Available online at www.sciencedirect.com

SciVerse ScienceDirect

Electronic Notes in Theoretical Computer Science

Electronic Notes in Theoretical Computer Science 282 (2012) 19–34

www.elsevier.com/locate/entcs

XPath Query Processing in a Functional-Logic Language

J.M. Almendros-Jiménez[a,1], R. Caballero[b,1], Y. García-Ruiz [b,1], F. Sáenz-Pérez [c,1],

[a] *Dpto. de Lenguajes y Computación*
Universidad de Almería, Spain

[b] *Dpto. de Sistemas Informáticos y Computación*
Universidad Complutense de Madrid, Spain

[c] *Dpto. de Ingeniería del Software e Inteligencia Artificial*
Universidad Complutense de Madrid, Spain

Abstract

XPath is a well-known query language for finding and extracting information from XML documents. This paper shows how the characteristics of this domain-specific language fits very well into the functional-logic paradigm. The proposed framework allows the user to write XPath-like queries as first-class citizens of the functional-logic language $\mathcal{TOY}$, using higher-order combinators for constructing the queries and non-determinism in order to obtain the different answers that XPath queries can return. The result is a very good example of cross-fertilization of two different areas. In the case of $\mathcal{TOY}$, the users can now integrate XML queries in their programs without using any external library or ad hoc interface. In the case of XPath, the use of higher-order patterns allow us to define functions for easily processing the queries. In particular, the paper shows how to trace and debug erroneous queries, and how to detect that a query is a refinement of another query, which can be useful for improving the efficiency of query processing.

Keywords: Functional-Logic Programming, Non-Deterministic Functions, XPath, Higher-Order Patterns

[1] This work has been supported by the Spanish projects STAMP (TIN2008-06622-C03-01, TIN2008-06622-C03-03), S-0505/TIC/0407, Prometidos-CM (S2009TIC-1465), and GPD (UCM-BSCH-GR35/10-A-910502)

doi:10.1016/j.entcs.2011.12.003

1 Introduction

In the last years the eXtensible Markup Language XML [18] has become the *de facto* standard for exchanging structured data in plain text files. This was the key for its success as data structures are revealed and therefore they are readily available for its processing (even with usual text editors if one wishes to manually edit them). Structured data means that new, more involved access methods must be devised. XQuery [21,23] has been defined as a query language for finding and extracting information from XML documents. It extends XPath [19], a domain-specific language 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. Due to its acknowledged importance, XML and its query languages have been embodied in many applications as in database management systems, which include native support for XML data and documents both in data representations and query languages (e.g., Oracle and SQL Server). Some of them extend SQL to include support for XQuery, so that results from XML queries can be used by the more declarative SQL language in the context of a database, making possible to share relational and XML data sources.

Many general-purpose declarative programming languages include support for XPath and XQuery. In the functional programming area, works about Haskell can be found in [17,2,22,16]. There are also proposals based on logic programming as [15,14,3,12,7,13]. In the field of functional-logic languages, [10] proposes a rule-based language for processing semistructured data that is implemented and embedded in the functional logic language Curry [9]. Recently, in [5], we have proposed an implementation of XPath in the functional-logic language $\mathcal{TOY}$ [11], where a XPath query becomes at the same time implementation (code) and representation (data term). XML documents are represented in this proposal by means of data terms, and the basic constructors of XPath: `child`, `self`, `descendant`, etc. are defined as non-deterministic higher-order functions that can be applied to XML terms.

This paper continues this work, showing that XPath fits very well into the functional-logic paradigm. The proposed framework allows the user to write XPath-like queries as first-class citizens of the functional-logic language $\mathcal{TOY}$. To this end, higher-order combinators are used for constructing queries, and we take advantage of non-determinism in order to obtain the different answers that XPath queries may return. The result is a very good example of cross-fertilization of two different areas:

- In the case of $\mathcal{TOY}$ (introduced in Section 2), users can now integrate XML queries in their programs without using any external library or *ad hoc*

interface.

