as \\
&\quad\ \textsf{else}\ \texttt{Cons}\ a\ (\texttt{put}\ v\ ts\ as)
\end{aligned}$$

where *env* is the type of $(\mathit{Var} \times \mathit{Xtf})$ `list`. To define the function `put`, a function g,

$$g : \mathit{Var} \rightarrow \ \mathit{Xtf} \rightarrow \mathit{Env} \rightarrow (\mathit{Env} \times \mathit{Env})$$

with $g\ v\ ts\ m = (m,\ \texttt{put}\ v\ ts\ m)$, is introduced.

For variable $v : \mathit{Var}$ and value $ts : \mathit{Xtf}$, g is expressed as

$$\begin{aligned}
g\ v\ ts\ \texttt{nil} &= (\texttt{nil},\ \texttt{Cons}\ (v,\ ts)\ \texttt{nil}) \\
g\ v\ ts\ (\texttt{Cons}\ a\ as) &= (\texttt{Cons}\ a\ as,\ \textsf{if}\ v == a.0 \\
&\qquad \textsf{else}\ \texttt{Cons}\ a\ (\texttt{put}\ v\ ts\ as)) \\
&= \lambda(k,\ z)^{\mathit{Env}\times\mathit{Env}}.\ (\texttt{Cons}\ a\ k,\ \textsf{if}\ v == a.0 \\
&\qquad \textsf{then}\ \texttt{Cons}\ (v,\ ts)\ k\ \textsf{else}\ \texttt{Cons}\ a\ z) \\
&\qquad a\ (as,\ \texttt{Cons}\ a\ (\texttt{put}\ v\ ts\ as)) \\
&= \textsf{let}\ f = \lambda\, a^{\mathit{Var}\times\mathit{Xtf}}.\ \lambda(k,\ z)^{\mathit{Env}\times\mathit{Env}}.\ (\texttt{Cons}\ a\ k, \\
&\quad\ \textsf{if}\ v == a.0\ \textsf{then}\ \texttt{Cons}\ (v,\ ts)\ k\ \textsf{else}\ \texttt{Cons}\ a\ z \\
&\quad \textsf{in}\ f\ \ a\ (g\ v\ ts\ as)
\end{aligned}$$

and matches the forms of list Equation (3.6), and is defined as

$$\begin{aligned}
g \ \overset{\text{def}}{=}\ & \lambda\, v^{\mathit{Var}}.\ \lambda\, ts^{\mathit{Xtf}}.\ \lambda\, m^{\mathit{Env}}.\ m\ [\mathit{Env} \times \mathit{Env}] \\
& \textsf{let}\ f = \lambda\, a^{\mathit{Var}\times\mathit{Xtf}}.\ \lambda(k,\ z)^{\mathit{Env}\times\mathit{Env}}.\ (\texttt{Cons}\ a\ k,\ \textsf{if}\ v == a.0 \\
& \qquad\qquad \textsf{then}\ \texttt{Cons}\ (v,\ ts)\ k\ \textsf{else}\ \texttt{Cons}\ a\ z \\
& \textsf{in}\ (\texttt{nil},\ \texttt{Cons}\ (v,\ ts)\ \texttt{nil})\ f
\end{aligned}$$

and function `put` is the second element of g, so $\texttt{put} \overset{\text{def}}{=} g.1$

get Operation $\texttt{get}\ v\ \ env$ retrieves the value of variable v stored in the environment env if v exists; otherwise, $\texttt{get}$ returns nil, where $\texttt{get}$ has type $\texttt{get} : \mathit{Var} \rightarrow\ \mathit{Env} \rightarrow \mathit{Xtf}$, and is described as follows:

$$\begin{aligned}
\texttt{get}\ v\ \texttt{nil} &= \mathsf{nil} \\
\texttt{get}\ v\ (\texttt{Cons}\ a\ as) &= \mathsf{if}\ v == a.0\ \mathsf{then}\ a.1\ \mathsf{else}\ \texttt{get}\ v\ as \\
&= \mathsf{let}\ f = \lambda\, a^{\mathit{Var}\times\mathit{Xtf}}.\ \lambda\, z^{\mathit{Xtf}}.\ \mathsf{if}\ v == a.0\ \mathsf{then}\ a.1 \\
&\qquad\qquad \mathsf{else}\ z \\
&\quad\ \mathsf{in}\ f\ \ a\ (\texttt{get}\ v\ as)
\end{aligned}$$

which fits the iterator form of list in Definition 11. Consequently, $\texttt{get}$ is denoted as

$$\begin{aligned}
\texttt{get} &\overset{\text{def}}{=} \lambda\, v^{\mathit{Var}}.\ \lambda\, m^{\mathit{Env}}.\ m\ [\mathit{Xtf}] \\
&\quad\ \mathsf{let}\ f = \lambda\, a^{\mathit{Var}\times\mathit{Xtf}}.\ \lambda\, z^{\mathit{Xtf}}.\ \mathsf{if}\ v == a.0\ \mathsf{then}\ a.1\ \mathsf{else}\ z \\
&\quad\ \mathsf{in}\ \mathsf{nil}\ f
\end{aligned}$$

4.4.3 Semantic Functions of XQ

With defining *Env* and its operations in *System F*, an XQ query evaluation can be described as a set of computations from the initial state (environment) to the final stat(environment) by using *state-transformation* functions [7],

$$\begin{aligned}
[\![\,-\,]\!]_{\mathit{XQexp}}E &= \mathit{XQexp} \rightarrow \mathit{Env}\ \rightarrow \mathit{Xtf} \\
[\![\,-\,]\!]_{\mathit{Boolexp}}E &= \mathit{Boolexp} \rightarrow \mathit{Env}\ \rightarrow \mathit{Bool}
\end{aligned}$$

where E is an environment of type *Env*.

