XQuery in the Functional-Logic Language Toy

Jesus M. Almendros-Jiménez[1], Rafael Caballero[2], Yolanda García-Ruiz[2], and Fernando Sáenz-Pérez[3,⋆]

[1] Dpto. de Lenguajes y Computación, Universidad de Almería
[2] Departamento de Sistemas Informáticos y Computación
Universidad Complutense de Madrid
[3] Departamento de Ingeniería del Software e Inteligencia Artificial
Universidad Complutense de Madrid
Spain

Abstract. This paper presents an encoding of the XML query language XQuery in the functional-logic language $\mathcal{TOY}$. The encoding is based on the definition of for-let-where-return constructors by means of $\mathcal{TOY}$ functions, and uses the recently proposed XPath implementation for this language as a basis. XQuery expressions can be executed in $\mathcal{TOY}$ obtaining sequences of XML elements as answers. Our setting exploits the non-deterministic nature of $\mathcal{TOY}$ by retrieving the elements of the XML tree once at a time when necessary. We show that one of the advantages of using a rewriting-based language for implementing XQuery is that it can be used for optimizing XQuery expressions by query rewriting. With this aim, XQuery expressions are converted into higher order patterns that can be analyzed and modified by $\mathcal{TOY}$ functions.

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

1 Introduction

In the last few years the eXtensible Markup Language XML [33] has become a standard for the exchange of semistructured data. Thus, querying XML documents from different languages has become a convenient feature. XQuery [35,37] has been defined as a query language for finding and extracting information from XML documents. It extends XPath [34], a domain-specific language that has become part of general-purpose languages. Recently, in [10], we have proposed an implementation of XPath in the functional-logic language $\mathcal{TOY}$ [22]. The implementation is based on the definition of XPath constructors by means of $\mathcal{TOY}$ functions. As well, XML documents are represented in $\mathcal{TOY}$ by means of terms, and the basic constructors of XPath: `child`, `self`, `descendant`, etc.

⋆ This work has been supported by the Spanish projects TIN2008-06622-C03-01, TIN2008-06622-C03-03, S-0505/TIC/0407, S2009TIC-1465, and UCM-BSCH-GR58/08-910502.

H. Kuchen (Ed.): WFLP 2011, LNCS 6816, pp. 35–51, 2011.

are defined as functions that apply to XML terms. The goal of this paper is to extend [10] to XQuery.

The existing XQuery implementations either use functional programming or Relational Database Management Systems (RDBMS's). In the first case, the *Galax* implementation [23] encodes XQuery into *Objective Caml*, in particular, encodes XPath. Since XQuery is a functional language (with some extensions) the main encoding is related with the type system for allowing XML documents and XPath expressions to occur in a functional expression. With this aim, a specific type system for handling XML tags, the hierarchical structure of XML, and sequences of XML items is required. In addition, XPath expressions can be implemented from this representation. There are also proposals for new languages based on functional programming rather than implementing XPath and XQuery. This is the case of *XDuce* [19] and *CDuce* [5,6], which are languages for XML data processing, using regular expression pattern matching over XML trees and subtyping as basic mechanism. There are also proposals around *Haskell* for handling XML documents, such as *HaXML* and *UUXML* [31,4,36,30]. XML types are encoded with Haskell's type classes providing a Haskell library in which XML types are encoded as algebraic datatypes. HXQ [14] is a translator from XQuery to embedded Haskell code, using the Haskell templates. HXQ stores XML documents in a relational database, and translates queries into SQL queries.

This is also followed in some RDBMS XQuery implementations: XML documents are encoded with relational tables, and XPath and XQuery with SQL. The most relevant contribution in this research line is *MonetDB/XQuery* [7]. It consists of the *Pathfinder* XQuery compiler [8] on top of the *MonetDB* RDBMS, although *Pathfinder* can be deployed on top of any RDBMS. MonetDB/XQuery encodes the XML tree structure in a relational table following a pre/post order traversal of the tree (with some variant). XPath can be implemented from such table-based representation, and XQuery by encoding *flwor* expressions into the *relational algebra*, extended with the so-called *loop-lifted staircase join*.