- In the case of XPath (introduced in Section 3), the use of higher-order patterns allows us to define functions for processing easily queries.

Justification for the embedding is also provided in this paper in two forms, which constitute our main contributions: First, we show in Section 4 how to debug and trace erroneous queries. Second, we apply the idea of *compensation* [24] in the field of relational databases to our scheme. This refers to the ability of reusing previous cached query results for enhancing query solving performance (Section 5). Finally, in Section 6 we draw some conclusions and point out some future work.

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

A $\mathcal{TOY}$ [11] program is composed of data type declarations, type alias, infix operators, function type declarations and defining rules for functions symbols. The syntax of (total) *expressions* in $\mathcal{TOY}$ $e \in Exp$ is $e ::= X \mid h \mid (e\ e')$ where X is a variable and h either a function symbol or a data constructor. Expressions of the form $(e\ e')$ stand for the application of expression e (acting as a function) to expression e' (acting as an argument). Similarly, the syntax of (total) *patterns* $t \in Pat \subset Exp$ can be defined as $t ::= X \mid c\, t_1 \ldots t_m \mid f\, t_1 \ldots t_m$ where X represents a variable, c a data constructor of arity greater or equal to m, and f a function symbol of arity greater than m, while the t_i are patterns for all $1 \leq i \leq m$.

Data type declarations and type alias are useful for representing XML documents in $\mathcal{TOY}$:

```
data node      = txt     string
               | comment string
               | tag     string [attribute] [node]
data attribute = att     string string
type xml       = node
```

The data type `node` represents nodes in a simple XML document. It distinguishes three types of nodes: texts, tags (element nodes), and comments, each one represented by a suitable data constructor and with arguments representing the information about the node. For instance, the constructor `tag` includes the tag name (an argument of type `string`) followed by a list of attributes, and finally a list of child nodes. The data type `attribute` contains the name of the attribute and its value (both of type `string`). The last type alias, `xml`, renames the data type `node`. Of course, this list is not exhaustive, since it misses several types of XML nodes, but it is enough for this presentation. Figure 1 in next page shows an XML document and its representation

in $\mathcal{TOY}$.

```
<?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>
```

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

Fig. 1. XML example (left) and its representation in $\mathcal{TOY}$ (right)

Each rule for a function f in $\mathcal{TOY}$ has the form:

$$\underbrace{f\ t_1 \ldots t_n}_{\text{left-hand side}} \rightarrow \underbrace{r}_{\text{right-hand side}} \Leftarrow \underbrace{e_1, \ldots, e_k}_{\text{condition}} \text{ where } \underbrace{s_1 = u_1, \ldots, s_m = u_m}_{\text{local definitions}}$$

where e_i, u_i and r are expressions (that can contain new extra variables) and t_i, s_i are patterns.

In $\mathcal{TOY}$ variable names must start with either an uppercase letter or an underscore (for anonymous variables), whereas other identifiers start with lowercase. $\mathcal{TOY}$ includes two primitives for loading and saving XML documents, called `load_xml_file` and `write_xml_file` respectively. For convenience all the documents are started with a dummy node `root`. This is useful for grouping several XML fragments. If the file contains only one node `N` at the outer level, `root` can be removed defining the following simple function:

```
load_doc F = N <== load_xml_file F == tag "root" [att "version"
"1.0"] [N]
```

where `F` is the name of the file containing the document. Observe that the strict equality `==` in the condition forces the evaluation of `load_xml_file F` and succeeds if the result has the form `tag "root" [att "version" "1.0"] [N]` for some `N`. If this is the case, `N` is returned.

3 Representing XPath Queries

This section introduces the subset of XPath that we intend to integrate with $\mathcal{TOY}$, omitting all the features of XPath that are supported by $\mathcal{TOY}$ but not used in this paper, such as preprocessing of reverse axes. See [5,4] for a more detailed introduction to XPath in $\mathcal{TOY}$.