The semantics of XQ expressions are indeed a sequence of operations that

[7]The expression type is added to the semantic functions.

change the environments. In this thesis, the intention is to represent these semantic functions and describe them in *System F*.

SEM EQ: Variables

$$[\![v]\!]_{XQexp}E \overset{\text{def}}{=} E\ v = \texttt{get}\ v\ E$$

which means that the variable expression returns its value in environment E.

SEM EQ: Let-Assignment

$$\begin{aligned}[\![\mathsf{let}\ x = e\ \mathsf{in}\ e'\ ]\!]_{XQexp}E &\overset{\text{def}}{=} [\![e']\!]_{XQexp}(E[x := ([\![e]\!]_{XQexp}E)] \\ &= (\lambda\, E'^{Env}.\ [\![e']\!]_{XQexp}E')\ (\lambda f^{Env\to Env}.\ f\ E) \\ &\quad (\texttt{put}\ x\ [\![e]\!]_{XQexp}E) \\ &= \mathsf{let}\ v' = [\![e]\!]_{XQexp}E \\ &\qquad f = \lambda\, v'^{Xtf}.\ \texttt{put}\ x\ v' \\ &\quad \mathsf{in}\ [\![e']\!](f\ v'\ E)\end{aligned}$$

The meaning of $\mathsf{let}\ x := e\mathsf{in}\ e'$ is that, after e is evaluated in E obtaining a value v', $\texttt{put}\ x\ v'$ changes environment E to E' by mapping x to v, and then expression e' is evaluated in new environment E'.

SEM EQ: Boolean Condition

$$\begin{aligned}[\![\texttt{equal}\ e\ e'\ ]\!]_{Boolexp}E &\overset{\text{def}}{=} \texttt{equal}\ ([\![e]\!]_{XQexp}E)\ ([\![e']\!]_{XQexp}E) \\ [\![\texttt{less}\ e\ e'\ ]\!]_{Boolexp}E &\overset{\text{def}}{=} \texttt{less}\ ([\![e]\!]_{XQexp}E)\ ([\![e']\!]_{XQexp}E) \\ [\![\texttt{empty}\ e\ ]\!]_{Boolexp}E &\overset{\text{def}}{=} \texttt{empty}\ ([\![e]\!]_{XQexp}E)\end{aligned}$$

The purpose of the boolean condition is to evaluate each *XQexp* expression in environment E, and then check the boolean relationship of results.

SEM EQ: Where-Return

$$[\![\texttt{where}\ \varphi\ \mathsf{return}\ e]\!]_{XQexp}E \overset{\text{def}}{=} \mathsf{if}\ ([\![\varphi]\!]_{Boolexp}E)\ \mathsf{then}\ ([\![e]\!]_{XQexp}E)$$
$$\mathsf{else\ nil}$$

where φ is a boolean condition whose semantic equation is defined above. If condition φ holds, *XQexp* expression e is evaluated in environment E, and the result of type *Xtf* is returned.

SEM EQ: For-Do

$$[\![\mathsf{for}\ x\ \in\ e\ \mathsf{do}\ e']\!]_{XQexp}E \overset{\text{def}}{=} [\![e']\!]_{XQexp}(E[x := v_1])\ @\ldots @$$
$$[\![e']\!]_{XQexp}E([x := v_k])$$
$$\text{where } [v_1,\ \ldots,\ v_k] = [\![e]\!]E$$

The semantic function of For-Do is complex. The operation for $x\ \in\ e$ do e' first evaluates expression e in environment E, returning a resulting list $[v_1,\ \ldots,\ v_k]$, and then '$x\ \in$" transfers environment E to be a list of new environments $[E_1,\ \ldots,\ E_k]$. Finally expression e' is evaluated on the list of the new environments.

In order to encode the For-Do semantic equation, a *mapping* function is employed which has

$$\texttt{mapf} : (\theta \to \theta') \to \theta\ \mathsf{list}\ \to \theta'\ \mathsf{list}$$

whose definition is

$$\texttt{mapf} \overset{\text{def}}{=} \lambda\, f^{\theta\to\theta'}.\, \lambda\, x^{\theta\ \mathsf{list}}.x\ [\theta'\ \mathsf{list}]\ \texttt{nil}\ (\lambda\, t^{\theta}.\, \lambda\, z^{\theta'\ \mathsf{list}}.\texttt{Cons}\ (f\ \ t)\ z)$$

which applies function f to each element in list x.

For the For-Do clause, there are a set of state-transformation actions:

- the evaluation of expression e in environment E creates a *Xtf* list for x

$$\mathit{newVals} = [\![e]\!]E\ \ : \mathit{Xtf}\ \mathsf{list}$$

- `mapPut` $x\ \mathit{newVals}\ E$ creates a list of new environments *newEs* by applying `put` $x\ \ e\ \ E$ to value e in the *newVals* list, where

$$\begin{array}{lcl} \texttt{mapPut} & = & \lambda\, x^{\mathit{Var}}.\texttt{mapf}\ (\texttt{put}\ x) \\ & = & \lambda\, x^{\mathit{Var}}.\, \lambda\, \mathit{newVals}^{\mathit{Xtf}\ \mathsf{list}}.\, \lambda\, E^{\mathit{Env}}.\ \mathit{newVals}\ [\mathit{Env}\ \mathsf{list}] \\ & & \qquad\quad \texttt{nil}\ (\lambda\, e^{\mathit{XQexp}}.\ \ \lambda\, z^{\mathit{Env}\ \mathsf{list}}.\ \texttt{Cons}\ (\texttt{put}\ x\ \ e\ \ E)\ z) \\ \texttt{mapPut} & : & \mathit{Var} \to \mathit{Xtf}\ \mathsf{list} \to \mathit{Env} \to \mathit{Env}\ \mathsf{list} \end{array}$$

