Integrating XPath with the Functional-Logic Language Toy

R. Caballero and Y. García-Ruiz and F. Sáenz-Pérez

TECHNICAL REPORT SIC 05/10

Dep. Lenguajes y Sistemas Informáticos y Computación
Univ. Complutense de Madrid

September 6, 2010

Contents

Abstract

The goal of this paper is to define a programming framework supporting XPath queries into the functional-logic language $\mathcal{TOY}$. This is done by exploiting the language characteristics, including non-determinism, logic variables, and higher-order functions and patterns. Our setting covers a wide range of standard XPath axes and tests. In particular reverse axes like *parent* or *ancestor* are implemented thanks to the double nature of XPath queries, which are both higher-order functions and data terms in our setting. Considering XPath queries as data terms, allows us to replace reverse axes by equivalent XPath expressions including only forward axes. The combination of these two different worlds, the functional-logic paradigm and the XML query language XPath, is very enriching for both of them. From the point of view of the functional-logic paradigm, the language is now able to deal with XML documents in a very simple way. From the point of view of XPath, our approach presents several nice properties that cannot be found in other related proposals. In particular, it allows the generation of XML test-cases for XPath queries, which can be very useful for finding bugs in erroneous queries.

Chapter 1

Introduction

In the last few years the Extensible Markup Language XML [10] has become the *de facto* standard for the exchange of different types of data. XQuery [12, 13] has been defined as a query language for finding and extracting information from XML documents. It extends XPath, a domain-specific language (DSL) that has become part of general-purpose languages. Although less expressive than XQuery, the simplicity of XPath makes it a perfect tool for many types of queries. In this paper, we address the task of incorporating XPath into the functional-logic system $\mathcal{TOY}$ [6].

The usual approach for integrating XPath in an existing programming language consists of:

1. Representing the XPath query by means of some suitable data type.
2. Employs some evaluator which takes the XPath query and the XML document as inputs, and produces the desired result as output.

However, in functional and functional-logic languages, a different approach is possible: XPath queries can be represented by higher-order functions connected by higher-order combinators. Using this approach, an XPath query becomes at the same time implementation (code) and representation (data term). In this paper we follow this idea, which has been used in the past, for instance for defining parsers in functional and functional-logic languages [2, 5].

The specific characteristics of functional-logic languages match perfectly the nature of XPath queries:

- *Non-deterministic functions* are used to nicely represent the evaluation of an XPath query, which consists of fragments of the input XML document.

- *Logic variables* are employed for instance when obtaining the contents XPath text nodes. Also, they will play an important role when defining XML test-cases for XPath queries, one of the most appealing features of our setting.
- By defining rules with *higher-order patterns* XPath queries become truly first-class citizens in our setting, which can be examined and transformed before being evaluated. This will allow us to define the transformation for introducing reverse axes like `parent` or checking that the query is constructed using XPath standard components.

The rest of the paper is organized as follows. Chapter 2 describes how to install the XPath library in $\mathcal{TOY}$ and how to use it. Chapter 3 introduces briefly the functional-language $\mathcal{TOY}$ and the XPath subset considered in this work. Chapter 4 defines the basic components of XPath queries in $\mathcal{TOY}$, as axes, tests and filters. Chapter 5 shows how XML test-cases for XPath queries can be readily generated, while Chapter 6 takes advantage of higher-order patterns for introducing several improvements in our framework. Finally, Chapter 7 presents some conclusions.

Chapter 2

Installation

2.1 Downloading, Installing and Executing ToyXPath

The latest version of the system $\mathcal{TOY}$ including the current prototype of the XPath library can be found at:

`http://gpd.sip.ucm.es/rafa/systems/xpath/toy.zip`

After downloading the files and unzipping them, the system is ready; no further installation steps are needed. $\mathcal{TOY}$ can be alternatively started as follows:

- From the binary distribution: Move to the folder toy and, depending on the OS:
  - Windows. Execute either toywin.exe (Windows-based application) or toy.exe (console- based application)
  - Linux. Execute ./toy (console-based application)
- From the source distribution (regardless of the OS):
  1. Move to the toy folder.
  2. Start SICStus Prolog version 3.11 or earlier.
  3. Type [toy] at the SICStus prompt.

After starting the system, the following prompt is shown:

```
Toy>
```

In order to load the XPath library, it is needed to run the file "toyXPath.toy". This process is performed by means of the $\mathcal{TOY}$ command:

```
Toy> /run(toyXPath)
```

Chapter 3

Preliminaries

Next we introduce briefly the functional-logic language $\mathcal{TOY}$ and the subset of XPath that we intend to integrate with $\mathcal{TOY}$.

3.1 The Functional-Logic Language $\mathcal{TOY}$

All the examples in this paper are written in the concrete syntax of the lazy functional-logic language $\mathcal{TOY}$ [6], but most of the code can be easily adapted to other similar languages like Curry [4]. We start explaining a possible representation of basic XML documents in $\mathcal{TOY}$.

3.1.1 XML Documents as $\mathcal{TOY}$ Data Types

A $\mathcal{TOY}$ program is composed of data type declarations, type alias, infix operators, function type declarations and defining rules for functions symbols. Data type declarations and type alias are useful for representing XML documents in $\mathcal{TOY}$, as illustrated next:

```
data xmlNode   = xmlText    string
               | xmlComment string
               | xmlTag     string [attribute] [xmlNode]
data xmlAttribute = xmlAtt     string string
type xml       = xmlNode
```

XML documents form a tree structure that starts at the root and branches to the leaves. The data type `xmlNode` represents nodes in a simple XML document. It

distinguishes three types of nodes: tags (element nodes), texts, and comments, each one represented by a suitable data constructor and with arguments representing the information about the node. For instance, constructor `xmlTag` includes the tag name (an argument of type `string`) followed by a list of attributes, and finally a list of child nodes. XML element nodes are represented by the constructor `xmlTag`, which includes a name (an argument of type `string`) followed by a list of attributes, and finally a list of child nodes. The data type `xmlAttribute` contains the name of the attribute and its value. The last type alias, `xml`, renames the data type `xmlNode`. Of course, this list is not exhaustive, since it misses several types of XML nodes, but it is enough for this presentation.

```
<?xml version='1.0'?>
<food>
 <item type="fruit">
   <name>watermelon</name>
   <price>32</price>
 </item>
 <item type="fruit">
   <name>oranges</name>
   <variety>navel</variety>
   <price>74</price>
 </item>
 <item type="vegetable">
  <name>onions</name>
  <price>55</price>
 </item>
 <item type="fruit">
  <name>strawberries</name>
  <variety>alpine</variety>
  <price>210</price>
 </item>
</food>
```

```
xmlTag "root" [xmlAtt  "version"  "1.0"] [
xmlTag "food"  [][
  xmlTag "item" [xmlAtt "type" "fruit"][
    xmlTag "name" [][xmlText "watermelon"],
    xmlTag "price" [][xmlText "32" ]
  ],
  xmlTag "item" [xmlAtt "type" "fruit"][
    xmlTag "name" [][xmlText "oranges" ],
    xmlTag "variety" [][xmlText "navel"],
    xmlTag "price" [][xmlText "74" ]
  ],
  xmlTag "item" [xmlAtt "type" "vegetable"][
    xmlTag "name" [][xmlText "onions" ],
    xmlTag "price" [][xmlText "55" ]
  ],
  xmlTag "item" [xmlAtt "type" "fruit"] [
    xmlTag "name" [][xmlText "strawberries"],
    xmlTag "variety" [][xmlText "alpine"],
    xmlTag "price" [][xmlText "210" ]
  ]
 ]]
```

Figure 3.1: XML example (left) and its representation in $\mathcal{TOY}$ (right)

The $\mathcal{TOY}$ primitive `load_xml_file` automatically loads an XML file returning its representation as a value of type `document`. Figure 3.1 shows an example of XML file and its representation in $\mathcal{TOY}$. All of the query examples in this paper use as input the XML document "food.xml". This document is a product catalog containing general information about products. It is quite simplistic, but it is useful for our purpose because it is easy to learn and remember while looking at query examples.