There are also proposals based on logic programming. In most cases, new languages for XML processing are proposed. The *Xcerpt* project [27,9] proposes a pattern and rule-based query language for XML documents, using the so-called *query terms* including logic variables for the retrieval of XML elements. Another contribution to XML processing is the language *XPathLog* (integrated in the the *Lopix* system) [24] which is a *Datalog*-style extension for *XPath* with variable bindings. *XCentric* [13] is an approach for representing and handling XML documents by logic programs, by considering terms with functions of flexible arity and regular types. $XPath^L$ [26] is a logic language based on rules for XML processing including a specific predicate for handling *XPath* expressions in Datalog programs. *FNPath* [29] is also a proposal for using *Prolog* as a query language for XML documents. It maps XML documents to a Prolog Document Object Model (DOM), which can either consist of facts (graph notation) or a term structure (field notation). *FNPath* can evaluate XPath expressions based on that DOM. [2,3] aim to implement XQuery by means of logic programming, providing two

alternatives: a top-down and a bottom-up approaches (the latter in the line of Datalog programs). Finally, some well-known *Prolog* implementations include libraries for loading XML documents, such as *SWI-Prolog* [38] and *Ciao* [12].

In the field of functional-logic languages, [18] proposes a rule-based language for processing semistructured data that is implemented and embedded in the functional logic language Curry [17]. The framework is based on providing operations to describe partial matchings in the data and exploits functional patterns and set functions for the programming tasks.

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). This is the approach we have followed in our previous work [10]. In the case of XQuery, for-let-where-return constructors can be encoded in $\mathcal{TOY}$, which uses the XPath query language as a basis. XQuery expressions can be encoded by means of (first-order) functions. However, we show that we can also consider XQuery expressions as higher order patterns, in order to manipulate XQuery programs by means of $\mathcal{TOY}$. For instance, we have studied how to transform XQuery expressions into $\mathcal{TOY}$ patterns in order to optimize them. 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 [11,20]. A completely declarative proposal for integrating part of XQuery in $\mathcal{TOY}$ can be found in [1], which restricts itself to the completely declarative features of the language. This implies that the subset of XQuery considered is much narrower than the framework presented here. The advantage of restricting to the purely declarative view is that proofs of correctness and completeness are provided. In this work we take a different point of view, trying to define a more general XQuery framework although using non-purely declarative features as the (meta-)primitive `collect`. Another difference of this work is the use of higher-order patterns for rewriting queries, which was not available in [1].

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

- *Non-deterministic functions* are used to nicely represent the evaluation of an XPath/XQuery query, which consists of fragments of the input XML document. In addition, the `for` constructor of XQuery can be defined with non-deterministic behavior.
- *Logic variables* are employed for instance when obtaining the contents of XPath text nodes, and for solving nested XQuery expressions, capturing the non-deterministic behavior of inner `for` and XPath expressions.
- By defining rules with *higher-order patterns*, XPath/XQuery queries become truly first-class citizens in our setting. In the case of XQuery, this allows us to rewrite queries in order to be optimized. XPath can also be optimized (see [10] for more details).

The rest of the paper is organized as follows. Section 2 briefly introduces the XPath subset presented in [10]. Section 3 defines the encoding of XQuery in

$\mathcal{TOY}$. Section 4 shows how to use $\mathcal{TOY}$ for the optimization of XQuery. Finally, Section 5 presents some conclusions.

2 XPath in $\mathcal{TOY}$

This section introduces the functional-logic language $\mathcal{TOY}$ [22] and the subset of XPath that we intend to integrate with $\mathcal{TOY}$, omitting all the feaures of XPath that are supported by $\mathcal{TOY}$ but not used in this paper, such as filters, abbreviations, attributes and preprocessing of reverse axes. See [10] for a more detailed introduction to XPath in $\mathcal{TOY}$.

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

All the examples in this paper are written in the concrete syntax of the lazy functional-logic language $\mathcal{TOY}$ [22], but most of the code can be easily adapted to other similar languages as Curry [17]. $\mathcal{TOY}$ is a lazy functional-logic language. A $\mathcal{TOY}$ program is composed of data type declarations, type alias, infix operators, function type declarations and defining rules for functions symbols. The syntax is similar to the functional language Haskell, except for the capitalization, which follows the approach of Prolog (variables start by uppercase, and other symbols by lowercase[1]). Each rule for a function f has the form:

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

where 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 $(f\ e_1 \ldots e_n)$ returns 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 $i = 1 \ldots m$.