- `mapSem` $e'\ \ E'$ evaluates e' in each environment in *newEs* list, and concatenates the results:

$$\begin{array}{lcl} \texttt{mapSem} & = & \lambda\, e'^{\mathit{XQexp}}.\ \texttt{mapf}\ \ ([\![e']\!]) \\ & = & \lambda\, e'.\ \ \lambda\, \mathit{newEs}^{\mathit{Env}\ \mathsf{list}}.\ \mathit{newE}\ [\mathit{Xtf}\ \mathsf{list}]\ \texttt{nil} \\ & & (\lambda\, E^{\mathit{Env}}.\ \ \lambda\, z^{\mathsf{list}\ \mathit{Xtf}}.\ \texttt{Concat}\ ([\![e']\!]E)\ z) \\ \texttt{mapSem} & : & \mathit{XQexp} \to \mathit{Env}\ \mathsf{list} \to \mathit{Xtf}\ \mathsf{list} \end{array}$$

With these functions, the semantic equation for `For-Do` is defined as

$$
\begin{aligned}
&[\![\mathsf{for}\ x\ \in\ e\ \mathsf{do}\ e']\!]_{XQexp} E \overset{\text{def}}{=} \\
&\quad \mathsf{let}\ newVals = [\![e]\!] E \\
&\qquad \mathtt{mapPut} = \lambda\, x^{Var}.\ \lambda\, newVals^{Xtf\ \mathsf{list}}.\ \lambda\, E^{Env}.\ newVals\ [Env\ \mathsf{list}] \\
&\qquad\quad \mathtt{nil}\ (\lambda\, e^{XQexp}.\ \lambda\, z^{Env\ \mathsf{list}}.\ \mathtt{Cons}\ (\mathtt{put}\ x\ e\ E)\ z) \\
&\qquad \mathtt{mapSem} = \lambda\, e'^{XQexp}.\ \lambda\, newEs^{Env\ \mathsf{list}}.\ newE\ [Xtf\ \mathsf{list}] \\
&\qquad\quad \mathtt{nil}\ (\lambda\, E^{Env}.\ \lambda\, z^{Xtf\ \mathsf{list}}.\ \mathtt{Concat}\ ([\![e']\!] E)\ z) \\
&\quad \mathsf{in}\ \mathtt{mapSem}\ e'\ (\mathtt{mapPut}\ x\ newVals)
\end{aligned}
$$

SEM EQ: $\mathtt{XF_n}$

$$[\![\mathtt{XF_n}(e_1, \ldots, e_k)]\!] E \quad \overset{\text{def}}{=} \quad \mathtt{XF_n}\ ([\![e_1]\!] E, \ldots, [\![e_k]\!] E)$$

where $\mathtt{XF_n}$ is an XML operator, such as `select`, `subtreesdfs`, with k arguments. $[e_1, \ldots, e_k]$ is a expression list, es, where e_i is an *XQexp* expression. The argument list, es, is first evaluated in environment E, and the resulting list vs is supplied to $\mathtt{XF_n}$ as arguments. First, `mapArg` is defined as

$$
\begin{aligned}
\mathtt{mapArg} \quad &= \quad \mathtt{mapf}\ (\lambda\, e^{XQexp}.\, \lambda\, E^{Env}.[\![e]\!] E) \\
&= \quad (\lambda\, f^{\theta \to \theta'}.\, \lambda\, x^{\theta\ \mathsf{list}}.x\ [\theta'\ \mathsf{list}]\ \mathtt{nil}\ (\lambda\, t^{\theta}.\, \lambda\, z^{\mathsf{list}\ \theta'}.\mathtt{Cons}\ (f\ t)\ z)) \\
&\qquad (\lambda\, e^{XQexp}.\, \lambda\, E^{Env}.[\![e]\!] E) \\
&= \quad \lambda\, argList^{XQexp\ \mathsf{list}}.\ \lambda\, E^{Env}.argList\ [Xtf\ \mathsf{list}] \\
&\qquad\qquad \mathtt{nil}\ (\lambda\, e^{XQexp}.\, \lambda\, z^{Xtf\ \mathsf{list}}.\ \mathtt{Cons}\ ([\![e]\!] E)\ z) \\
\mathtt{mapArg} \quad &: \quad XQexp\ \mathsf{list} \to Env \to Xtf\ \mathsf{list}
\end{aligned}
$$

where `mapArg` $es\ E$ creates an *Xtf* list by applying $[\![e]\!]E$ to each *XQexp* expression e in expression list es. Finally, the semantic meaning of $\mathtt{XF_n}$ is defined as

$$
\begin{aligned}
&[\![\mathtt{XF_n}(e_1, \ldots, e_k)]\!]E \stackrel{\text{def}}{=} \\
&\quad \mathsf{let}\ argList = [e_1, \ldots, e_k] \\
&\qquad \mathtt{mapArg} = \lambda\, argList^{XQexp\ \mathsf{list}}.\, \lambda\, E^{Env}.\ argList\ [Xtf\ \mathsf{list}] \\
&\qquad\quad \mathtt{nil}\ (\lambda\, e^{XQexp}.\, \lambda\, z^{Xtf\ \mathsf{list}}.\mathtt{Cons}\ ([\![e]\!]E)\ z) \\
&\quad \mathsf{in}\ \mathtt{XF_n}(\mathtt{mapArg}\ argList\ E)
\end{aligned}
$$

So for, the XML data and all the basic operators defined in XQ are represented successfully by *System F*. As a query language, XQuery has been proved to be turing complete, which means its query evaluation termination is undecidable. As an XQuery *fragment*, XQ has been shown to be *System F representable*, which gives XQ a very important computability property.