3.1.2 Functions in $\mathcal{TOY}$

Each rule for a function $f \in FS^n$ in $\mathcal{TOY}$ has a *left-hand side*, a right-hand side and an optional local definitions:

$$\underbrace{f\ t_1 \ldots t_n}_{\text{left-hand side}} \rightarrow \underbrace{r}_{\text{right-hand side}} \text{ where } \underbrace{s_1 = u_1, \ldots, s_m = u_m}_{\text{local definitions}}$$

where u_i and r are expressions (that can contain new extra variables) and t_i, s_i are patterns. The overall idea is that a function call $(f\ e_1 \ldots e_n)$ will return an instance $r\theta$ of r, if:

- Each e_i can be reduced to some pattern a_i, $i = 1 \ldots n$, such that $(f\ t_1 \ldots t_n)$ and $(f\ a_1 \ldots a_n)$ are unifiable with most general unifier θ, and
- $u_i\theta$ can be reduced to pattern $s_i\theta$ for each $i = 1 \ldots m$.

In $\mathcal{TOY}$, variable names must start by either an uppercase letter or an underscore, whereas other identifiers (functions, constants, ...) start with lowercase. A more interesting example is the following choice operator `?`:

```
infixr 35 ?
X ? _Y = X
_X ? Y = Y
```

The infix declaration `infixr 35 ?` indicates that `?` is an infix operator that associates to the right (the r in `infixr`) and that its priority is 35. The priority will be used to assume precedences in the case of expressions involving different operators. The `?` operator represents the non-deterministic choice. Computations in $\mathcal{TOY}$ start when the user inputs some goal as:

```
Toy> 1 ? 2 ? 3 ? 4 == R
    { R -> 1 }
    Elapsed time: 0 ms.
  sol.1, more solutions (y/n/d/a) [y]?
       { R -> 2 }
       Elapsed time: 0 ms.
  sol.2, more solutions (y/n/d/a) [y]?
```

```
        { R -> 3 }
        Elapsed time: 0 ms.
sol.3, more solutions (y/n/d/a) [y]?
        { R -> 4 }
        Elapsed time: 0 ms.
sol.4, more solutions (y/n/d/a) [y]?
        no
        Elapsed time: 0 ms.
```

This goal asks $\mathcal{TOY}$ for values of the logical variable R that make true the (strict) equality `1 ? 2 ? 3 ? 4 == R`. This goal yields four different answers {`R` $\mapsto$ `1` }, {`R` $\mapsto$ `2` }, {`R` $\mapsto$ `3` }, and {`R` $\mapsto$ `4` }.

The next function extends the choice operator to lists:

```
member [X|Xs] = X ? member Xs
```

For instance, the goal `member [1,2,3,4] == R` has the same four answers that were obtained by trying `1 ? 2 ? 3 ? 4 == R`.

$\mathcal{TOY}$ is a *typed* language. Types do not need to be annotated explicitly by the user, they are inferred by the system, which rejects ill-typed expressions. However, function type declarations can also be made explicit by the user, which improves the clarity of the program and helps to detect some bugs at compile time. For instance, a function type declaration for `member` is:

```
member :: [A] -> A
```

This indicates that `member` takes a list of elements of type A, and returns a value which must be also of type A. As usual in functional programming languages, $\mathcal{TOY}$ allows partial applications in expressions. Consider for instance the function that returns the n-th value in a list:

```
nth :: int -> [A] -> A
nth N [X|Xs]  = if N==1 then X else nth (N-1) Xs
```

This function has program arity 2, which means that the program rule is applied when it receives two arguments (an integer and a list). Then, the following goal is valid:

```
Toy> nth 1 == R1, R1 ["hello","friends"] == R2
    { R1 -> (nth 1),
```

```
          R2 ->  "hello" }
        Elapsed time: 0 ms.
sol.1, more solutions (y/n/d/a) [y]?
        no
        Elapsed time: 0 ms.
```

This goal produces the answer { `R1` $\mapsto$ `(nth 1), R2` $\mapsto$ `"hello"` }. In this solution, `R1` is bound to the partial application `nth 1`. Observe that `R1` has type `([A] -> A)`, and thus it is a *higher-order* variable. Applying `R1` to a list of strings like in the second part of the goal `R1 ["hello","friends"] == R2` 'triggers' the use of the program rule for `nth`. A particularity of $\mathcal{TOY}$ is that partial applications with pattern parameters are also valid patterns. They are called *higher-order patterns*. For instance, one could define a function like:

```
first (nth N)  = N==1
```

because `nth N` is a higher-order pattern. However, a program rule like: `foo (nth 1 [2]) = true` is not valid, because `(nth 1 [2])` is reducible and thus it is not a valid pattern. Higher-order variables and patterns play an important role in our setting.

Functional-logic programming functions share with logic programming clauses the possibility of using logic variables as parameters. For instance, consider the `length` function defined as usually in Functional Programming:

```
length :: [A] -> int
length [] = 0
length [X|Xs] = 1 + length Xs
```

With this function we can try the goal `length [1,2,3] == R`, which returns the expected answer `R` $\rightarrow$ `3`.

```
Toy> length [1,2,3] == R
      { R -> 3 }
      Elapsed time: 0 ms.
sol.1, more solutions (y/n/d/a) [y]?
      no
      Elapsed time: 0 ms.
```

However, the function can be used as well for generating lists of a given length, like in `length L == 3`.

```
Toy> length L == 3
    { L -> [ _A, _B, _C ] }
    Elapsed time: 0 ms.
sol.1, more solutions (y/n/d/a) [y]? n
```

This goal produces the answer `R` → `[ _A, _B, _C ]`. The possibility of generating values for the parameters will be employed for generating test cases in Chapter 5. It is worth observing that the previous goal will loop if we ask for a further solution, because `length` will continue producing lists of length 4,5,... which will not satisfy the goal. A complete definition of the language including examples can be found in the `doc` folder of the $\mathcal{TOY}$ system.

3.2 The XML query language XPath

XPath is a functional typed language. We consider XPath queries represented by *location paths* of the form:

XPath	= `doc`(fileName) `/` *Relative*
Relative	= *Step*$_1$ `/` ... `/` *Step*$_n$ | *Relative* | *Relative*
Step	= *Axis* :: *Test* | *Axis* :: *Test*[Filter]
Axis	= `self` | *ForwardAxis* | *ReverseAxis*
ForwardAxis	= `child` | `descendant` | `descendant-or-self` | ...
ReverseAxis	= `parent` | `ancestor` | `ancestor-or-self` | ...
Test	= `node()` | *name* | `text()` | `comment()` | *

The grammar above specifies a subset of the XPath language, enough for representing easily most XPath queries. A complete description of XPath 2.0 can be found at [11]. In particular there are other axes that can be used in XPath, like `following-sibling`, but according to [13], implementations are not required to support them.

Absolute XPath location paths start with `doc`(fileName), which loads the XML *fileName*, and sets the context node to the root. Then a sequence of relative steps are used for navigating the document. Each step takes as starting node the context node, and it is composed by an *axis* that changes the context node, and by a test that returns only those nodes verifying the test. Tests can be *kind tests* like `comment()` which holds for every comment node, or *name tests* which check the name of the node. A special kind test is `*` which holds for *element* nodes.

Example 3.2.1 *For instance the XPath query:*

```
doc("food.xml")/child::food/
                child::item[child::name/child::text()="onions"]/
                child::price/child::text()
```

returns the price of onions in file "food.xml".

Assuming the XML document of Figure 3.1, this query returns in a XQuery/XPath system the value "55".
Observe the presence of the filter `[child::name/child::text()="watermelon"]`. Filters select some context nodes that verify certain conditions. In this case it means that we will select all the nodes N descendant of the root such that they have a children element with tag "name", which contains a text "watermelon". However the filter do not change the context node, that is, the node N is kept as context after the step. The rest of the location path navigates to the children of N with tag "price", returning its text value. XPath allows also abbreviated forms. For instance the XPath query of Example 3.2.1 can be written as `doc("food.xml")/food/item[name="onions"]/price/text()`.