Infix operators are also allowed as particular case of program functions. Consider for instance the definitions:

```
infixr 30 /\                infixr 30 \/            infixr 45 ?
false /\ X = false          true \/ X = true        X ? _Y = X
true /\ X = X               false \/ X = X          _X ? Y = Y
```

The /\ and \/ operators represent the standard conjunction and disjunction, respectively, while `?` represents the non-deterministic choice. For instance the infix declaration `infixr 45 ?` indicates that `?` is an infix operator that associates to the right (the r in `infixr`) and that its priority is 35. The priority is used to assume precedences in the case of expressions involving different operators. Computations in $\mathcal{TOY}$ start when the user inputs some goal as

[1] Also, only variables are allowed to start that way. If another identifier has to start with uppercase or underscore, it must be delimited between single quotes.

```
Toy> 1 ? 2 ? 3 ? 4 == R
```

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 ↦ 1 }`, `{R ↦ 2 }`, `{R ↦ 3 }`, and `{R ↦ 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 is: `member :: [A] -> A` which 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 and higher order parameters like `apply F X = F X`. 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 `nth 1 == R1, R1 ["hello","friends"] == R2` and produces the answer `{ R1 ↦ (nth 1), R2 ↦ "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, a program rule like `foo (apply member) = true` is valid, although `foo (apply member []) = true` is not because `apply member []` is a reducible expression and not a valid pattern. 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.

2.2 Representing XPath Queries

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

```
data xmlNode      = txt     string
                  | comment string
                  | xmlTag  string [xmlAttribute] [xmlNode]
data xmlAttribute = att     string string
```

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

Data type `xmlNode` represents nodes in a simple XML document. It distinguishes three types of nodes: texts, comments, and tags (element nodes), 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. Data type `xmlAttribute` contains the name of the attribute and its value (both of type `string`). Type alias `xml` is a renaming of the data type `xmlNode`. Finally, type alias `xPath` is defined as a function from nodes to nodes, and is the type of XPath constructors. Of course, this list is not exhaustive, since it misses several types of XML nodes, but it is enough for this presentation. Notice that in $\mathcal{TOY}$ we do not still consider the adequacy of the document to its underlying *Schema* definition [32]. This task has been addressed in functional programming defining regular expression types [30]. However, we assume well-formed input XML documents. In order to import XML documents, the $\mathcal{TOY}$ primitive `load_xml_file` loads an XML file returning its representation as a value of type `xmlNode`. Figure 1 shows an example of XML file and its representation in $\mathcal{TOY}$.

Typically, XPath expressions return several fragments of the XML document. Thus, the expected type in $\mathcal{TOY}$ for `xPath` could be `type xPath = xml -> [xml]` meaning that a list or sequence of results is obtained. This is the approach considered in [2] and also the usual in functional programming [16]. However,

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

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

in our case we take advantage of the non-deterministic nature of our language, returning each result individually. 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`. In order to apply an XPath expression to a particular document, we use the following infix operator definition:

```
(<--) :: 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 an 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)
```

We use the function operator names `.::.` and `./.` because `::` and `/` are already defined in $\mathcal{TOY}$. Also notice that their definitions are the same. Indeed, we could 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 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. 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 `xmlTag`. 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,

```
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 (xmlTag S Att L) =
         xmlTag 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 `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()
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 the $\mathcal{TOY}$ goal:

```
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::price/child::item` 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 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()
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 useful abbreviation is `//` which stands for the unabbreviated expression `/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`.

3 XQuery in $\mathcal{TOY}$

Now, we are in a position to define the proposed extension to XQuery. Firstly, the subset of XQuery expressions handled in our setting is presented (XQuery is a richer language than the fragment presented here):

XQuery ::= XPath | $Var | XQuery/XPath* |
let $Var **:=** XQuery [**where** BXQuery] **return** XQuery |
for $Var **in** XQuery [**where** BXQuery] **return** XQuery |
<tag> XQuery < /tag>
BXQuery ::= XQuery | XQuery=XQuery

Basically, the XQuery fragment handled in $\mathcal{TOY}$ allows building new XML documents employing new tags, and the traversal of XML documents by means of the `for` construction. XQuery variables are used in `for` and `let` expressions and can occur in the built documents and XPath expressions. It is worth observing that XPath can be applied to XQuery expressions, that is, for instance, XPath can be applied to the result of a `for` expression. Therefore, such XPath expressions are not rooted by documents (they are denoted by XPath*). In order to encode XQuery in $\mathcal{TOY}$ we define a new type:

```
type xQuery = [xml]
```