Typically, XPath expressions return several fragments of the XML document. Thus, the expected type in a functional language for `xPath` could be `type xPath = xml -> [xml]` meaning that a list or sequence of results is obtained. This is the approach considered in [1] and also the usual in functional programming [8]. However, in our case we take advantage of the non-deterministic nature of our language, returning each result individually. We define a 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`. In order to apply a XPath expression to a particular document, we use the following infix operator definition:

```
infix  20 <--
(<--)::string  -> xPath -> xml
S <-- Q = Q (load_xml_file S)
```

The input arguments of this operator are a string `S` representing the file name and a XPath query `Q`. The function applies `Q` to the XML document contained in file `S`. This operator plays in $\mathcal{TOY}$ the role of `doc` in XPath. The XPath combinators `/` and `::` which correspond to the connection between steps and between axis and tests, respectively, are defined in $\mathcal{TOY}$ as function composition:

```
infixr 55 .::.                          infixr 40 ./.
(.::.) :: xPath -> xPath -> xPath       (./.) :: xPath -> xPath -> xPath
(F .::. G) X  = G (F X)                 (F ./. G) X = G (F X)
```

The function operator names `.::.` and `./.` are employed because the standard XPath separators `::` and `/` are already defined in $\mathcal{TOY}$ with a different meaning. Notice that the two definitions are the same since they stand for the application of a XPath expression to another XPath expression and return also a XPath expression, although they are intended to be applied to different fragments of XPath: `./.` for steps and `.::.` for combining axes and tests producing steps. Indeed, we would use a single operator for representing both combinators, but we decided to do this way for maintaining a similar syntax for XPath practitioners, more accustomed to use such symbols. In addition, we do not check for the "appropriate" use of such operators and either rely on the provided automatic translation by the parser or left to the user. The variable `X` represents the input XML fragment (the context node). The rules specify how

```
self,child,descendant ::  xPath
descendant_or_self ::  xPath
self X = X
child (tag _ _ L) = member L
descendant X = child X
descendant X =
       if child X == Y
       then descendant Y
descendant_or_self =
       self ?  descendant
```

```
nodeT,elem ::  xPath
nameT,textT,commentT ::
         string->xPath
nodeT X = X
nameT S (tag S Att L) =
         tag S Att L
textT S (txt S) = txt S
commentT S (comment S) =
         comment S
elem = nameT _
```

Fig. 2. XPath axes and tests in $\mathcal{TOY}$

the combinator applies the first XPath expression (`F`) followed by the second one (`G`). Figure 2 shows the $\mathcal{TOY}$ definition of XPath main axes and tests. The first one is `self`, which returns the context node. In our setting, it corresponds simply to the identity function. A more interesting axis is `child` which returns, using the non-deterministic function `member`, all the children of the context node. Observe that in XML only *element nodes* have children, and that in the $\mathcal{TOY}$ representation these nodes correspond to terms rooted by constructor `tag`. Once `child` has been defined, `descendant` and `descendant-or-self` are just generalizations. The first rule for this function specifies that `child` must be used once, while the second rule corresponds to two or more applications of `child`. In this rule, the `if` statement is employed to ensure that `child` succeeds when applied to the input XML fragment, thus avoiding possibly infinite recursive calls. Finally, the definition of axis `descendant-or-self` is straightforward. Observe that the XML input argument is not necessary in this natural definition. With respect to test nodes, the first test defined in Figure 2 is `nodeT`, which corresponds to `node()` in the usual XPath syntax. This test is simply the identity. 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`. For instance, the answer to:

```
Toy> collect ("food.xml" <-- descendant_or_self.::.nodeT) == R
```

produces a single answer, with `R` instantiated to a list whose elements are the nodes in `"food.xml"`. XPath abbreviated syntax allows the programmer to omit the axis `child::` from a location step when it is followed by a name. Thus, the query `child::food/child::item/child::price` becomes in XPath simply as `food/item/price`. 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 new unitary operators `name` (and similarly `text`), which transform strings into XPath expressions:

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

So, we can write in $\mathcal{TOY}$ `name "food"./.name "item"./.name "price"`. Other tests as `nameT` and `textT` *select* fragments of the XML input, which can be returned in a logical variable, as 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` obtains the prices contained in the example document. Another XPath abbreviation is `//` which stands for `/descendant-or-self::node()/`. 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 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 usefulness of higher-order patterns since for instance pattern `(A.::.B)` has type `xPath`, i.e., `xml -> xml`.