Chapter 4

XPath Queries in $\mathcal{TOY}$

In this section we present the basis of our setting, including the type for XPath queries, the step combinators, tests and forward axes. The reverse axes will be defined in Chapter 6.

4.1 The Type xPath

Typically, XPath expressions return several fragments of the XML document. Thus, the expected type for `XPath` could be

```
type xPath = xml -> [xml]
```

meaning that a list or sequence of results is obtained. This is also the approach considered in [1] and also the usual in functional programming [3]. However, in our case we will take advantage of the non-deterministic features of our language, returning each result individually and avoiding the introduction of lists. Thus, we define an XPath expression as a function taking a (fragment of) XML as input and returning a (fragment of) XML as its result:

```
type xPath = xml -> xml
```

4.2 Loading XML Documents

In order to apply an XPath expression to a particular document, we will use the following infix operator definition:

```
(<--)::string  -> xPath -> xml
S <-- Q =  Q  (load_xml_file S)
```

The input arguments of this operator are the string `S` representing the file name and an XPath query `Q`. The function applies `Q` to the XML document contained in file `S`. This operator will play in $\mathcal{TOY}$ the role of `doc` in XPath.

4.3 XPath Combinators

Next, we define the XPath combinators `/` and `::` which correspond to the connection between steps and between axis and tests, respectively. In $\mathcal{TOY}$, these symbols are defined simply as function composition:

```
infixr  55 .::.
(.::.) :: xPath -> xPath -> xPath
(F .::. G) X  = G (F X)

infixr 40 ./.
(./.) :: xPath -> xPath -> xPath
(F ./. G) X = G (F X)
```

We use the function operator names `.::.` and `./.` instead of `::` and `/` respectively, because these are already defined in the host language. The variable `X` represents the input XML fragment (in fact it is the context node), and the rules specify how the output XPath expression will first apply the first XPath expression (`F`) and then the second one (`G`). Observe that the precedences and associativity indicate that an expression like:

```
A.::.B ./. C.::.D ./. E.::.F
```

is understood by $\mathcal{TOY}$ as:

```
(A.::.B) ./. ((C.::.D) ./. (E.::.F))
```

The disjunction operator `|` of XPath is is represented in $\mathcal{TOY}$ simply by the choice operator `?` defined in Seption 3.1.

4.4 Basic Axes

We are ready to define the basic axes. The first one is `self`, which returns the context node itself. In our setting, this corresponds simply to the identity function:

```
self :: xPath
self X = X
```

A more interesting axis is `child` which returns using the non-deterministic function `member` all the children of the context node. In XML only *element nodes* have children. These nodes correspond in $\mathcal{TOY}$ representation to terms rooted by constructor `tag`. Therefore we can define:

```
child :: xPath
child (tag _Name _Attr L) = member L
```

Once `child` has been defined, `descendant` is just a generalization:

```
descendant :: xPath
descendant X = child X
descendant X = if (child X == Y) then descendant Y
```

The first rule for this function specifies that `child` must be used once, while the second corresponds to two or more applications of `child`. Finally, the axis `descendant-or-self` is straightforward:

```
descendant_or_self :: xPath
descendant_or_self = self
descendant_or_self = descendant
```

Observe that the XML input argument is not necessary in this natural definition. Chapter 6 will introduce the remaining axes (`parent` and `ancestor`).

4.5 Tests

The first test we define is `nodeT`, which corresponds to *node()* in the usual XPath syntax. This test is simply the identity:

```
nodeT :: xPath
nodeT X = X
```

For instance, here is the XPath expression that returns all the nodes in an XML document, together with its $\mathcal{TOY}$ equivalent:

XPath → `doc("food.xml")/descendant-or-self::node()`
$\mathcal{TOY}$ → `("food.xml" <-- descendant_or_self.::.nodeT)==R`

The only difference is that the $\mathcal{TOY}$ expression returns one result at a time in the variable `R`, asking the user if more results are needed. If the user wishes to obtain all the solutions at a time, as usual in XPath evaluators, then it is enough to use the primitive `collect`.

Example 4.5.1 *The following example selects all the nodes in the XML document "food.xml".*

```
Toy> collect ("food.xml" <-- descendant_or_self.::.nodeT) == R
       { R -> [ (xmlTag  "root" [ (xmlAtt  "version"  "1.0") ]
                    [ (xmlTag  "food" []
                     [ (xmlTag  "item" [ (xmlAtt  "type"  "fruit") ]
                ...
                (xmlText  "strawberries"),
                (xmlText  "alpine"),
                (xmlText  "210") ] }
       Elapsed time: 78 ms.
 sol.1, more solutions (y/n/d/a) [y]?
       no
       Elapsed time: 0 ms.
```

Notice the answer to the $\mathcal{TOY}$ goal will produce a single answer, with `R` *instantiated to a list whose elements are the nodes in "food.xml".*

The *name test* checks if the context node is an element with a certain name `S`:

```
nameT :: name ->  xPath
nameT S (tag S Attr L ) = tag S Attr L
```

The test either returns as output the same XML fragment received as input, or fails. Here is an example of a relative location path using this test:

XPath → `child::food / child::item`
$\mathcal{TOY}$ → `child.::.nameT "food" ./.  child.::.nameT "item"`

Observe that the expression in $\mathcal{TOY}$ is longer in length due to the presence of the test `nameT`, which does not required in XPath. In the next section we will see how this situation improves when introducing the abbreviated forms.

Example 4.5.2 *The following example selects all the price nodes:*

```
Toy> ("food.xml" <--child.::.nameT "food" ./. child.::.nameT "item"./.
                    child.::.nameT "price" ) == R

      { R -> (xmlTag  "price" [] [ (xmlText  "32") ]) }
      Elapsed time: 16 ms.
sol.1, more solutions (y/n/d/a) [y]?
      { R -> (xmlTag  "price" [] [ (xmlText  "74") ]) }
      Elapsed time: 0 ms.
sol.2, more solutions (y/n/d/a) [y]?
      { R -> (xmlTag  "price" [] [ (xmlText  "55") ]) }
      Elapsed time: 0 ms.
sol.3, more solutions (y/n/d/a) [y]?
      { R -> (xmlTag  "price" [] [ (xmlText  "210") ]) }
      Elapsed time: 0 ms.
sol.4, more solutions (y/n/d/a) [y]?
      no
      Elapsed time: 0 ms.
```

Other useful tests are `textT` and `commentT`, which correspond to *text()* and *comment()*, respectively, in XPath:

```
textT :: string -> xPath
textT S (txt S) = txt S

commentT :: string -> xPath
commentT S (comment S) = comment S
```

In the case of $\mathcal{TOY}$, the text (respectively comment) string is obtained by means of a logic variable as, for instance, in:

XPath → `child::food/child::item/child::price/child::text()`
$\mathcal{TOY}$ → `child.::.nameT "food" ./.  child.::.nameT "item" ./.`
`child.::.nameT "price" ./.  child.::.textT P`

The logic variable `P` will obtain the prices contained in the example document. Finally, the text `elem` represents in $\mathcal{TOY}$ the XPath test `*` which is satisfied only for *element* nodes.

```
elem:: xPath
elem  = nameT _
```

Notice the use of the anonymous variable _ in `nameT _`. It indicates that any tag name is accepted.