Theorem 4 Any XQ query evaluation terminates and the equivalence of XQ queries is decidable.

This property is supported by the *strong normalization* property of *System F* shown in Chapter 3. As an XML query language, XQ query's computation decidability does not contradict the undecidability of the relational query language because an XML document is ordered.

Chapter 5

Extension of XQ

The principal purpose of the dynamic interval method is to implement the XML query by a relation query engine, which ensures the polynomial time complexity in the query evaluation. After this is achieved, it is time to discuss the expressiveness of XQ language. Not only is its expressiveness investigate, but also its limitations are studied by examining simple queries. Although they cannot be defined in XQ language, they still can be expressed in relational SQL statements, leading us to the extension. In this chapter, first a simple query that is XQ undefinable is given, then an extension is presented to overcome this limitation, and retain XQ's *System F* encodable property to guarantee the *strong* query evaluation termination.

5.1 Limitation of XQ: A Case Study

As an XQuery *fragment*, XQ captures the core feature presented in XQuery/XPath expressions. XQ's simple syntax allows arbitrary compositions of basic function invocations, local variable definitions, conditional selections, and iterations over XML data. Although XQ is powerful, it still cannot express certain simple queries.

Figure 5.1 illustrates two XML queries, where query Q_1 asks to flat a multilevel list-like XML tree[1] A into a two-level XML tree B, and query Q_2 conversely transforms XML tree B back to tree A.

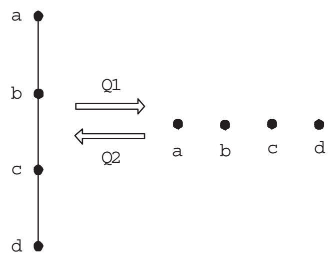

Figure 5.1: Two XML queries examples

The query Q_1 is represented in XQ syntax as

$$\texttt{roots}\ (\texttt{subtreesdfs}\ A)$$

where `subtreedfs` puts all the sub-self trees of A into a forest in the document(tree) order, and the `roots` extracts the root nodes for all the sub-trees. The purpose of function composition `roots` $\circ$ `substreedfs` is to flat a complex multilevel hierarchical tree into a forest with each node at the same top level. In addition, the XQ expression of query Q_1 can be translated into an SQL statement with the XQ-to-SQL templates of `subtreedfs`, `roots`, defined in [6], and evaluated by a relational query engine. Now query Q_2 picks up all the nodes of tree B and constructs a deep tree without *sibling* relationships. However, Q_2 cannot be expressed by *XQ*, even though the

[1]Tree and forest are used to describe the XML document in this chapter.

s	l	r
a	0	1
b	2	3
c	4	5
d	6	7

Figure 5.2: Table T_B

s	l	r
a	0	14
b	2	12
c	4	10
d	6	8

Figure 5.3: $T_{verticalTree}$ from Tree B

query still can be expressed by the SQL statement after tree B is *interval encoded* into a relation.

5.1.1 An XQ Undefinable Query in SQL

The implementation of query Q_2 by the relational method includes two steps:

- encoding XML tree B into an interval relation by performing a depth first traversal on the tree B and marking each node's left and right end points, as depicted in Figure 5.2;
- constructing a tree by linking all the nodes of tree B using the parent-child relationship with the document order, shown in Figure 5.4.

5.1.2 Analysis of the XQ Limitations

The reason of Q_2 is definable in relational SQL queries is that, when tree B is encoded into relation, the node tuples can be accessed arbitrarily, and their parent-children, ancestor-descendent, and sibling relationship can be reassembled by changing the value of interval attributes l and r. For tree B $[a, b, c, d]$ (expressed simply), by using a SQL statement, query Q_2 can be implemented by extending the right interval value of c to contain the right

```
SELECT $u.s$ AS $s$, $u.l$ AS $l$, $2 * v.r - u.l$ AS $r$
FROM $T_B$ $u$,
        (SELECT $v.s$, $v.l$, $v.r$
        FROM $T_B$ $v$
        WHERE NOT EXISTS(
        SELECT $*$
        FROM $T_B$ $m$
        WHERE $m.l > v.r$) $v$
```

Figure 5.4: SQL Expression for Constructing a Vertical Tree

interval of node d to obtain a new tree with c being the parent of d, and so on. With the interval relation, the *vertical* tree is successfully constructed without sibling relationship.

After the XQ *undefinable* query is expressed in SQL statements, we would discuss the limitations of XQ language, that is, why can XQ not express such a query as Q_2? With analyzing the syntax of XQ in Chapter 2, we find that only a *horizontal* tree constructor `Xnode` is defined, which creates a new tree using string s and forest ts, denoted as `<s> ts </s>`. The `Xnode` expresses only the tree construction on the top level, with wrapping up a list of of XML elements by tag s.

Fact 1 Only with the tree constructor `Xnode`, XQ cannot express the nesting queries whose results are much deeper than the original document, except enumerating each individual element.

In the thesis, the objective is to extend XQ by adding a new tree operator that constructs a tree in *vertical* direction.

5.2 New Tree Constructor `Xtree`

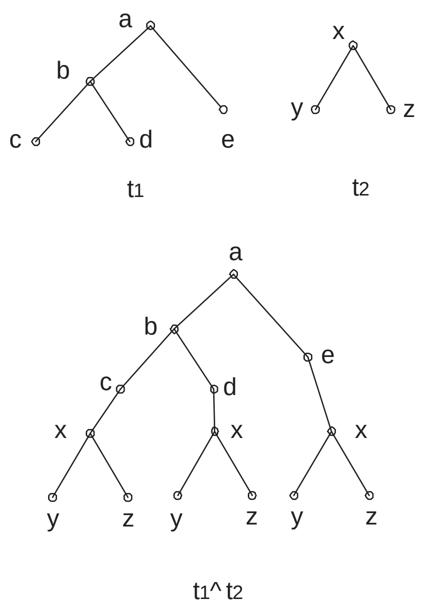

Figure 5.5: The operation of `Xtree` t_1 t_2

An example of a vertical tree construction is offered in Figure 5.5. The new tree operator is named `Xtree`, where `Xtree` t_1 t_2 appends tree t_2 to each leaf node of tree t_1. After extending XQ syntax by `Xtree`, `Xtree` will be represented in *System F* to ensure this extension retains the strong normalization property of XQ.