In Section 2, XPath has been represented as functions from xml nodes to xml nodes. However, XQuery expressions are defined as sequences of xml nodes represented in $\mathcal{TOY}$ by lists. This does not imply that our approach returns the answers enclosed in lists, it still uses non-determinism for evaluating `for` and XPath expressions. We define functions for representing `for-let-where-return` expressions as follows. Firstly, `let` and `for` expressions are defined as:

```
xLet :: xQuery -> xQuery -> xQuery
xLet X [Y]       = if X == collect Y then X
xLet X (X1:X2:L) = if X == (X1:X2:L) then X

xFor :: xQuery -> xQuery -> xQuery
xFor X [Y]       = if X == [Y] then X
xFor X (X1:X2:L) = if X == [member (X1:X2:L)] then X
```

`xLet` uses `collect` for capturing the elements of `Y` in a list, whereas `xFor` retrieves non deterministically the elements of `Expr` in unitary lists. It fits well, for instance, when `Y` is an XPath expression in $\mathcal{TOY}$. The definition of `for` relies on the non-deterministic function `member` defined in Section 2. Now $\mathcal{TOY}$ goals like `xFor X ("food.xml" <$- name "food" ./. name "item")==R` or `xLet X`

`("food.xml" <$- name "food" ./. name "item")==R` can be tried. Let us remark that XPath expressions have been modified in XQuery as follows. A new operator `<$-` is defined in terms of `<-`:

```
infixr 35 <$--
(<$--) :: string -> xPath -> xQuery
(<$--) Doc Path = [(<--) Doc Path]
```

The function `<$-` returns (non deterministically) unitary lists with the elements of the given document in the corresponding path. Therefore, XPath and `for` expressions have the same behavior in the $\mathcal{TOY}$ implementation of XQuery. In other words, `(<$-)` serves for type conversion from XPath to XQuery. Now, we can define `where` and `return` as follows:

```
infixr 35 `xWhere`
(`xWhere`) :: xQuery -> bool -> xQuery
(`xWhere`) X Y = if Y then X

infixr 35 `xReturn`
(`xReturn`) :: xQuery -> xQuery -> xQuery
(`xReturn`) X Y = if X == _ then Y
```

The definition of `xWhere` is straightforward: the query `X` is returned if the condition `Y` can be satisfied. The `if` statement in `xReturn` forces the evaluation of `X`. The anonymous variable (`_`) can be read as *if the query* `X` *does not fail, then return* `Y`. With these definitions, we can simulate many XQuery expressions in $\mathcal{TOY}$. However, there are two elements still to be added. XPath expressions can now be rooted by XQuery expressions. Thus, we add a new function:

```
infixr 35 <$
(<$) :: xQuery -> xPath -> xQuery
(<$) [Y] Path = [Path Y]
(<$) (X:Y:L) Path = map Path (X:Y:L)
```

The first argument is an XPath variable or, more generally, an XQuery expression. The XPath expression represented by variable `Path` is applied to all the values produced by the XQuery expression. According to the commented behavior, XQuery expressions can be unitary lists (`for`'s and XPath's) and non-unitary lists (`let`'s). The `xmlTag` constructor is also converted into a function `xmlTagX`:

```
xmlTagX :: string -> [xmlAttribute] -> xQuery -> xQuery
xmlTagX Name Attributes [Expr] =
     if Y == collect Expr then [xmlTag Name Attributes Y]
xmlTagX Name Attributes (X:Y:L) = [xmlTag Name Attributes (X:Y:L)]
```

Basically, this conversion is required to apply `collect` when either a `for` or an XPath expression provides the elements enclosed in an XML tag. With the previous definitions, $\mathcal{TOY}$ accepts the following query:

```
R == xmlTagX "names" []
         (xLet X ("food.xml" <$-- name "food")
         `xReturn`
         xmlTagX "result" [] (X <$ (name "item"./.name "name")))
```

which simulates the query:

```
<names>
let $x:=doc("food.xml")/food return
<result> { $x/item/name } </result>
</names>
```

and outcomes the following answer:

```
{R -> [xmlTag "names" []
         [xmlTag "result" [] [
          xmlTag "name" [] [xmlText "watermelon" ],
          xmlTag "name" [] [xmlText "oranges" ],
          xmlTag "name" [] [xmlText "onions" ],
          xmlTag "name" [] [xmlText "strawberries" ]]]] }
```