Optionally, XPath tests can include a predicate or filter. Filters in XPath are enclosed between square brackets. In $\mathcal{TOY}$, they are 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`, 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. Multiple predicates can be chained together to filter items as with the `and` operator, which can be formulated as follows:

```
infixr 60 /&
(X /& Y) Z = if X Z==_ then Y Z
```

4 Debugging XPath Queries

One of the most appealing features of our setting is that XPath queries can be manipulated. In this section we use this feature for tracing and debugging queries. We distinguish two types of possible errors in a XPath Query depending on the erroneous result produced: *wrong* queries when the query returns an unexpected result, and *missing* when the query does not produce some expected result. We present a different proposal depending on the error.

4.1 Wrong XPath Queries

Consider for instance the goal `("bib.xml" <-- name "bib" ./. name "book" ./. name "author" ./. name "last" ) == R` and suppose that it produces the unexpected answer `R -> (tag "last" [] [ (txt "Abiteboul") ])` (see [20], sample data 1.1.2 and 1.1.4 to check the structure of these documents and an example). If the error is just some misspelling of the author's last name it is easy to look for the wrong information in the document in order to correct the error. However in some situations, and in particular when dealing with complicated, large documents, the error can be in the XPath query, that has selected an erroneous path in the document. In these cases it is also useful to find the answer in the document and then trace back the XPath query until the error is found. Observe that the erroneous answer can be just one of the produced answers (in the example the query can produce many other, expected, answers), and that we are interested only in those portions of the document that produce this unexpected result.

In $\mathcal{TOY}$ we can obtain each intermediate step with its associated answer by defining a suitable function `wrong` that receives three arguments: the query, its input (initially the whole document) and the unexpected output (initially the unexpected answer). The implementation is straightforward:

```
wrong (A.::.B) I O = [((A.::.B), I, O)] <== (A.::.B) I == O
wrong (A./.B) I O  = [(A, I, O1 ) | wrong B O1 O]
```

```
<== A I == O1, B O1 == O
```

In the case of a single step (`A.::.B`) the first rule checks that indeed the step applied to the input produces the output returning the three elements. In the case of two or more steps, the second rule looks for the value `O1` produced by the single step `A` such that the rest of the query `B` applied to `O1` produces the erroneous result `O`. The variable `O1` is a new logic variable, and that the code uses the *generate and test* feature typical of functional languages. The function `wrong` produces a list where each step is associated with its input and its output. However, using `wrong` directly produces a verbose, difficult to understand output due to the representation of XML elements as a data terms in $\mathcal{TOY}$. This can be improved by building a new XML document containing all the information and saving it to a file using the primitive `write_xml_file`:

```
traceStep (Step,I,O) = tag "step" [] [ tag "query" []
                       [txt (show Step)],
                        tag "input" [] [I],
                        tag "output" [] [O] ]

generateTrace L = tag "root" [] (map traceStep (rev L))

writeTrace XPath InputFile WrongOutput OutputFile =
      write_xml_file (
           generateTrace (
              wrong XPath (
                 load_xml_file InputFile)
              WrongOutput))
           OutputFile

show (A.::.B)    = (show A)++".::."++(show B)
show nodeT       = "nodeT"
...
show child      = "child"
...
```