5.2.1 Representing `Xtree` in *System F*

As a vertical tree operation, `Xtree` t_1 t_2 constructs a new tree by appending t_2 to each leaf node of tree t_1 as

$$\begin{aligned}
\texttt{Xtree}\ (\texttt{Xnode}\ s\ ts)\ t_2 &= \textsf{if}\ \texttt{empty}\ ts\ \textsf{then}\ \texttt{Xnode}\ s\ t_2 \\
&\quad\ \textsf{else}\ \texttt{Xnode}\ s\ (\texttt{Xtree}\ ts\ t_2) \qquad (5.1)\\
\texttt{Xtree}\ \texttt{nil}\ t_2 &= \texttt{nil} \qquad (5.2)\\
\texttt{Xtree}\ (\texttt{Cons}\ t\ ts)\ t_2 &= \texttt{Cons}\ (\texttt{Xtree}\ t\ t_2)\ (\texttt{Xtree}\ ts\ t_2) \qquad (5.3)
\end{aligned}$$

If t_1 is a tree as `Xnode` s ts, and if ts is [ ], then t_2 is directly appended as the children of root s as `<s>` t_2 `</s>`; otherwise, the iterator visits t_1 children's level, which is XML forest ts, and applies operator `Xtree` to each tree element of ts.

Equation (5.1) does not match (4.9) because of ts is employed as an argument of `empty`. It is necessary to define a pair function g to store the input argument and the corresponding result of `Xtree` as follows:

$$g\ t_1\ t_2 \overset{\text{def}}{=} (t_1,\ \texttt{Xtree}\ t_1\ t_2)$$

with the following details:

$$\begin{aligned}
g\ (\texttt{Xnode}\ s\ ts)\ t_2 &= (\texttt{Xnode}\ s\ ts,\ \textsf{if}\ \texttt{empty}\ ts\ \textsf{then}\ \texttt{Xnode}\ s\ t_2 \\
&\qquad\qquad \textsf{else}\ \texttt{Xnode}\ s\ (\texttt{Xtree}\ ts\ t_2)) \qquad (5.4)\\
g\ \texttt{nil}\ t_2 &= (\texttt{nil},\ \texttt{nil})\\
g\ (\texttt{Cons}\ t\ ts)\ t_2 &= (\texttt{Cons}\ t\ ts, \texttt{Cons}\ (\texttt{Xtree}\ t\ t_2)\ (\texttt{Xtree}\ ts\ t_2))
\end{aligned}$$

In order to match the form of (4.9), with $g\ t_1\ t_2 = (t_1,\ \texttt{Xtree}\ t_1\ t_2)$, (5.4) is rewritten as

$$\begin{aligned}
g\ (\texttt{Xnode}\ s\ ts)\ t_2 &= (\lambda\, s^{string}.\, \lambda(k,\ z)^{Xtf \times Xtf}.(\texttt{Xnode}\ s\ k,\ \textsf{if empty}\ k \\
&\quad \textsf{then}\ \texttt{Xnode}\ s\ t_2\ \textsf{else}\ \texttt{Xnode}\ s\ z))\ s\ (ts,\ \texttt{Xtree}\ ts\ t_2) \\
&= \textsf{let}\ f_T = \lambda\, s^{String}.\, \lambda(k,\ z)^{Xtf \times Xtf}.(\texttt{Xnode}\ s\ k, \\
&\qquad \textsf{if}\ \texttt{empty}\ k\ \textsf{then}\ \texttt{Xnode}\ s\ t_2\ \textsf{else}\ \texttt{Xnode}\ s\ z) \\
&\quad \textsf{in}\ f_T\ s\ (g\ ts)
\end{aligned}$$

which matches the form of Equation (4.9). Function g is defined as

$$\begin{aligned}
g &\overset{\text{def}}{=} \lambda\, t_1^{Xtf}.\ t_1\ [Xtf \times Xtf] \\
&\quad \textsf{let}\ f_T = \lambda\, t_2^{Xtf}.\, \lambda\, s^{String}.\ \lambda(k,\ z)^{Xtf \times Xtf}.\ (\texttt{Xnode}\ s\ k,\ \textsf{if}\ \texttt{empty}\ k \\
&\qquad\qquad \textsf{then}\ \texttt{Xnode}\ s\ t_2\ \textsf{else}\ \texttt{Xnode}\ s\ z) \\
&\qquad f_{XF} = \lambda\, t_2^{Xtf}.\ \lambda(k,\ z)^{Xtf \times Xtf}.(\texttt{Cons}\ t\ k,\ \texttt{Cons}\ (\texttt{Xtree}\ k\ t_2) \\
&\qquad\qquad (\texttt{Xtree}\ z\ t_2)) \\
&\quad \textsf{in}\ f_T\ (\lambda\, t_2^{Xtf}.(\texttt{nil},\ \texttt{nil}))\ f_{XF}
\end{aligned}$$

and operator `Xtree` is defined as follows:

