

Using Reflection for Querying XML Documents*

Markus Kirchberg, Faizal Riaz-ud-Din, Klaus-Dieter Schewe, Alexei Tretiakov

Massey University, Department of Information Systems &
Information Science Research Centre
Private Bag 11 222, Palmerston North, New Zealand
Email: [m.kirchberg|f.din|k.d.schewe|a.tretiakov]@massey.ac.nz

Abstract

XML-based databases have become a major area of interest in database research. Abstractly speaking they can be considered as a resurrection of complex-value databases using constructors for records, lists, unions plus optionality and references. XQuery has become the standard query language for XML. In this paper an implementation of XQuery based on linguistic reflection is proposed. That is, XQuery is translated into a query algebra for rational tree types based on simple operations and structural recursion for lists. The major purpose of using reflection is to expand path expressions in a type-safe way.

1 Introduction

As emphasised in (Lobin 2001) the original purpose of XML – same as SGML – was to support the description of content rather than layout of text documents. For instance, in the Orlando project (Ruecker 2000) XML is used to markup the content of novels of female British and Irish writers. Nevertheless, XML has become a major area of interest in database research. If some aspects of XML that make sense for text markup but not so much for databases are neglected, XML can be considered as a complex-value data model using constructors for records, lists, unions plus optionality and references (Abiteboul, Buneman & Suciú 2000, Siméon & Wadler 2003).

Using XML for databases requires schema definition, query and update languages. By now, XML Schema (World Wide Web Consortium (W3C) 2001) has become the W3C standard for defining schemata, while XQuery (Katz 2003, World Wide Web Consortium (W3C) 2004) is the recommended standard for querying XML documents. For updates only little work has been done so far, e.g. (Tatarinov, Ives, Halevy & Weld 2001).

In fact, XML Schema supports almost directly the definition of tree types using the mentioned constructors. XQuery combines ideas from various predecessor proposals for XML query languages (Abiteboul, Quass, McHugh, Widom & Wiener 1997, Buneman, Davidson, Hillebrand & Suciú 1996, Deutsch, Fernandez, Florescu, Levy & Suciú 1999). Most importantly, queries are composed of a matching part that binds variables to values according to a given XML document, and a construction part that creates new XML

documents from these variables.

When it comes to implementing XML databases and in particular XQuery, there are two major lines of research. The first one, e.g. (DeHaan, Toman, Consens & Özsu 2003), attempts a translation to SQL based on a reification of XML via relational database technology. The drawback of this approach is that semantics may be lost in the translation from trees to relations. The alternative is to approach a direct implementation of XQuery, e.g. (Paparizos, Wu, Lakshmanan & Jagadish 2004). Our own research follows this second line.

The major difficulty in implementing XQuery results from path expressions, i.e. from the fragment of the language that subsumes XPath (Chen, Davidson & Zheng 2004). These difficulties are supported by the theoretical analysis of the complexity of XPath (Gottlob, Koch & Pichler 2003, Marx 2004).

Our work reported in this paper follows the idea of translating XQuery to a (general purpose) query algebra for rational trees as defined in (Schewe 2001). For short let RTA denote this query algebra. RTA uses operators for the type system, i.e. for the abstract system of types as defined in (Katz 2003). In particular, it exploits structural recursion (Tannen, Buneman & Wong 1992, Wadler 1992) for lists. Koch in (Koch 2005) has used a similar approach based on list comprehensions. However, his work is placed in a complexity-theoretic setting. The advantages of such a translation to a query algebra are the support of algebraic query optimisation, the easy implementation of the operations and the integration with programming languages, e.g. using the physical architecture from (Kirchberg, Schewe & Tretiakov 2003), and the easy extension to other constructors such as sets and multisets in case the order that comes with the list constructor is considered unnecessary or even undesired.

However, the translation of XQuery to RTA requires schema information, in particular for the path expressions. Thus, it is a natural idea to exploit type-safe linguistic reflection (Stemple, Fegaras, Sheard & Socorro 1990) to deal with this problem. This is what we do in this paper, i.e. we present a translation from essential parts of XQuery to RTA and show how this translation benefits from linguistic reflection.

Therefore, we first introduce an abstract model of XML, XQuery and RTA in Sections 2 and 3, respectively. In Section 4 we then outline the translation from XQuery to RTA. We focus on structural recursion and show that some of the functions used as parameters have a “complicated nature”, as they refer to path finding. For this we introduce linguistic reflection and show how it can be used to expand these functions of “complicated nature”. We conclude with a brief summary.