It is worth noticing that $\mathcal{TOY}$ shows not only the binding for `R`, but also for the variable `X`. If we are interested in the query without the values of the variables, we can introduce a function containing the code:

```
query = xmlTagX "names" []
         (xLet X ("food.xml" <$-- name "food")
         `xReturn`
         xmlTagX "result" [] (X <$ (name "item"./.name "name")))
```

and try the goal `query == R` to get the same result. In the case of `for` expressions, we can write:

```
query2 = xFor Y
        (xFor X ("food.xml" <$-- name "food")
              `xReturn` (X <$ (name "item" ./. name "name")))
        `xReturn` Y
```

which simulates the following query:

```
for $Y in
(for $X in doc("food.xml")/food return $X/item/name)
return $Y
```

The following $\mathcal{TOY}$ query returns four answers, once at a time, due to the use of non-determinism in the `for` expression:

```
Toy> query2== X
      { X -> [xmlTag "name" [] [xmlText "watermelon"]] }
      { X -> [xmlTag "name" [] [xmlText "oranges"]] }
      { X -> [xmlTag "name" [] [xmlText "onions"]] }
      { X -> [xmlTag "name" [] [xmlText "strawberries"]] }
```

4 XQuery Optimization in $\mathcal{TOY}$

In this section we present one of the advantages of using $\mathcal{TOY}$ for running XQuery expressions. In [10] we have shown that XPath queries can be preprocessed by replacing the reverse axes by predicate filters including forward axes, as shown in [25]. In the case of XQuery, one of the optimizations to be achieved is to avoid XPath expressions at outermost positions. Here is an example of optimization. Consider the following query:

```
exam = xFor X
            (xFor Y ("food.xml" <$-- name "food" ./. name "item")
            `xReturn`
            (xmlTagX "elem" []
                (xFor Z (Y <$ name "name")
                    `xReturn` (xmlTagX "ids" [] Z))))
        `xReturn`
            (X <$ ((name "ids") ./. (name "name")))
```

In such a query, `(X <$ ((name "ids") ./. (name "name"))` is an XPath expression applied to an XML term constructed by the same query. By removing outermost XPath expressions, we can optimize XQuery expressions. In general, a place for optimization are nested XQuery expressions [21,15]. In our case, we argue that XPath can be statically applied to XQuery expressions. The optimization comes from the fact that unnecessary XML terms can be built at run-time, and that removing them improves memory consumption. We observe in the previous query that `"elem"` and `"ids"` tags are useless, once we retrieve `"name"` from the original file. Therefore, the previous query can be rewritten into a more simpler and equivalent one:

```
examo = ("food.xml" <$-- name "food"./.name "item"./.name "name")
```

4.1 XQuery as Higher Order Patterns

In order to proceed with optimizations, we follow the same approach as in XPath. In [10] we have used the representation of XPath expressions for optimizing. As it was commented before, XPath operators are higher order operators, and then we can take advantage of the $\mathcal{TOY}$ facilities for using higher order patterns to rewrite them. This is not the case, however, for XQuery expressions in $\mathcal{TOY}$ because, for instance, `xFor` and `xLet` are always applied to two arguments, and therefore constitute reducible expressions, not higher-order patterns. In order to convert XQuery-$\mathcal{TOY}$ expressions into higher-order patterns, we propose a redefinition of the functions adding a dummy argument. Then, XQuery constructors can be redefined as follows:

```
yLet :: (A -> xQuery) -> (A -> xQuery) -> A -> xQuery
yLet X Y _ = xLet (X _)  (Y _)
```

```
yFor :: (A -> xQuery) -> (A -> xQuery) -> A -> xQuery
yFor X Y _ = xFor (X _)  (Y _)
```

The anonymous variable plays a role similar to the `quote` operator in Lisp [28]. In our case the expressions will become reducible when any extra argument is provided. In the meanwhile it can be considered as a data term, and as such it can be analyzed and modified. In the definitions above, `yLet` is reduced to `xLet` when such extra argument is provided. The two arguments `X` and `Y` also need their extra variable to become reducible. A variable is a special case, which has to be converted into a function (`xvar`):

```
xvar :: xQuery -> A -> xQuery
xvar X _ = X
```

Now, a given query can be rewritten as a higher order pattern. For instance, the previous `exam` can be represented as follows:

```
xexam = yFor (xvar X)
       (yFor (xvar Y) ("food.xml" <$$-- name "food"./.name "item")
              `yReturn`
                 (xmlTagY "elem" []
                    (yFor (xvar Z) ((xvar Y) <$$ (name "name"))
                       `yReturn` (xmlTagY "ids" [] (xvar Z)) )))
        `yReturn`
        ((xvar X) <$$ ((name "ids") ./. name "name"))
```