The first function `traceStep` generates an XML element `step` containing the information associated with a XPath step: the step combinator, its input, and its output. This function uses the auxiliary function `show` to obtain the string representation of the step (only part of the code of this function is displayed). Then function `generateTrace` applies `traceStep` to a list of steps. It uses the functions `map` and `rev` whose definition is the same as in functional programming. In particular, `rev` is employed to ensure that the last step of the query is the first in the document, which is convenient for tracing back the result. Finally `writeTrace` combines the previous functions. It receives four parameters: the query, the name of the initial document, the unexpected XML fragment we intend to trace, and the name of the output

file where the result is saved. Now we can try the goal:

```
Toy> writeTrace (name "bib" ./. name "book" ./. name "author" ./.
                  name "last" ) "bib.xml"
                (tag  "last" [] [txt  "Abiteboul"])
                "trace.xml"
```

In this case the symbol == is not used which indicates to $\mathcal{TOY}$ that the result of this expression must be true. After the goal is solved we can consult the document "trace.xml":

```
<step>
 <query>child.::.nameT last</query>
 <input>
  <author>
   <last>Abiteboul</last>
   <first>Serge</first>
  </author>
 </input>
 <output>
  <last>Abiteboul</last>
 </output>
</step>

<step>
 <query>child.::.nameT author</query>
 <input>
  <book year="2000">
   <title>Data on the Web</title>
   <author>
    <last>Abiteboul</last>
```

For the sake of space only the first step and part of the second step is displayed above, although the document contains all the information that allows the user to trace the query.

4.2 Missing XPath Queries

Sometimes a XPath query produces no answer, although some result was expected. For instance the goal "food.xml" <-- name "food" ./. name "item" ./. name "type" ./. child.::.textT "navel" == R simply fails in $\mathcal{TOY}$. The reason is that the user wrote type where it should be variety. This error is very common, and the source of the error can be difficult to detect. Observe that the previous idea of tracing the result back cannot be applied because there is no result to trace. For these situations we propose trying the XPath query without the last step, then if it fails without the last two steps and so on. The idea is to find the first step that produces an empty result, because it is usually the source of the error.

```
missing  (A.::.B) R = (A.::.B)
missing (X ./. Y) R = if (collect (X R) == []) then X
                      else missing Y (X R)
```

The previous definition of function `missing` relies on the primitive `collect` that accumulates all the results of a function call in a list. Therefore `collect (X R) == []` means that `X` applied to the input XML fragment `R` fails. Now we can try the goal:

```
Toy> missing (name "food" ./. name "item" ./. name "type" ./.
              child.::.textT "navel") (load_xml_file "food.xml") == R
    { R -> child .::. (nameT  "type") }
```

The answer indicates that the step `child .::. (nameT "type")` is the possible source of the error. Thus this simply function is useful, but we can do better. Instead of simply returning the erroneous step we can try to *guess* how the error can be corrected. In the case of *name* tests as the example the error is usually the same erroneous string has been used. Replacing the string by a logic variable such that the query now succeeds can help to find the error. Therefore, we implement a second version of `missing`:

```
missing (A.::.B) R =  guess (A.::.B) self R

missing (Step ./. Y) R = if (collect (Step R) == [])
                         then guess Step Y R
                         else missing Y (Step R)

guess Step Y R = if Step==(A.::.nameT B)
                 then if (StepBis ./. Y) R == _
                      then ( Step, "Substitute "++B++" by "++C )
                      else (Step, "No suggestion")
                 else (Step, "No suggestion")
    where StepBis = (A.::.nameT C)
```

In this case `missing` returns a pair. The first element of the pair is the same as in the first version, and the second element is a suggestion produced by function `guess`. This function first checks if the `Step` is of the form `(A.::.nameT B)`. If this is the case, it replaces the name `B` by a new variable `C` and uses the condition in the second `if` statement (`(StepBis ./. Y) R == _`) to check if `C` can take any value such that the query does not fail. If such value for `C` is found the returned string proposes replacing `B` by `C`. Otherwise `"No suggestion"` is returned. Since function `guess` requires an argument `Y` with the rest of the XPath query, the basis case of `missing` (first rule) uses `self` to represent the identity query. Now we can try

```
Toy>  missing (name "food" ./. name "item" ./. name "type" ./.
               child.::.textT "navel") (load_xml_file "food.xml") == R
{ R -> (child .::. (nameT  "type"),  "Substitute type by variety") }
```

Thus, the debugger finds the erroneous step and proposes the correct solution.