Example 4.5.3 *The following example selects all the element nodes under the "item" element:*

```
Toy> ("food.xml" <--child.::.nameT "food" ./. child.::.nameT "item" ./.
                    child.::.elem ) == R

      { R -> (xmlTag  "name" [] [ (xmlText  "watermelon") ]) }
      Elapsed time: 15 ms.
sol.1, more solutions (y/n/d/a) [y]?
      { R -> (xmlTag  "price" [] [ (xmlText  "32") ]) }
      Elapsed time: 0 ms.
sol.2, more solutions (y/n/d/a) [y]?
      { R -> (xmlTag  "name" [] [ (xmlText  "oranges") ]) }
      Elapsed time: 0 ms.
sol.3, more solutions (y/n/d/a) [y]?
      { R -> (xmlTag  "variety" [] [ (xmlText  "navel") ]) }
      Elapsed time: 0 ms.
sol.4, more solutions (y/n/d/a) [y]?
      { R -> (xmlTag  "price" [] [ (xmlText  "74") ]) }
      Elapsed time: 16 ms.
sol.5, more solutions (y/n/d/a) [y]?
      { R -> (xmlTag  "name" [] [ (xmlText  "onions") ]) }
      Elapsed time: 0 ms.
sol.6, more solutions (y/n/d/a) [y]?
      { R -> (xmlTag  "price" [] [ (xmlText  "55") ]) }
      Elapsed time: 0 ms.
sol.7, more solutions (y/n/d/a) [y]?
      { R -> (xmlTag  "name" [] [ (xmlText  "strawberries") ]) }
      Elapsed time: 0 ms.
sol.8, more solutions (y/n/d/a) [y]?
      { R -> (xmlTag  "variety" [] [ (xmlText  "alpine") ]) }
```

```
      Elapsed time: 0 ms.
sol.9, more solutions (y/n/d/a) [y]?
      { R -> (xmlTag  "price" [] [ (xmlText  "210") ]) }
      Elapsed time: 0 ms.
sol.10, more solutions (y/n/d/a) [y]?
      no
      Elapsed time: 0 ms.
```

Example 4.5.4 *The following example selects all the element nodes under the "name" element:*

```
Toy> ("food.xml" <--child.::.nameT "food" ./. child.::.nameT "item" ./.
                    child.::.nameT "name" ./. child.::.elem ) == R
  no
  Elapsed time: 0 ms.
```

In this case no answer is obtained because there is not element nodes under the "name" element nodes.

4.6 Abbreviations

A number of abbreviations are used frequently in XPath expressions. The most important abbreviation is that `child::` can be omitted from a location step. Thus,

```
child::food/child::price/child::item
```

becomes simply

```
food/price/item
```

In $\mathcal{TOY}$ we cannot do that directly because we are in a typed language and the combinator `./.` expects xPath expressions and not strings. However, we can introduce a similar abbreviation by defining a new unitary operator `name` and `text`, which transform strings in XPath expressions:

```
name :: string -> xPath
name S  =  child .::. (nameT S)

text :: string -> xPath
text S  =  child .::. (textT S)
```

An example:

XPath $\rightarrow$ `food/item/variety`
$\mathcal{TOY}$ $\rightarrow$ `name "food" ./. name "item" ./. name "variety"`

The XPath queries in Examples 4.5.2 and 4.5.3 can be write in $\mathcal{TOY}$ as

```
Toy> ("food.xml" <-- name "food" ./. name "item" ./. name "price" ) == R

Toy> ("food.xml" <-- name "food" ./. name "item" ./. child.::.elem ) == R
```

respectively. Notice that the last step in the second XPath query can not be abbreviated.

The same idea can be applied to other tests:

```
text :: string -> xPath
text A  =  child .::. textT A
```

Another useful XPath abbreviation is `//` which stands for

`/descendant-or-self::node()/.`

Analogously, in $\mathcal{TOY}$ we can define:

```
infixr  30 .//.
(.//.)::  xPath -> xPath -> xPath
A .//. B = append A (descendant_or_self .::. nodeT ./. B)

append::xPath -> xPath -> xPath
append (A.::.B) C = (A.::.B) ./. C
append (X ./.Y) C = X ./. (append Y  C)
```

Notice that a new function `append` is used for concatenating the XPath expressions. This function is analogous to the well-known `append` for lists, but defined over `xPath` terms. This is our first example of the utility of higher-order patterns. For instance pattern `(A.::.B)` has type `xPath`, i.e. `xml -> xml`.

The next example uses both `name`, `.//.` and the disjunction operator, asking for all the elements with name either "price" or "variety".

Example 4.6.1 *The The following example selects all the variety nodes in the XML document "food.xml".*

```
Toy> ("food.xml" <-- name "food" .//. name "variety") == R

       { R -> (xmlTag  "variety" [] [ (xmlText  "navel") ]) }
       Elapsed time: 16 ms.
sol.1, more solutions (y/n/d/a) [y]?
       { R -> (xmlTag  "variety" [] [ (xmlText  "alpine") ]) }
       Elapsed time: 16 ms.
sol.2, more solutions (y/n/d/a) [y]?
       no
       Elapsed time: 0
```

Another possible improvement is to define a new version of `.::.` whose right-hand side is an XML name (a string):

```
infixr  55 .::
(.::) :: xPath -> string -> xPath
F .:: G   = (F .::. (nameT G))
```

The next example uses both `name` and `.//.`:

XPath → `child::food // child::price`
$\mathcal{TOY}$ → `child.::"food" .//.  child.::"price"`

Another possible improvement is to define two versions of `./.`:

- One whose left-hand side is an XML name (a string):

```
infixr 35 /.
(/.) :: string -> xPath -> xPath
S /. X = name S ./. X
```

- One whose right-hand side is an XML name (a string):

```
infixr 35 ./
(./) :: xPath -> string -> xPath
X ./ S =  append X (child.::S)
```

Therefore, a dot (`.`) at any side of a combinator denotes that an XPath expression is expected, while the absence of such a dot indicates that a string at that position is expected. For instance:

XPath → `food/item/price/text()`
$\mathcal{TOY}$ → `"food"/."item"/."price"/.text P`

In XPath we obtain the output: `32 74 55 210`, while in $\mathcal{TOY}$ we get the associated four solutions: `P` ↦ `32`, `P` ↦ `74`, `P` ↦ `55`, and `P` ↦ `210`.

It is worth noticing that the wildcard `*` of XPath does not need to be defined in our setting, since it corresponds simply to the use of an anonymous variable. For instance, here we show two equivalent expressions in XPath and $\mathcal{TOY}$:

XPath → `food/*/price/text()`
$\mathcal{TOY}$ → `"food" /. _ /. "price" /. text P`

In XPath we obtain the output: `32 74 55 210`, while in $\mathcal{TOY}$ we get the associated four solutions: `P` ↦ `32`, `P` ↦ `74`, `P` ↦ `55`, and `P` ↦ `210`.

4.6.1 Filters

Optionally, XPath tests can include a predicate or filter. Filters in XPath are enclosed between square brackets. In $\mathcal{TOY}$, they will be enclosed between round brackets and connected to its associated XPath expression by the operator `.#`:

```
infixr  60 .#
(.#)::xPath -> xPath -> xPath
(Q .# F)  X =  if F Y == _ then Y  where Y = Q X
```

This definition can be understood as follows: first the query `Q` is applied to the context node `X`, returning a new context node `Y`. Then the `if` condition checks whether `Y` satisfies the filter `F`. This is done simply by checking that `F Y` does not fail, which means that it returns some value represented by the anonymous variable in `F Y == _`. Although XPath filter predicates allow several possibilities, in this presentation we restrict to XPath expressions and position predicates. As in the previous section, it is convenient to define a version of `.#` accepting a string instead of an XPath query:

```
infixr  60 #
```

```
(#)::string -> xPath -> xPath
S # F = child.::.(nameT S).# F
```