The query can be executed in $\mathcal{TOY}$ just providing any additional argument, in this case an anonymous variable:

```
Toy> xexam _ == R
      { R -> [xmlTag "name" [] [xmlText "watermelon" ] ] }
      { R -> [xmlTag "name" [] [xmlText "oranges" ] ] }
      { R -> [xmlTag "name" [] [xmlText "onions" ] ] }
      { R -> [xmlTag "name" [] [xmlText "strawberries" ] ] }
```

If the extra argument `_` is omitted, then the variable `R` is bound to the XQuery code `yFor (xvar X) (...name "name"))`. This behavior allows us to inspect and modify the query in the next subsection.

4.2 XQuery Transformations

Now, we would like to show how to rewrite XQuery expressions in order to optimize them. We have defined a set of transformation rules for removing outermost XPath expressions, when possible. Let us remark that correctness of the transformation rules, that is, preserving equivalence, is out of the scope of this paper. An example of (a subset of) the transformation rules is:

```
reduce ((yFor (xvar Z) E) `yReturn` (xvar Z)) = E
reduce ((xmlTagY N A E) <$$ P)   = reduce_xml (xmlTagY N A E) P
reduce_xml (xmlTagY N A E) P     = reduce_xmlPath E P
reduce_xmlPath (xmlTagY N A E) P =
                        if P == (name N) ./. P2
                        then reduce_xmlPath E P2
                        else ((xmlTagY N A E) <$$ P)
...
```

The first reduction rule removes the unnecessary `for` expressions that define a variable `Z` taking a value `E` only to return `Z`. The second rule removes XPath expressions that traverse elements built in the same query. For instance, an expression of the form `$X/a/b`, with `$X` of the form `<a>E</a>` is reduced to `$Y/b` with `$Y/b` and `$X` taking the value `E` (this transformation is performed by function `reduce_xmlPath`). The optimizer can be defined as the fixpoint of function `reduce`:

```
optimize :: (A -> xQuery) -> (A -> xQuery)
optimize X = iterate reduce X

iterate :: (A -> A) -> A -> A
iterate G X = if Y == X then Y else iterate G Y
              where Y = (G X)
```

For instance, the running example is optimized as follows:

```
Toy> optimize xexam == X
      { X -> (<$$-- "food.xml" child .:::. (nameT "food") ./.
       child .:::. (nameT "item") ./. child .:::. (nameT "name")) }
```

Finally, an XQuery expression is executed (with optimizations) in $\mathcal{TOY}$ by calling the function `run`, which is defined as:

```
run :: (A -> xQuery) -> xQuery
run X = (optimize X) _
```

By using `run`, $\mathcal{TOY}$ obtains the same four answers as with the original query:

```
Toy> run xexam == X
      { R -> [xmlTag "name" [] [xmlText "watermelon" ] ] }
      { R -> [xmlTag "name" [] [xmlText "oranges" ] ] }
      { R -> [xmlTag "name" [] [xmlText "onions" ] ] }
      { R -> [xmlTag "name" [] [xmlText "strawberries" ] ] }
```

In order to analyze the performance of the optimization, the next table compares the elapsed time for the query running on $\mathcal{TOY}$ before and after the optimization, with respect to different sizes for file `"food.xml"`.

Items	*Initial Query*	*Optimized Query*	*Speed-up*
1,000	1.9	0.4	4.8
2,000	3.7	0.8	9.3
4,000	7.4	1.7	4.4
8,000	18.1	3.9	4.6
16,000	36.0	7.8	4.6

The first column indicates the number of `item` elements included in `"food.xml"`, the second and third column display the time in seconds required by the original and the optimized query, respectively, and the last column displays the speed-up of the optimized code. In order to force the queries to find all the answers, the submitted goals are `(exam == R, false)` and `(run xexam == R, false)`, corresponding to the initial and the optimized query, respectively. The atom `false` after the first atomic subgoal always fails, forcing the reevaluation until no more solutions exist. As can be seen in the table, in this experiment the optimized query is above 4.5 times faster in the average than the initial one. In other experiments (for instance, replacing `for` by `let` in this example) the difference can be noticeable also in terms of memory, since the system runs out of memory computing the query before optimization, but works fine with the optimized query. Of course, more extensive benchmarks would be needed to assess this preliminary results. However, the purpose of this paper is not to propose or to evaluate XQuery optimizations, but to show how they can be easily incorporated and tested in our framework.