5 Compensation

Now, we would like to show how to take advantage of our implementation based on a functional-logic language for preprocessing queries. For instance, one of the well-known procedures in databases is the so-called *compensation of views*. Let us suppose the case in which a certain user queries the XML database, obtaining a certain answer. Such answer is stored by the system (we can suppose that the answer is locally cached). Later, the user wants to refine the query. A suitable database manager should be able to compute the answer of the refined query from the previous (cached) answer in order to improve answer time and performance.

For instance, let V be the query `"food.xml" <--name "food" ./. name "item"`. Such query retrieves the items from the file "food.xml". Suppose the user wants to refine the first query with a new query P defined as `"food.xml" <-- name "food" ./. name "item".#(name "variety")`. The second query can be answered from the first one, since the user requests the items for which variety is specified. In other words, the second answer is a piece of the first answer. Therefore, the second query can be computed from a new query P' defined as `name "item".#(name "variety")` applied to the answer of the first query. In such a case, P' is called the *compensation* of V w.r.t. P. Compensation of XPath queries has been studied by some authors [24], including the case of multiple views [6]. There are two related problems to compensation: the existence of the compensation and to find a minimal compensation.

Following [24], the compensation to a unique view can be computed by defining a certain *concatenation* operator between queries. Such concatenation operator is defined as follows. Firstly, we need to consider the representation of XPaths by the so-called *tree patterns*. Such tree patterns are tree based representation of XPath expressions in which nodes are labels and edges are axes. We can restrict XPath expressions to the same case as [24], the so-called $XP^{\{/,//,*,[]\}}$ fragment, in which XPath expressions are built from label tests, child axes (`./.`), descendant axes (`.//.`), branches (`.#`) and wildcards (`descendant.::.nodeT`). For instance, let us suppose V to be the XPath expression `name "a" .# name "c" .//. name "b" .# name "f"`, which can be represented as in Figure 3 (b). Black nodes represent the *output node* of the XPath expression, that is, the tag(s) of the answer to the given XPath expression. Now, suppose the XPath expression P' defined as `descendant.::.nodeT .# name "e" .//. name "f"`, and represented by a tree pattern in Figure 3

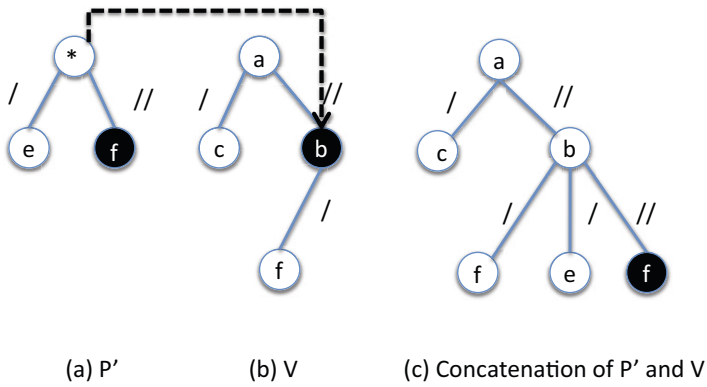

Fig. 3. Concatenation Operator

(a). Then the concatenation operator, denoted by $P' \oplus V$, is defined as in Figure 3 (c), representing `name "a" .# name "c" .//. name "b" .# (name "f" /& name "e") .//. name "f"`.

Basically, given two patterns P' and V, the concatenation of P' and V is constructed by merging the root of P' and the output node of V into one node. The root of V becomes the root of $P' \oplus V$, and the output node of P' becomes the output node of $P' \oplus V$. The merged node has as children both the children of the output node of V and the children of P'. When the nodes to be merged have different labels (for instance, '*' and b of Figure 3), the node assumes the more restrictive label (i.e., b), except when they have different label tests, which means that concatenation is not possible.