$$\begin{aligned}
\texttt{Xtree} &\overset{\text{def}}{=} g.1 \\
&= (\lambda\, t_1^{Xtf}.\ t_1\ [Xtf \times Xtf] \\
&\quad \textsf{let}\ f_T = \lambda\, t_2^{Xtf}.\, \lambda\, s^{String}.\, \lambda(k,\ z)^{Xtf \times Xtf}.\ (\texttt{Xnode}\ s\ k, \\
&\qquad\qquad \textsf{if}\ \texttt{empty}\ k\ \textsf{then}\ \texttt{Xnode}\ s\ t_2\ \textsf{else}\ \texttt{Xnode}\ s\ z) \\
&\qquad f_{XF} = \lambda\, t_2^{Xtf}.\, \lambda(k,\ z)^{Xtf \times Xtf}.(\texttt{Cons}\ k\ z,\ \texttt{Cons}\ (\texttt{Xtree}\ k\ t_2) \\
&\qquad\qquad (\texttt{Xtree}\ z\ t_2)) \\
&\quad \textsf{in}\ f_T\ (\lambda\, t_2^{Xtf}.(\texttt{nil},\ \texttt{nil}))\ f_{XF}).1
\end{aligned}$$

```
1        CREATE VIEW Xtree($T_1$, $T_2$) AS
2          (SELECT $u.s$ AS $s$, $u.l * w_{T_2}$ AS $l$, $u.r * w_{T_2}$ AS $r$
3          FROM $T_1$ $u$)
4         UNION ALL
5          (SELECT $m.s$ AS $s$, $m.l + u.l * w_{T_2}$ AS $l$, $m.r + u.l * w_{T_2}$ AS $r$
6          FROM
7             (SELECT $u.s$, $u.l$, $u.r$
8             FROM $T_1$ $u$
9             WHERE NOT EXISTS
10               (SELECT $*$
11               FROM $T_1$ $v$
12               WHERE $u.l < v.l$ AND $u.r > v.r$),
13            (SELECT $m.s$, $m.l$, $m.r$
14            FROM $T_2$ $m$
15            WHERE EXISTS
16               (SELECT $*$
17               FROM $T_2$ $p$
18               WHERE $p.l < m.l$ AND $p.r > m.r$)
```

Figure 5.6: The relational template (View) for `Xtree`

5.2.2 SQL Template for `Xtree`

After the new operator `Xtree` has been successfully represented in *System F*, indicating that the extended XQ still exhibits the strong normalization property, this thesis attempts to show that a corresponding SQL template exists for `Xtree`, ensuring the polynomial time complexity of XQ query evaluation. The `Xtree` template is signified in Figure 5.6.

First, the interval values of each node of tree t_1, encoded in table T_1, are extended by the width of tree t_2 (Line 2, Line 3) to confirm that each node has enough interval space to contain tree t_2 as its children, and then each leaf nodes of tree t_1 has been picked up (Line 7 to Line 12) and attached to the children nodes of tree t_2 (Line 13 to Line 18) by a cross-product (Line 6 to Line 18), where the intervals of children nodes of t_2 have been adjust to fit the interval of each leaf node (Line 5). Finally, the tree nodes of t_1 with the extended interval and appended t_2 nodes, are returned as the result of view `Xtree`.

5.3 New Vertical Vfor Clause

XQ syntax defines FLWR-like clauses, where the for clause for $x \in e$ do e' has the following semantic meaning:

$$[\![\mathsf{for}\ x\ \in\ e\ \mathsf{do}\ e']\!]E \overset{\text{def}}{=} [\![e']\!](E[x := v_1])\ @ \ldots @\ [\![e']\!]E([x := v_k]) \\ \text{where } [v_1,\ \ldots,\ v_k] = [\![e]\!]E$$

The expression for $x\ \in e$ do e' first evaluates expression e in *Environment* E, and obtains a list of result $[v_1,\ \ldots,\ v_k]$, and then the assignment operation "$x \in$" translates *Environment* E to a list of new *Environments* $[E_1,\ \ldots,\ E_k]$. Finally, the expression e' is evaluated in the list of new environments, and the results are concatenated. The concatenating operation `@` has been defined as an *infix* operator in [6], and expressed as a *prefix* operator `concat` in this thesis by

$$\texttt{concat}\ ts_1\ ts_2 \equiv ts_1\ \texttt{@}\ ts_2$$

where ts_1 and ts_2 are XML forests.

With the concatenating operator `@`, the semantics of the for clause is easy to express. The idea of evaluating a for XQ expression with concatenating a sequence of individual result is used and extended to represent the semantics of vertical construction. A new vertical Vfor clause is proposed as

$$\mathsf{Vfor}\ x \in e\ \mathsf{do}\ e'$$

and its semantic equation is expressed, with the infix form $\wedge$ of `Xtree`, as

$$[\![\mathsf{Vfor}\ x\ \in\ e\ \mathsf{do}\ e']\!]E \overset{\text{def}}{=} [\![e']\!](E[x := v_1])\ \wedge \cdots \wedge\ [\![e']\!]E([x := v_k]) \\ \text{where } [v_1,\ \ldots,\ v_k] = [\![e]\!]E \qquad (5.5)$$

where the expression e' is first evaluated in *Environment* E, returning a list of result $[v_1,\ \ldots,\ v_k]$, and then the assignment operation "$x \in$" translates *Environment* E into a sequence of new *Environments* $[E_1,\ \ldots,\ E_k]$. Finally, the expression e' is evaluated in these new environments, and the sequence of results are combined by the vertical tree operation "$\wedge$", where

$$\texttt{Xtree}\ t_1\ t_2 \equiv t_1\ \wedge\ t_2$$

with ts_1 and ts_2 being XML trees.