Example 4.6.2 *The following example selects all the "item" element nodes that have a "variety" child.*

```
Toy> ("food.xml" <-- "food" /. "item"#(child.::.nameT "variety")) == R
      { R -> (xmlTag  "item" [ (xmlAtt  "type"  "fruit") ]
                [ (xmlTag  "name" [] [ (xmlText  "oranges") ]),
                  (xmlTag  "variety" [] [ (xmlText  "navel") ]),
                  (xmlTag  "price" [] [ (xmlText  "74") ]) ]) }
      Elapsed time: 0 ms.
sol.1, more solutions (y/n/d/a) [y]?
      { R -> (xmlTag  "item" [ (xmlAtt  "type"  "fruit") ]
                [ (xmlTag  "name" [] [ (xmlText  "strawberries") ]),
                  (xmlTag  "variety" [] [ (xmlText  "alpine") ]),
                  (xmlTag  "price" [] [ (xmlText  "210") ]) ]) }
      Elapsed time: 0 ms.
sol.2, more solutions (y/n/d/a) [y]?
      no
      Elapsed time: 0 ms.
```

Filters in XPath are defined usually by means of comparison operators like `=` or $>$. For instance, the following XPath query asks for the price of watermelons:

```
food/item[name="watermelon"]/price
```

Observe that `name="watermelon"` means in fact: check whether the context node has a children `name`, which has a children text `watermelon`. In $\mathcal{TOY}$ we can mimic this behavior by defining:

```
(.=) :: string -> string -> xPath -> bool
(.=) A B X = (A  /. text B) X == _
```

This operator takes as input parameters both sides of the equality, represented by the strings `A` and `B`, and the input XML context `X`. The strict equality with anonymous variable at the right-hand side is used to check whether `A` has a text child `B` in the XPath context `X`. An example of application of this operator:

XPath $\rightarrow$ `food/item[name="onions"]`
$\mathcal{TOY}$ $\rightarrow$ `"food"/."item"#("name".="onions")`

The same approach can be used for other operators, as >:

```
(.>):: string -> int ->xPath
(.>) A B X =  if (((name A ./. text C) X == Y) /\ (val C > B)) then Y
```

In this case the text node is converted into an integer using the simple function `val`:

```
val :: [char] -> int
val [X] = ord(X) - ord('0')
val [X|Xs] = val(Xs) + val([X])*10^(length Xs)
```

The conjunction operator /\ is defined as usual in functional programming:

```
infixr 30 /\
false /\ X = false
true /\ X = X
```

The next example employs the operator `.>` for obtaining the items with price greater than 100:

XPath → `food/item[price > 100]`
$\mathcal{TOY}$ → `"food"/."item"#("price" .> 100)`

Filters selecting attributes with certain values are of particular interest, and are represented in XPath by symbol `@`. In $\mathcal{TOY}$ they will be represented by the operator `@=`:

```
(@=) :: xmlName -> xmlName -> xPath
(@=) S V X = if (xmlAtt S V == member Attr) then X
             where (xmlTag _Name Attr _L) = X
```

This filter checks if the attribute `S` of the context element takes the value `V`. The next example shows the prices of the elements of type `fruit`:

XPath → `food/item[@type="fruit"]/price`
$\mathcal{TOY}$ → `"food"/."item"#("type"@="fruit")./"price"`

Example 4.6.3 *The following example selects the name of all fruits in the document.*

```
Toy> ("food.xml" <-- "food"/."item"#("type"@="fruit")./.
              name "name" ./. text T) == R

   { T ->  "watermelon",
     R -> (xmlText  "watermelon") }
   Elapsed time: 15 ms.
sol.1, more solutions (y/n/d/a) [y]?
   { T ->  "oranges",
     R -> (xmlText  "oranges") }
   Elapsed time: 0 ms.
sol.2, more solutions (y/n/d/a) [y]?
   { T ->  "strawberries",
     R -> (xmlText  "strawberries") }
   Elapsed time: 0 ms.
sol.3, more solutions (y/n/d/a) [y]?
   no
   Elapsed time: 0 ms.
```

Or course, other comparison operators as @> can be defined analogously.

4.7 Position Filters

A particular type of predicate filters are those that select an element in the sequence of results by its position. Position filters can use `last` for accessing to the last element of the sequence. Although positions in XPath are written with the same syntax as other predicates, in $\mathcal{TOY}$ we use a new operator to distinguish their particularities:

```
infixr  60 .!
(.!) :: xPath -> (int -> int) -> xPath
(P .!  F)  X = nth (F N) L
               where L = collect (P X)
                     N = length L
```

The definition of the functions `length` and `nth` can be found in Section 3.1. The operator first collects in the new variable `L` the results obtained when applying the query `P` to the input context `X`. Then, the function `F` provided by the user is employed for returning the position that will be used for selecting the n-th element of `L`. This function receives the length of the list as input parameter.

Example 4.7.1 *The following example selects the first "item" element in the current context node.*

```
Toy> ("food.xml" <-- name "food"./.child.::.nameT "item".!(pos 1)) == R

   { R -> (xmlTag  "item" [ (xmlAtt  "type"  "fruit") ]
            [ (xmlTag  "name" [] [ (xmlText  "watermelon") ]),
              (xmlTag  "price" [] [ (xmlText  "32") ]) ]) }
   Elapsed time: 16 ms.
sol.1, more solutions (y/n/d/a) [y]?
   no
   Elapsed time: 0 ms.
```

As usual, it will be convenient to define a version of the operator that accepts a string in the left-hand side:

```
infixr  50 !
(!) :: string -> (int -> int) -> xPath
S ! F = (child.::S).! F
```

Now, we can define different functions for accessing the elements:

```
pos :: int -> int -> int
pos N _ = N

lastMinus :: int -> int -> int
lastMinus M N = N-M

last :: int -> int
last = lastMinus 0
```

Function `pos` will be used for accessing elements starting from the first element, while `lastMinus` takes as starting point the end of the list. Finally `last` indicates that last element is selected. Next, we show a few examples comparing XPath queries and their equivalences in $\mathcal{TOY}$:

Example 4.7.2 *The following example selects the "item" element in second from last place in the document.*

```
Toy> ("food.xml" <-- "food" /. child.::.nameT "item".!(lastMinus 1))== R

       { R -> (xmlTag  "item" [ (xmlAtt  "type"  "vegetable") ]
                  [ (xmlTag  "name" [] [ (xmlText  "onions") ]),
                    (xmlTag  "price" [] [ (xmlText  "55") ]) ]) }
       Elapsed time: 15 ms.
sol.1, more solutions (y/n/d/a) [y]?
       no
       Elapsed time: 0 ms.
```

This goal in $\mathcal{TOY}$ is equivalent to `food/item[last-1]` *in XPath.*

Example 4.7.3 *The following example selects the "item" element in first place in the document.*

```
Toy>  ("food.xml" <-- "food"/."item"!(pos 1))== R
    { R -> (xmlTag  "item" [ (xmlAtt  "type"  "fruit") ]
       [ (xmlTag  "name" [] [ (xmlText  "watermelon") ]),
         (xmlTag  "price" [] [ (xmlText  "32") ]) ]) }
    Elapsed time: 16 ms.
sol.1, more solutions (y/n/d/a) [y]?
    no
    Elapsed time: 0 ms.
```

Example 4.7.4 *The "item" element in last place in the document can be selects by the next query:*

```
Toy> ("food.xml" <-- "food"/."item"!lastPos)== R
    { R -> (xmlTag  "item" [ (xmlAtt  "type"  "fruit") ]
             [ (xmlTag  "name" [] [ (xmlText  "strawberries") ]),
               (xmlTag  "variety" [] [ (xmlText  "alpine") ]),
               (xmlTag  "price" [] [ (xmlText  "210") ]) ]) }
    Elapsed time: 16 ms.
sol.1, more solutions (y/n/d/a) [y]?
    no
    Elapsed time: 0 ms.
```

Chapter 5

Generating Test-Cases for XPath Expressions

Suppose that we wish to know the price of onions as stored in our XML document. According to the previous section, we can write in $\mathcal{TOY}$:

```
Toy> ("food.xml"<--"food"/."item"#("type"@="onions")./"price" )==R
```

This goal returns no answer. However, checking "food.xml" we find that it contains at least one answer for our goal. Where is the error? Sometimes it is useful to have a test case, i.e., an XML file which contains some answer for the query. Comparing the test case and the original XML document can help to find the error.