Following [24], the problem of finding a compensation of V w.r.t. P is equivalent to finding P' such that $P' \oplus V$ is equal to P. In order to compute in $\mathcal{TOY}$ the compensation of a given XPath expression we have defined the concatenation operator as a $\mathcal{TOY}$ function called `concat`, and the compensation can be defined as: `compensation V P = if concat P' V == P then P'` . It exploits the use of logic variables in $\mathcal{TOY}$, by computing the compensation of a given XPath expression. The function `concat` uses the representation of XPath by means of higher order patterns. The main rules are:

```
concat ((child .:::. nameT N) ./. P)
           (child .:::. nameT N) = (child .:::. nameT N) ./. P
concat P' (P ./. Q) = P ./. (concat P' Q)
concat P' (G .# F) =
             if F==S then (if Fl == self then Rt ./. S
                           else (Rt .# Fl) ./. S)
                     else (if Fl == self then (Rt .# F) ./. S
                           else (Rt .# (F /& Fl))./. S)
             where R = concat P' G
                   S = children R
```

```
Fl = filter R
Rt = root R
```

The first rule handles the merged node, assigning the children of the first argument to the second argument whenever they have the same label test. Second and third rules are recursive rules for handling steps and filters. Third rule accumulates filters by means of the XPath operator `/&`. The auxiliary functions `children`, `filter` and `root` compute the children and filter of the root, and the root of a XPath expression, respectively. Let us now see an example of use. For instance, defining:

```
v = name "food" ./. name "item"
p = name "food" ./. name "item" .# (name "variety")
```

```
Toy> compensation v p == P'
{ P' -> (child.::.(nameT "item")).#(child.::.(nameT "variety")) }
{ P' -> (descendant.::.nodeT).#(child.::.(nameT "variety")) }
```

We can see that we obtain two answers corresponding to `/item[variety]` and `/*[variety]` in the standard XPath syntax. In summary, we are able to use the logic features of $\mathcal{TOY}$ for building the compensation of a certain view. Currently, $\mathcal{TOY}$ is able to solve one of the problems related to compensations: to find them. $\mathcal{TOY}$ offers as output a set of compensations when they exist, otherwise fails. To find a minimal compensation is considered as future work of this implementation. Moreover, we would like to extend our implementation to cover with compensations to multiple views.

6 Conclusions

This paper shows how the framework for introducing XPath in the functional-logic language $\mathcal{TOY}$ allows us to define readily simple but powerful applications such as preprocessing, tracing and debugging. The implementation makes use of the main features of the language including:

- *Non-determinism* for defining easily the XPath framework, and also in the trace of wrong answers of Section 4, where only those parts of the computation that produce a particular fragment of the answer are required.
- *Higher-order patterns*. This feature allows us to consider $\mathcal{TOY}$ expressions that are not yet reducible as patterns. This means in our case that XPath expressions can be considered at the same time executable code (when applied to an input XML document), or data structures when considered as higher-order patterns. This powerful characteristic of the language is heavily used in our proposals of sections 4 and 5.
- *Logic variables*, specially when used in *generate and test* expressions are very suitable for obtaining the values of intermediate computations, and in our case

also for guessing values in the debugger of missing answers.

Summarizing, we consider that the declarative nature of XPath matches very well the characteristics of functional-logic languages. This benefits both paradigms: the functional-logic language can include easily XPath queries without using any external library, and XPath practitioners can implement easily functions that manipulate the queries in order to devise prototypes of XPath tools such as optimizers or debuggers.

A $\mathcal{TOY}$ implementation that includes the XPath primitives for loading and saving XML documents and most of the examples of this paper can be downloaded from: `http://gpd.sip.ucm.es/rafa/xpath/toyxpath.rar`. As future work, we plan to consider exploiting the same $\mathcal{TOY}$ characteristics for optimizing XQuery expressions.

References

[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] F. Atanassow, D. Clarke, and J. Jeuring. UUXML: A Type-Preserving XML Schema Haskell Data Binding. In *Proc. of Practical Aspects of Declarative Languages*, pages 71–85, Heidelberg, Germany, 2004. Springer LNCS 3057.