5.3.1 Expressing the Semantics of Vfor in *System F*

With the extension of `Xtree` operator and Vfor clause, XQ has become more powerful to express the queries that are undefinable in the original XQ syntax. The focus is still on XQ's computational properties in the attempt to represent the vertical "for" clause Vfor in *System F*, which ensures that the extended XQ language still holds the strong normalization property.

The semantic equation (5.5) of Vfor is similar to that of For-Do. Also, it is necessary to use the general list *mapping* function `mapf`, defined in

Section 4.4.3, which applies the function f to each element in the list ts with type

$$\texttt{mapf} : (\theta \rightarrow \theta') \rightarrow \theta \ \mathsf{list} \ \rightarrow \theta' \ \mathsf{list}$$

First, the expression $\mathsf{Vfor}\ x\ \in e\ \mathsf{do}\ e'$ evaluates expression e in environment E and obtains a list of *Xtf* results, $[v_1,\ \ldots,\ v_k]$, defined as an *Xtf* list, *newVals*, such that

$$\mathit{newVals} = [\![e]\!]E \ : \mathit{Xtf}\ \mathsf{list}$$

Next, the assignment operation "$x \in$" transfers the given environment E into a list of new environments $[E_1,\ \ldots,\ E_k]$ by inserting each x value into a different environment. This transformation is implemented by a *mapping* function `mapPut`, where `mapPut` x *newVals* E creates a list of new environments, *newEs*. This is achieved by utilizing `put` $x\ e_j\ E$, defined in Section 4.4.3, to insert each value, $e_j,\ j = 1 \ldots k$, in the *newVals* list into environment E:

$$\begin{aligned}
\texttt{mapPut} \quad &= \quad \lambda\, x^{\mathit{Var}}.\ \texttt{mapf}\ (\texttt{put}\ x) \\
&= \quad \lambda\, x^{\mathit{Var}}.\ \lambda\, \mathit{newVals}^{\mathit{Xtf}\ \mathsf{list}}.\ \lambda\, E^{\mathit{Env}}.\ \mathit{newVals}\ [\mathit{Env}\ \mathsf{list}]\ \texttt{nil} \\
&\qquad\qquad (\lambda\, e^{\mathit{XQexp}}.\ \lambda\, z^{\mathit{Env}\ \mathsf{list}}.\ \texttt{Cons}\ (\texttt{put}\ x\ e\ E)\ z) \\
\texttt{mapPut} \quad &: \quad \mathit{Var} \rightarrow \mathit{Xtf}\ \mathsf{list} \rightarrow \mathit{Env} \rightarrow \mathit{Env}\ \mathsf{list}
\end{aligned}$$

After expression e' is evaluated in the list of environments, *newEs*, the results are combined by the vertical tree operator `Xtree`, and expressed as

$$[\![e']\!]E_1 \ \wedge \cdots \wedge \ [\![e']\!]E_k$$

The function, `semVfor`, expresses the evaluation of e' in the environment list, *newEs*:

$$\begin{aligned}
\texttt{semVfor}\ [\![e']\!]\ \texttt{nil} &= \texttt{nil} \\
\texttt{semVfor}\ [\![e']\!]\ (\texttt{Cons}\ E\ Es) &= \texttt{Xtree}\ ([\![e']\!]E)\ (\texttt{semVfor}\ [\![e']\!]\ Es)
\end{aligned}$$

and `semVfor` is defined as

$$\begin{aligned}
\texttt{semVfor} &= \lambda([\![e']\!])^{Env \to Xtf}.\, \lambda\, newEs^{Env\ \mathsf{list}}.\ newEs\ [Xtf] \\
&\qquad\qquad \texttt{nil}\ (\lambda\, E^{Env}.\ \lambda\, z^{Xtf}.\ \texttt{Xtree}\ ([\![e']\!]E)\ z) \\
\texttt{semVfor} &: (XQexp \to Env \to Xtf) \to Env\ \mathsf{list} \to Xtf
\end{aligned}$$

Finally, the semantic equation for `Vfor` can be defined as

$$\begin{aligned}
&[\![\mathsf{Vfor}\ x\ \in\ e\ \mathsf{do}\ e']\!]_{XQexp} E \overset{\text{def}}{=} \\
&\quad \mathsf{let}\ newVals = [\![e]\!]E \\
&\qquad \texttt{mapPut} = \lambda\, x^{Var}.\ \lambda\, newVals^{Xtf\ \mathsf{list}}.\ \lambda\, E^{Env}.\ newVals\ [Env\ \mathsf{list}] \\
&\qquad\quad \texttt{nil}\ (\lambda\, e^{XQexp}.\ \lambda\, z^{Env\ \mathsf{list}}.\ \texttt{Cons}\ (\texttt{put}\ x\ e\ E)\ z) \\
&\qquad \texttt{semVfor} = \lambda([\![e']\!])^{Env \to Xtf}.\, \lambda\, newEs^{Env\ \mathsf{list}}.\ newEs\ [Xtf] \\
&\qquad\quad \texttt{nil}\ (\lambda\, E^{Env}.\ \lambda\, z^{Xtf}.\ \texttt{Xtree}\ ([\![e']\!]E)\ z) \\
&\quad \mathsf{in}\ \texttt{semVfor}\ [\![e']\!]\ (\texttt{mapPut}\ x\ newVals)
\end{aligned}$$

5.3.2 Expressing the XQ Undefinable Query

With the syntax of the vertical Vfor clause and corresponding semantic meaning, the extended XQ can easily express query Q_2 in Figure 5.1, which transforms forest B into a *linked-list* deep tree that has no sibling relationships:

$$\mathsf{Vfor}\ \$1 = B\ \mathsf{do}\ \$1$$

where variable \$1 is bound to each tree element of B, which are trees a, b, c, and d (here the trees are simply expressed by their root labels), as seen in Figure 5.7(a); the Vfor clause utilizes operator `Xtree` to append all trees bound by 1 in a vertical direction to a deeply linked tree, shown in Figure 5.7(b).