In our setting, such test cases are obtained for free. For instance, in the previous query we can try the goal:

```
Toy> ("food.xm" <-- "food"/."item"#("type"@="onions")./"price") X == _
```

which asks for an XML document `X` such that the query succeeds. The anonymous variable at the right-hand side of the strict equality indicates that we are not interested in the output. However, the answer is difficult to read and understand:

```
X -> tag _A _B
    [
      tag  "food" _C
        [
         tag  "item"
          [ (att  "type"  "onions") | _D ]
```

```
          [ (tag  "variety" _E _F) | _G ]
        | _H
       ]
     | _I
   ]
```

The logic variables indicate that replacing them by any valid XML fragment will produce a valid XML test case for the query. In particular, in the case of lists they indicate that other elements can be added to the list. The smaller test case corresponds to substituting these variables by the empty list.

In order to enhance the readability of the result we define a function that uses the primitive `write_xml_file` to write the XML document in an output file

```
generateTC:: xPath -> string -> bool
generateTC F S = if (F X == _) then  write_xml_file X S
```

Function `generateTC` receives the XPath expression `F` and the file name `S` as input parameters. It looks for an XML test case `X`, and writes it to the file using the primitive `write_xml_file`.

The goal:

```
Toy> generateTC ("food"/."item"#("type"@="onions")./"price") "tc.xml" == R
```

produces the following XML file "tc.xml":

```
<food>
 <item type="oranges">
  <variety />
 </item>
</food>
```

It can be observed that the primitive has replaced the logic variables by empty elements as expected. Comparing this file and our example "food.xml", we see that "oranges" is not an attribute, but a child node. The correct query should be:

```
("food.xml"  <--  "food" /."item"#("name".="onions")./ "price" ) == R
```

which returns the answer:

```
R -> tag  "price" [ ] [text "55"]}.
```

Chapter 6

Higher Order Patterns

The possibility of employing higher order patterns in $\mathcal{TOY}$ allows the user considering XPath queries as truly data terms. Queries can be examined and modified before and during its evaluation, as any constructed term. In this chapter, we take advantage of this feature in two ways. First, we define a function that checks if an XPath query follows the XPath standard. Then, we apply a transformation similar to those described in [9] for introducing the reverse axes `parent` and `ancestor`.

6.1 Validating XPath Queries

So far we have described several different tests and axes that can be combined for defining XPath queries. Moreover, our setting allows the user defining their own combinators, axes and tests, or combining the existing ones in a non-standard way. For instance, the query `nodeT.::.child` is allowed by our setting, although it does not follows the XPath grammar (it should be `child.::.nodeT`, first the axis and then the test). The reason is that the expression is well-typed from the point of view of a $\mathcal{TOY}$ expression. Although in principle such unusual queries can work and even be useful in some cases, it could be interesting to define a function that indicates whether a query conforms to the XPath standard or not. In order to define such a function, we first consider the basic pieces of our queries. In the sections, we have defined many different abbreviations. Should we consider all of them for detecting standard queries? Fortunately, the answer is negative. It is enough to recognize the few basic axes and tests. For instance, the goal

```
Toy> ("name" /. text T) == R
```

yields:

```
R ->  child.::.nameT "name" ./. child.::.textT T
```

which has the expected form for standard XPath.

Now we are ready to define the function `standard` using higher-order patterns:

```
standard :: xPath -> bool
standard A         = step A
standard (A ./. B) = step A /\ standard B

step :: xPath -> bool
step (Axis.::.Test)    = (axis Axis) /\ (test Test)

test :: xPath -> bool
test A                 = simpleTest A
test (A .# B)          = simpleTest A /\ standard B

% Axes
axis :: xPath -> bool
axis A                 = (A==child) \/ (A==self) \/ (A==descendant)

simpleTest :: xPath -> bool
simpleTest nodeT       = true
simpleTest (nameT S)   = true
simpleTest (textT S)   = true
simpleTest (commentT S)= true
```

Function `standard` succeeds if the query is either a single step of several steps combined by the operator `./.`. Steps are defined by an axis and a test connected by `(.::.)`. We also consider filters, which must be standard XPath queries. Finally, the definition of functions `test`, `test` and `axis` is self-explanatory. For instance the goal:

```
Toy> standard ("food" /. name "item")
```

produces the answer `yes`, but

```
Toy> standard (nodeT.::.child)
```

produces `no` meaning that the query is not standard.

It can be interesting to redefine the operator ⟵ defined in Section 4.2 for applying a query to the XML document stored in a given file. The new version checks first if the query is standard, failing otherwise.

```
infix  20 <--!
(<--!) :: string -> xPath -> xml
S <--! F =  if standard F then S <-- F
```

For instance the goal:

```
Toy> ("food.xml" <--! (nodeT.::.child)) == R
    no
    Elapsed time: 0 ms.
```

fails because the query is not standard, while

```
Toy> ("food.xml" <--! (child.::.nodeT)) == R

      { R -> (xmlTag  "food" []
      [ (xmlTag  "item" [ (xmlAtt  "type"  "fruit") ]
          [ (xmlTag  "name" [] [ (xmlText  "watermelon") ]),
            (xmlTag  "price" [] [ (xmlText  "32") ]) ]),
        ...
        (xmlTag  "item" [ (xmlAtt  "type"  "fruit") ]
        [ (xmlTag  "name" [] [ (xmlText  "strawberries") ]),
          (xmlTag  "variety" [] [ (xmlText  "alpine") ]),
          (xmlTag  "price" [] [ (xmlText  "210") ]) ]) ]) }
      Elapsed time: 0 ms.
sol.1, more solutions (y/n/d/a) [y]?
      no
      Elapsed time: 0 ms. == R
```

will return the expected result.

6.2 Introducing Reverse Axes

The queries defined so far only use forward axes such as `descendant` or `child`. However, in XPath reverse axes such as `parent` and `ancestor` are also allowed. Implementing these axes is not trivial in our approach, since each `xPath` function

receives as input the fragments of the XML document that satisfied the previous steps. This fragments corresponds to a subtree of the XML document. Therefore it is not possible to obtain the `parent` of the current XML fragment. A possible solution is to include the whole XML document and a representation of the path leading to the context node as input parameters. Nevertheless, this complicates the implementation, and the simple definitions of the previous sections would be no longer valid. An alternative is to preprocess the query, replacing the ascending axes by predicate filters, as shown in [7, 8].

In our case we will use the following rules for removing `parent` and `ancestor` outside filter predicates, although the same approach can be extended to `parent` and `ancestor` in filter predicates, and to `following-sibling` (see [8] for the equations in these cases).:

$$
\begin{array}{lll}
(P_1) & \text{child::}T_1\text{/S/parent::}T_2 & \equiv \text{self::}T_2\text{[child::}T_1\text{/S]} \\
(P_2) & \text{descendant::}T_1\text{/S/parent::}T_2 & \equiv \text{self::}T_2\text{[child::}T_1\text{/S]} \\
(P_3) & \text{descendant::}T_1\text{/S/parent::}T_2 & \equiv \text{descendant::}T_2\text{[child::}T_1\text{/S]}
\end{array}
$$

$$
\begin{array}{lll}
(A_1) & S_1/S_2/\text{ancestor::}T & \Rightarrow S_1/\text{self::}T[S_2] \\
(A_2) & \text{descendant::}T_1/\text{ancestor::}T_2 & \Rightarrow \text{descendant::}T_2\text{[descendant::}T_1\text{]}
\end{array}
$$

where T_1, T_2 are tests that optionally can include filters, and S is a (possibly empty) sequence of steps using the `self` axis. S_1, S_2 are sequences of steps such that S_2 is not empty and its first step is not of the form `self`::T. For instance the relative location path `child::variety/parent::node()` is transformed by (P_1) into the equivalent expression `self::node()[child::variety]`. The equations are implemented in $\mathcal{TOY}$ through the program rules for `delParent` which can be found in Figure 6.1. The first program rule is used for skipping the sequence S, while the three following rules resemble closely $(P_1), (P_2), (P_3)$ when S is the empty sequence.