5 Conclusions

We have shown how the declarative nature of the XML query language XQuery fits in a very natural way in functional-logic languages. Our setting fruitfully combines the collection of results required by XQuery `let` statements and the use of individual values as required by `for` statements and XPath expressions. For the users of the functional-logic $\mathcal{TOY}$, the advantage is clear: they can use queries very similar to XQuery in their programs. Although adapting to the $\mathcal{TOY}$ syntax can be hard at first, we think that the queries are close enough to their equivalents in native XQuery. However, we would like to go further by providing a parser from XQuery standard syntax to the equivalent $\mathcal{TOY}$ expressions.

From the point of view of the XQuery apprentices, the tool can be useful, specially if they have some previous knowledge of declarative languages. The possibility of testing query optimizations can be very helpful. The paper shows a technique based on the use of additional dummy variables for converting queries in higher-order patters. A similar idea would be to use a data type for representing the query and then a parser/interpreter for evaluating this data type. However, we think that the approach considered here has a higher abstraction level, since the queries can not only be analyzed, they can also be computed by simply providing an additional argument. Finally, the framework can also

be interesting for designers of XQuery environments, because it allows users to easily define prototypes of new features such as new combinators and functions.

A version of the $\mathcal{TOY}$ system including the examples of this paper can be downloaded from `http://gpd.sip.ucm.es/rafa/wflp2011/toyxquery.rar`

References

1. Almendros-Jiménez, J., Caballero, R., García-Ruiz, Y., Sáenz-Pérez, F.: A Declarative Embedding of XQuery in a Functional-Logic Language. Technical Report SIC-04/11, Facultad de Informática, Universidad Complutense de Madrid (2011), `http://gpd.sip.ucm.es/rafa/xquery/`
2. Almendros-Jiménez, J.M.: An Encoding of XQuery in Prolog. In: Bellahsène, Z., Hunt, E., Rys, M., Unland, R. (eds.) XSym 2009. LNCS, vol. 5679, pp. 145–155. Springer, Heidelberg (2009)
3. Almendros-Jiménez, J.M., Becerra-Terón, A., Enciso-Baños, F.J.: Querying XML documents in logic programming. Journal of Theory and Practice of Logic Programming 8(3), 323–361 (2008)
4. Atanassow, F., Clarke, D., Jeuring, J.: UUXML: A type-preserving XML schema–haskell data binding. In: Jayaraman, B. (ed.) PADL 2004. LNCS, vol. 3057, pp. 71–85. Springer, Heidelberg (2004)
5. Benzaken, V., Castagna, G., Frish, A.: CDuce: an XML-centric general-purpose language. In: Proc. of the ACM SIGPLAN International Conference on Functional Programming, pp. 51–63. ACM Press, New York (2005)
6. Benzaken, V., Castagna, G., Miachon, C.: A Full Pattern-Based Paradigm for XML Query Processing. In: Hermenegildo, M.V., Cabeza, D. (eds.) PADL 2004. LNCS, vol. 3350, pp. 235–252. Springer, Heidelberg (2005)
7. Boncz, P., Grust, T., van Keulen, M., Manegold, S., Rittinger, J., Teubner, J.: MonetDB/XQuery: a fast XQuery processor powered by a relational engine. In: Proceedings of the 2006 ACM SIGMOD International Conference on Management of Data, pp. 479–490. ACM Press, New York (2006)
8. Boncz, P.A., Grust, T., van Keulen, M., Manegold, S., Rittinger, J., Teubner, J.: Pathfinder: XQuery - The Relational Way. In: Proc. of the International Conference on Very Large Databases, pp. 1322–1325. ACM Press, New York (2005)
9. Bry, F., Schaffert, S.: The XML Query Language Xcerpt: Design Principles, Examples, and Semantics. In: Chaudhri, A.B., Jeckle, M., Rahm, E., Unland, R. (eds.) NODe-WS 2002. LNCS, vol. 2593, pp. 295–310. Springer, Heidelberg (2003)
10. Caballero, R., García-Ruiz, Y., Sáenz-Pérez, F.: Integrating XPath with the Functional-Logic Language Toy. In: Rocha, R., Launchbury, J. (eds.) PADL 2011. LNCS, vol. 6539, pp. 145–159. Springer, Heidelberg (2011)
11. Caballero, R., López-Fraguas, F.: A functional-logic perspective on parsing. In: Middeldorp, A. (ed.) FLOPS 1999. LNCS, vol. 1722, pp. 85–99. Springer, Heidelberg (1999)
12. Cabeza, D., Hermenegildo, M.: Distributed WWW Programming using (Ciao-)Prolog and the PiLLoW Library. Theory and Practice of Logic Programming 1(3), 251–282 (2001)
13. Coelho, J., Florido, M.: XCentric: logic programming for XML processing. In: WIDM 2007: Proceedings of the 9th Annual ACM International Workshop on Web Information and Data Management, pp. 1–8. ACM Press, New York (2007)
14. Fegaras, L.: HXQ: A Compiler from XQuery to Haskell (2010)