[3] F. Bry and S. Schaffert. The XML Query Language Xcerpt: Design Principles, Examples, and Semantics. In *Proc. of Web, Web-Services, and Database Systems*, pages 295–310, Heidelberg, Germany, 2002. Springer LNCS 2593.

[4] R. Caballero, Y. García-Ruiz, and F. Sáenz-Pérez. Integrating XPath with the Functional-Logic Language $\mathcal{TOY}$ (Extended Version). Technical Report SIP-05/10, Facultad de Informática, Universidad Complutense de Madrid, 2010. `http://federwin.sip.ucm.es/sic/investigacion/publicaciones/pdfs/SIC-5-10.pdf`.

[5] R. Caballero, Y. García-Ruiz, and F. Sáenz-Pérez. Integrating XPath with the Functional-Logic Language $\mathcal{TOY}$. In *Proceedings of the 13th international conference on Practical aspects of declarative languages*, PADL'11, pages 145–159, Berlin, Heidelberg, 2011. Springer-Verlag.

[6] B. Cautis, A. Deutsch, and N. Onose. XPath rewriting using multiple views: Achieving completeness and efficiency. In *Proceedings of the International Workshop on Web and Databases*, 2008.

[7] J. Coelho and M. Florido. XCentric: logic programming for XML processing. In *WIDM '07: Proceedings of the 9th annual ACM international workshop on Web information and data management*, pages 1–8, NY, USA, 2007. ACM Press.

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

[9] 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.

[10] M. Hanus. Declarative processing of semistructured web data. Technical report 1103, Christian-Albrechts-Universität Kiel, 2011.

[11] F. J. López-Fraguas and J. S. Hernández. $\mathcal{TOY}$: 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.

[12] W. May. XPath-Logic and XPathLog: A Logic-Programming Style XML Data Manipulation Language. *Theory and Practice of Logic Programming*, 4(3):239–287, 2004.

[13] R. Ronen and O. Shmueli. Evaluation of datalog extended with an XPath predicate. In *Proceedings of the 9th annual ACM international workshop on Web information and data management*, pages 9–16. ACM New York, NY, USA, 2007.

[14] S. Schaffert and F. Bry. A Gentle Introduction to Xcerpt, a Rule-based Query and Transformation Language for XML. In *Proc. of International Workshop on Rule Markup Languages for Business Rules on the Semantic Web*, page 22 pages, Aachen, Germany, 2002. CEUR Workshop Proceedings 60.

[15] D. Seipel, J. Baumeister, and M. Hopfner. Declaratively Querying and Visualizing Knowledge Bases in XML. In *In Proc. Int. Conf. on Applications of Declarative Programming and Knowledge Management*, pages 140–151. Springer, 2004.

[16] M. Sulzmann and K. Z. Lu. Xhaskell — adding regular expression types to haskell. In *Implementation and Application of Functional Languages: 19th International Workshop, IFL 2007, Freiburg, Germany, September 27-29, 2007. Revised Selected Papers*, pages 75–92, Berlin, Heidelberg, 2008. Springer-Verlag.

[17] P. Thiemann. A typed representation for HTML and XML documents in Haskell. *Journal of Functional Programming*, 12(4&5):435–468, 2002.

[18] W3C. Extensible Markup Language (XML), 2007.

[19] W3C. XML Path Language (XPath) 2.0, 2007.

[20] W3C. XML Query Use Cases, 2007. http://www.w3.org/TR/xquery-use-cases/.

[21] W3C. XQuery 1.0: An XML Query Language, 2007.

[22] M. Wallace and C. Runciman. Haskell and XML: generic combinators or type-based translation? *SIGPLAN Not.*, 34(9):148–159, 1999.

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

[24] W. Xu and Z. Özsoyoglu. Rewriting XPath queries using materialized views. In *Proceedings of the 31st international conference on Very large data bases*, pages 121–132. VLDB Endowment, 2005.