The relational implementation of the Vfor $x \in e$ do e' is similar to the imple-

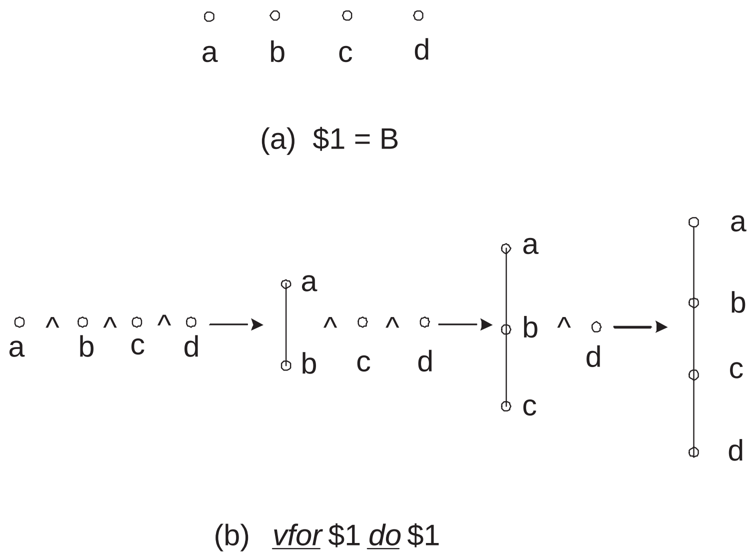

Figure 5.7: The illustration of query Q_2 with the Vfor clause

mentation of for $x \in e$ do e', except in the last step, the for-do horizontally concatenates, with operator `@`, of all the results evaluated in the sequence of environments created by $x \in e$; the Vfor applies the vertical tree operator, `Xtree`, to append the results evaluated in the sequence of environments vertically.

With the new tree operator `Xtree` and the new `Vfor` clause, XQ has been successfully extended to be more expressive. Also, the extended XQ has been shown to be *System F* definable and XQ-to-SQL translatable, which ensure this extension keeps the polynomial time complexity of query evaluation and *strong* termination property.

Chapter 6

Conclusion and Future Work

6.1 Conclusion

In this thesis *System F* is shown to be able to express queries in the core of XQ, an XQuery fragment proposed in [6]. As a second order typed lambda calculus, *System F* holds a proven expressive power [7] and a strong normalization property. Therefore, the XML data is represented in *System F* as inductive data types, XML tree and XML forest, with the corresponding iterators. Also, all the basic XML operators of XQ are encoded into *System F*. The semantics of XQ are expressed in *System F* by encoding the semantic environment using an *Environment* data with a set of operations. The successful encoding of XQ in *System F* guarantees the termination of XQ query evaluation and the decidability of query equivalence.

After XQ is represented in *System F*, an extension of XQ is investigated. Due to the limitations of XQ, a new tree operator, XTree, and a vertical Vfor clause are proposed to express some of the *undefinable* XQ queries. It is demonstrated that this extension still retains the XQ-to-SQL translation property to achieve polynomial evaluation time complexity by utilizing the relation query engine, and also XQ's *System F* encodable property to ensure the termination of XQ query evaluation.

6.2 Future Work

The suggestions for future work include the followings:

- to further extend XQ towards its completeness with keeping its XQ-to-SQL translatable and *System F* encodable properties, along with proof of the completeness of the extended XQ;
- an XQ-to-SF parser to translate XQ queries into *System F* syntax terms; also, an XQ *System F* processor should be implemented by a functional language to perform XQ query normalization and detect the query equivalence;
- to evaluate *System F* normalized XQ queries with a relational evaluation engine, an SF-to-SQL translator should be implemented to translate the *System F* syntax fragments to SQL statements.

Bibliography

[1] The XML Query Use Cases, Available from http://www.w3.org/TR/2005/WD-xquery-use-cases-20050404/.

[2] H.P. Barendregt. *The Lambda Calculus, Its Syntax and Semantics*, volume 103 of *Studies in Logics and the Foundations of Mathmatics*. North-Holland, 1981.

[3] H.P. Barendregt. *The Lambda Calculus with Types*. Handbook of Logic in Computer Science. Oxford University Press, 1990.

[4] M. Fernández B.Choi and J. Siméon. The XQuery Formal Semantics: A Foundation for Implementation and Optimization. Technical report, 2002.

[5] M. Fernández A. Malhotra K. Rose M. Rys D. Draper, P. Fankhauser and J. Siméon. XQuery 1.0 and XPath 2.0 Formal Semantics, Available from http://www.w3.org/TR/query-semantics/, 2002.

[6] D. DeHaan, D. Toman, M. Consens, and M. Ozsu. A Comprehensive XQuery to SQL Translation Using Dynamic Interval Coding, 2003.

[7] Y. Lafont J.-Y.Girard and P.Taylor. *Proof Theory and Types.* Cambridge Tracts in Theorectical Computer Science 7. Cambridge University Press, 1989.

[8] H. V. Jagadish, Laks V. S. Lakshmanan, Divesh Srivastava, and Keith Thompson. Tax: A tree algebra for xml. In *DBPL '01: Revised Papers from the 8th International Workshop on Database Programming Languages*, pages 149–164, London, UK, 2002. Springer-Verlag.

[9] Howard Katz. *XQuery from the Experts.* Addison-Wesley, 2004.

[10] J. C. Reynolds. Polymorphic Lambda-Calculus: Introduction. In *Logical Foundations of Functional Programming*, pages 77–86. Addison-Wesley, Reading, MA, 1990.

[11] J.C. Reynolds. *Theories of Programming Languages.* Cambridge University Press, 1998.