15. Grinev, M., Pleshachkov, P.: Rewriting-based optimization for XQuery transformational queries. In: 9th International Database Engineering and Application Symposium, IDEAS 2005, pp. 163–174. IEEE Computer Society, Los Alamitos (2005)
16. Guerra, R., Jeuring, J., Swierstra, S.D.: Generic validation in an XPath-Haskell data binding. In: Proceedings Plan-X (2005)
17. Hanus, M.: Curry: An Integrated Functional Logic Language (2003), `http://www.informatik.uni-kiel.de/~mh/curry/` (version 0.8.2 March 28, 2006)
18. Hanus, M.: Declarative processing of semistructured web data. Technical report 1103, Christian-Albrechts-Universität Kiel (2011)
19. Hosoya, H., Pierce, B.C.: XDuce: A Statically Typed XML Processing Language. ACM Transactions on Internet Technology 3(2), 117–148 (2003)
20. Hutton, G., Meijer, E.: Monadic parsing in Haskell. J. Funct. Program. 8(4), 437–444 (1998)
21. Koch, C.: On the role of composition in XQuery. In: Proc. WebDB (2005)
22. Fraguas, F.J.L., Hernández, J.S.: TOY: A Multiparadigm Declarative System. In: Narendran, P., Rusinowitch, M. (eds.) RTA 1999. LNCS, vol. 1631, pp. 244–247. Springer, Heidelberg (1999)
23. Marian, A., Simeon, J.: Projecting XML Documents. In: Proc. of International Conference on Very Large Databases, pp. 213–224. Morgan Kaufmann, Burlington (2003)
24. May, W.: XPath-Logic and XPathLog: A Logic-Programming Style XML Data Manipulation Language. Theory and Practice of Logic Programming 4(3), 239–287 (2004)
25. Olteanu, D., Meuss, H., Furche, T., Bry, F.: XPath: Looking forward. In: Chaudhri, A.B., Unland, R., Djeraba, C., Lindner, W. (eds.) EDBT 2002. LNCS, vol. 2490, pp. 109–127. Springer, Heidelberg (2002)
26. Ronen, R., Shmueli, O.: Evaluation of datalog extended with an XPath predicate. In: Proceedings of the 9th Annual ACM International Workshop on Web Information and Data Management, pp. 9–16. ACM, New York (2007)
27. Schaffert, S., Bry, F.: 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. CEUR Workshop Proceedings, vol. 60, p. 22 (2002)
28. Seibel, P.: Practical Common Lisp. Apress (2004)
29. Seipel, D.: Processing XML-Documents in Prolog. In: Procs. of the Workshop on Logic Programming 2002, p. 15. Technische Universität Dresden, Dresden (2002)
30. Sulzmann, M., Lu, K.Z.: XHaskell – adding regular expression types to haskell. In: Chitil, O., Horváth, Z., Zsók, V. (eds.) IFL 2007. LNCS, vol. 5083, pp. 75–92. Springer, Heidelberg (2008)
31. Thiemann, P.: A typed representation for HTML and XML documents in Haskell. Journal of Functional Programming 12(4&5), 435–468 (2002)
32. W3C. XML Schema 1.1
33. W3C. Extensible Markup Language, XML (2007)
34. W3C. XML Path Language (XPath) 2.0 (2007)
35. W3C. XQuery 1.0: An XML Query Language (2007)
36. Wallace, M., Runciman, C.: Haskell and XML: Generic combinators or type-based translation? In: Proceedings of the International Conference on Functional Programming, pp. 148–159. ACM Press, New York (1999)
37. Walmsley, P.: XQuery. O'Reilly Media, Inc., Sebastopol (2007)
38. Wielemaker, J.: SWI-Prolog SGML/XML Parser, Version 2.0.5. Technical report, Human Computer-Studies (HCS), University of Amsterdam (March 2005)