In order to apply these functions we change the definition of the operator `<--`, which now preprocesses the query before applying it to the XML document:

```
S <-- F =  (preprocess F) (load_xml_file S).
```

Then we define an initial version of `parent` that indicates that it will fail without preprocessing:

```
parent :: xPath
parent S =  if false then S
```

Function `preprocess` uses a version of the well-known catamorphism `fold` acting over XPath queries to apply a function `transform` to each individual steps, which

in turn employs `delParent` as auxiliary function. The result is obtained with the steps associated to the left, as in `(`S_1`./.`S_2`)./.`S_3. This is corrected by function `rev` which is the analogous of the reverse function used in functional program for lists. All this code is possible thanks to the use of higher-order patterns. The next example looks for nodes having at least one "variety" child.

XPath $\rightarrow$ `doc("food.xml")/food//variety/parent::node()`
$\mathcal{TOY}$ $\rightarrow$ `name "food".//."variety"/.parent.::.nodeT`

```
%%%%%%%%%%%%%%%%%%%%%%% reverse axes %%%%%%%%%%%%%%%%%%%%%%
undefined S = if false then S
parent::xPath
parent S =  undefined S
ancestor::xPath
ancestor S =  undefined S

preprocess::xPath -> xPath
preprocess A = rev (foldl transform id  A)

transform::xPath -> xPath -> xPath
transform X (self.::.T)       =  X ./.(self.::.T)
transform X (child.::.T)      =  X ./.(child.::.T)
transform X (descendant.::.T) =  X ./.(descendant.::.T)
transform X (ancestor.::.T)   = removeAncestor  X T
transform X (parent.::.T)     = delParent  X T


addFilter:: xPath->xPath->xPath
addFilter (X./.A.::.(T.#F)) G = X ./. (A.::. (T.# (F ./. G)))

delParent::  xPath -> xPath ->xPath
delParent (X./.self.::.T1) T2 =  addFilter (delParent X T2) (self.::.T1)
delParent (X./.child.::.T1) T2      = X./. self.::.(T2.#(child.::.T1))
delParent (X./.descendant.::.T1) T2 = X./. self.::.(T2.#(child.::.T1))
delParent (X./.descendant.::.T1) T2 = X./. descendant.::. (T2.#(child.::.T1))


removeAncestor::  xPath -> xPath ->xPath
removeAncestor (X./.Y)                T  = insertFilter (Y ./. id) X T
removeAncestor (X./.descendant.::.T1) T2 =
                                  X./. (descendant.::.T2.#(descendant.::.T1))

insertFilter::xPath -> xPath -> xPath -> xPath
insertFilter ((A.::.B)./.Y) X T = if A /= self then
                                addFilter (X./.self.::.T) ((A.::.B)./.Y)
insertFilter Y (X./.(A.::.B)) T = insertFilter ((A.::.B)./.Y) X T


rev::xPath -> xPath
rev (A.::.B) = A.::.B
rev (F./.G) = rev' F G
rev' (A.::.B) G = (A.::.B) ./. G
rev' (X ./. Y) G = rev' X (Y./. G)


foldl::(xPath->xPath->xPath)->xPath->xPath->xPath
foldl F Z (A.::.T) = F Z (A.::.T)
foldl F Z (G ./. H) = foldl F (F Z G) H
```

Chapter 7

Conclusions and Future Work

We have shown how the declarative nature of XPath fits in a very natural way in functional-logic languages. XPath queries are represented in this setting by non-deterministic higher-order expressions, thus becoming first-class citizens of the language that can be readily extended and adapted by the programmer. In the case of the functional-logic language $\mathcal{TOY}$, the possibility of using higher-order patterns make this affirmation even more valid. We have shown in Chapter 6 two applications of this feature. The result is enriching for both XPath and $\mathcal{TOY}$ users:

- For the users of the functional-logic $\mathcal{TOY}$ the advantage is clear: they can use XPath queries in their programs in a natural way. The queries are written in their usual language and thus using them requires little effort. Moreover, since the combinators, tests and axes are available they can be freely modified and extended. The situation can be compared to the introduction of parsers in functional [5] and functional-logic languages [2].
- From the point of view of the XPath apprentices, the tool can be useful, specially if they have some previous knowledge of declarative languages. The possibility of generating test cases for XPath queries is an easy and powerful tool that can be very helpful for understanding the basics of XPath.
- The framework can also be interesting for designers of XPath environments, because it allows the users to define easily prototypes of new features such as new combinators or functions.

Our proposal also contains some drawbacks that deserve to be discussed. First of all, the syntax of the queries resembles quite closely XPath, but the differences can be confusing at first. However, in our experience this difficulty is soon overcome by practice, and in any case is easy to write a parser converting standard XPath

format to the format explained in this paper. Another difficulty arises from the implementation of features using the position of the node in the sequence. This features can be introduced in our non-deterministic setting, but only using some impure primitive like `collect` that bundles in a list the results of a non-deterministic expression. The problem with this impure primitive is that cannot deal with logic variables, which can be a problem for instance for the generation of test cases.

Appendix A

The XPath library

In Appendix we list the source code corresponding to the toyXPath library explained in previous chapters.

```
%%%%%%%%%%%%%%%% auxiliary functions %%%%%%%%%%%%%%
member :: [A] -> A
member  [X | Xs] =  X ? member Xs

infixr 30 ?
(?):: A -> A -> A
X ? Y = X
X ? Y = Y

nth ::  int -> [A] -> A
nth N [X|Xs]  = if N==1 then X else nth (N-1) Xs

val::[char] -> int
val [X] = ord(X) - ord('0')
val [X|Xs] = val(Xs) + val([X])*10^(length Xs)

first (nth N)  = N==1


infixr 30 /\
false /\ X = false
true /\ X = X
```

```
infixr 30 \/
true \/ X = true
false \/ X = X

length [] = 0
length [X|Xs] = 1 + length Xs


%%%%%%%%%%%%%%%%%%%%%%%% DATA TYPES %%%%%%%%%%%%%%%%%%%%%%%%%%%%
% datatype representing XML documents
type string       = [char]
type xmlName      = string
type xml          = xmlNode

% xpath queries are functions taking and returning xml fragments
type xPath = xml-> xml

%%%%%%%%%%%%% starting point %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

% xml documents represented in Toy start with a "xml" tag
infix  20 <--
(<--)::string  -> xPath -> xml
S <-- F =  (preprocess F)  (load_xml_file S)


% allowing only queries following the standard form
infix  20 <--!
(<--!)::string  -> xPath -> xml
S <--! F =  if standard F then S <-- F

%%%%%%%%%%%%%%%%%%%%%%%% connecting elements %%%%%%%%%%%%%%%%%%

%%% connecting axes and node/name tests (:: in Xpath)
infixr  55 .::.
(.::.):: xPath -> xPath -> xPath
(F .::. G) X  = G (F X)

%% connecting steps ( / in xpath)
infixr 40 ./.
(./.):: xPath -> xPath -> xPath
(F ./. G) X = G (F X)

%%%%%%%%%% a few tests %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%% name test
```

```
nameT:: xmlName ->  xPath
nameT S (xmlTag S Atr L ) = xmlTag S Atr L


% node test
nodeT:: xPath
nodeT X = X

% text elements
textT:: string -> xPath
textT S (xmlText S) = xmlText S

% comments
commentT::string -> xPath
commentT S (xmlComment S) = xmlComment S

% element test
elem:: xPath
elem  = nameT _


%%%%%%%%%%%% a few axes %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

self,child,descendant,descendant_or_self,ancestor_or_self :: xPath

self X = X

child (xmlTag _Name _Attr L) = member L

descendant X = child X
descendant X = if (child X == Y) then descendant Y

descendant_or_self = self
descendant_or_self = descendant

ancestor_or_self = self
ancestor_or_self = ancestor

%%%%%%%%%%%%% the identity step %%%%%%%%%%%%%%%%%%%%%%%%%
id :: xPath
id = self.::.nodeT

%%%%%%%%%%%%% filters %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
```

```
infixr  60 .#
(.#)::xPath -> xPath -> xPath
(P .# F)  X =  if F Y == R then Y  where Y = P X

infixr  60 #
(#)::string -> xPath -> xPath
S # F =child.::.(nameT S).# F

check :: bool->xPath
check C  = if C==true then id

infixr  60 .!
(.!)::xPath -> (int -> int) -> xPath
(P .!  F)  X = nth (F N) L
               where L = collect (P X)
                     N = length L

pos:: int -> int -> int
pos N _ = N

lastMinus:: int -> int -> int
lastMinus M N = N-M

lastPos:: int -> int
lastPos = lastMinus 0

infixr  50 !
(!)::string -> (int -> int) -> xPath
S ! F = child.::.nameT S.! F

%%%%%%%%%%%%%% attribute filter  %%%%%%%%%%%%%%%%%%%%%%%%%

(@=) :: xmlName -> xmlName -> xPath
(@=) S V X = if (xmlAtt S V == member Attr) then X
             where (xmlTag _Name Attr _L) = X

%%%%%%%%%%%%% abbreviatures %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

% self::node() = . in xpath
%(.):: xPath
%(.) = self .::. nodeT

% X is any descendant of the context node or
% the context node itself (//X in xpath)
```

```
infixr  30 .//.

(.//.)::  xPath -> xPath -> xPath
A .//. B = append A (descendant_or_self .::. nodeT ./. B)

append::xPath -> xPath -> xPath
append (A.::.B) C = (A.::.B) ./. C
append (X ./.Y) C = X ./. (append Y  C)

infixr 35 //.
(//.)::  string -> xPath -> xPath
A //. B = append (child .:: A) (descendant_or_self.::.nodeT./.B)

% return the text
text :: string -> xPath
text A  =  (child .::. textT A)

% name abbreviature
name:: string -> xPath
name S  =  child .::. (nameT S)

infixr  55 .::
(.::):: xPath -> string -> xPath
F .:: G   = (F .::. (nameT G))

infixr 35 /.
(/.):: string -> xPath -> xPath
S /. X = name S ./. X

infixr 35 ./
(./):: xPath -> string -> xPath
X ./ S =  append X (child.::S)

infixr 40 .=
(.=):: string -> string ->xPath
(.=) A B  = (A /. text B)

infixr 40 .>
(.>):: string -> int ->xPath
(.>) A B X =  if (((A /. text C) X == Y) /\ (val C > B)) then Y

%%%%%%%% generating Test Cases %%%%%%%%%%%%%%%%%%%%%%%%%%%%

generateTC:: xPath -> string -> bool
```

```
generateTC F FileName = if (F X == _) then  write_xml_file X FileName


%%%%%%%%%%%%%%%%%%%%%%% reverse axes %%%%%%%%%%%%%%%%%%%%%%
undefined S = if false then S
parent::xPath
parent S =  undefined S
ancestor::xPath
ancestor S =  undefined S

preprocess::xPath -> xPath
preprocess A = rev (foldl transform id  A)

transform::xPath -> xPath -> xPath
transform X (self.::.T)       =  X ./.(self.::.T)
transform X (child.::.T)      =  X ./.(child.::.T)
transform X (descendant.::.T) =  X ./.(descendant.::.T)
transform X (ancestor.::.T)   = removeAncestor  X T
transform X (parent.::.T)     = delParent  X T


addFilter:: xPath->xPath->xPath
addFilter (X./.A.::.(T.#F)) G = X ./. (A.::. (T.# (F ./. G)))

delParent::  xPath -> xPath ->xPath
delParent (X./.self.::.T1) T2 =  addFilter (delParent X T2) (self.::.T1)
delParent (X./.child.::.T1) T2      = X./. self.::.(T2.#(child.::.T1))
delParent (X./.descendant.::.T1) T2 = X./. self.::.(T2.#(child.::.T1))
delParent (X./.descendant.::.T1) T2 = X./. descendant.::. (T2.#(child.::.T1))


removeAncestor::  xPath -> xPath ->xPath
removeAncestor (X./.Y)                T  = insertFilter (Y ./. id) X T
removeAncestor (X./.descendant.::.T1) T2 =
                            X./. (descendant.::.T2.#(descendant.::.T1))

insertFilter::xPath -> xPath -> xPath -> xPath
insertFilter ((A.::.B)./.Y) X T = if A /= self then
                             addFilter (X./.self.::.T) ((A.::.B)./.Y)
insertFilter Y (X./.(A.::.B)) T = insertFilter ((A.::.B)./.Y) X T


rev::xPath -> xPath
rev (A.::.B) = A.::.B
```

```
rev (F./.G) = rev' F G
rev' (A.::.B) G = (A.::.B) ./. G
rev' (X ./. Y) G = rev' X (Y./. G)


foldl::(xPath->xPath->xPath)->xPath->xPath->xPath
foldl F Z (A.::.T) = F Z (A.::.T)
foldl F Z (G ./. H) = foldl F (F Z G) H

%%%%%%%%%%%%%% standard queries  %%%%%%%%%%


standard,step,test:: xPath -> bool
simpleTest,axis::xPath -> bool

standard A         = step A
standard (A ./. B) = step A /\ standard B

step (Axis.::.Test) =  (axis Axis) /\ (test Test)

test A        = simpleTest A
test (A .# B) = simpleTest A /\ standard B
test (A .! B) = simpleTest A

simpleTest nodeT     = true
simpleTest (nameT _) = true
simpleTest (textT _) = true
simpleTest (commentT _)=true

axis A = (A==child) \/ (A==self) \/ (A==descendant)
                    \/ (A==ancestor) \/ (A== parent)

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
```

Bibliography

[1] J. M. Almendros-Jiménez. An Encoding of XQuery in Prolog. In *XSym '09: Proceedings of the 6th International XML Database Symposium on Database and XML Technologies*, pages 145–155, Berlin, Heidelberg, 2009. Springer-Verlag.

[2] R. Caballero and F. López-Fraguas. A functional-logic perspective on parsing. In *FLOPS '99: Proceedings of the 4th Fuji International Symposium on Functional and Logic Programming*, pages 85–99, London, UK, 1999. Springer-Verlag.

[3] R. Guerra, J. Jeuring, and S. D. Swierstra. Generic validation in an xpath-haskell data binding. In *Proceedings Plan-X*, 2005.

[4] M. Hanus. Curry: An Integrated Functional Logic Language (version 0.8.2 march 28, 2006). Available at: `http://www.informatik.uni-kiel.de/~mh/curry/`, 2003.

[5] G. Hutton and E. Meijer. Monadic parsing in haskell. *J. Funct. Program.*, 8(4):437–444, 1998.

[6] F. J. López-Fraguas and J. S. Hernández. TOY: A Multiparadigm Declarative System. In *RTA '99: Proceedings of the 10th International Conference on Rewriting Techniques and Applications*, pages 244–247, London, UK, 1999. Springer-Verlag.

[7] D. Olteanu. Spex: Streamed and progressive evaluation of xpath. *IEEE Transactions on Knowledge and Data Engineering*, 19:934–949, 2007.

[8] D. Olteanu, H. Meuss, T. Furche, and F. Bry. Symmetry in xpath. Technical Report Research Report PMS-FB-2001-16, Computer Science Institute, Munich, Germany, 2001.

[9] D. Olteanu, H. Meuss, T. Furche, and F. Bry. Xpath: Looking forward. In *Workshop on XML-Based Data Management*, pages 109–127. Springer, 2002.

[10] W3C. Extensible markup language (xml), 2007.

[11] W3C. Xml path language (xpath) 2.0, 2007.

[12] W3C. XQuery 1.0: An XML query language, 2007.

[13] P. Walmsley. *XQuery*. O'Reilly Media, Inc., 2007.