* The work reported in this paper was supported by FRST/NERF grant MAUX0025 “DIMO – Distributed Multi-Level Object Bases”.

Copyright ©2006, Australian Computer Society, Inc. This paper appeared at Seventeenth Australasian Database Conference (ADC2006), Hobart, Australia. Conferences in Research and Practice in Information Technology, Vol. 49. Gill Dobbie and James Bailey, Ed. Reproduction for academic, not-for profit purposes permitted provided this text is included.

2 Abstract Model of XML and XQuery

In this section we describe some basics of XML and XQuery. Of course, as both of these are complex languages, we cannot describe all details and therefore take a more abstract view focusing more on the semantics than on the syntax.

2.1 XML Documents as Trees

Start with a type system that supports records, lists and unions. Using abstract syntax this type system can be described by

$$t = b \mid (t_1, \dots, t_n) \mid [t] \mid t_1 \oplus \dots \oplus t_n.$$

Here b represents a (not further specified) collection of base types, e.g. the base types supported by XML such as *String*, *Integer*, *Double*, *ID*, etc. For reasons that will become clear, when we add references, we only use a single type *ID* for identifiers. Furthermore, assume that one of the base types is *Empty* with only one possible value. This type can be used to support optionality.

We use (t_1, \dots, t_n) to denote an ordered record type with component types t_i , the type $[t]$ is used for finite lists, and $t_1 \oplus \dots \oplus t_n$ is used for a (disjoint) union type with components t_i .

Each type t denotes a set of values called its *domain* $dom(t)$. Formally, we obtain these domains as follows:

- $dom(b_i) = V_i$, i.e. for each base type b_i we assume some set V_i of values of that type, e.g. $dom(EMPTY) = \{\perp\}$.
- $dom((t_1, \dots, t_n)) = dom(t_1) \times \dots \times dom(t_n)$.
- $dom([t]) = \{[v_1, \dots, v_k] \mid k \in \mathbb{N}, v_i \in dom(t)\}$.
- $dom(t_1 \oplus \dots \oplus t_n) = \{(i, v_i) \mid 1 \leq i \leq n, v_i \in dom(t_i)\}$.

Then an XML document can be represented by a value of some type t , which in turn is representable as a tree, provided the document does not contain references. In particular, we can treat attributes in the same way as subelements – which is no loss of generality for databases, whereas for text markup it may make a significant difference.

In order to also capture references, we extend the type system to

$$t = b \mid \ell \mid (t_1, \dots, t_n) \mid [t] \mid t_1 \oplus \dots \oplus t_n \mid \ell : t,$$

where ℓ represents reference labels. The domains are simply $dom(\ell) = dom(ID)$ and $dom(\ell : t) = \{(i, v) \mid i \in dom(\ell), v \in dom(t)\}$. Following (Abiteboul et al. 2000) each occurrence of a value i of type *ID* in some complex value v that corresponds to a labelled type $\ell : t$ defines a reference, whereas each occurrence of a value i of type *ID* in v that corresponds to a label ℓ uses the reference. In XML Schema the usage of references corresponds to the type *IDREF*, whereas the definition of references corresponds to the type *ID*. Furthermore, *IDREFS* corresponds to a list type $[\ell]$ – in fact, here we would prefer to use a set type, but for simplicity and orthogonality of the constructors let us use only one bulk type constructor.

EXAMPLE 2.1 Let us look at the following schema definition in XML Schema:

```
<xs:schema
  xmlns:xs="http://www.w3.org/2001/XMLSchema"
  <xs:element name="cellar">
    <xs:complexType>
      <xs:sequence>
        <xs:element ref="wines"/>
        <xs:element ref="wineries"/>
        <xs:element ref="regions"/>
      </xs:sequence>
    </xs:complexType>
  </xs:element>
  <xs:element name="wines">
    <xs:complexType>
      <xs:sequence>
        <xs:element ref="wine"
          minOccurs="0" maxOccurs="unbounded"/>
      </xs:sequence>
    </xs:complexType>
  </xs:element>
  <xs:element name="wineries">
    <xs:complexType>
      <xs:sequence>
        <xs:element ref="winery"
          minOccurs="0" maxOccurs="unbounded"/>
      </xs:sequence>
    </xs:complexType>
  </xs:element>
  <xs:element name="regions">
    <xs:complexType>
      <xs:sequence>
        <xs:element ref="region"
          minOccurs="0" maxOccurs="unbounded"/>
      </xs:sequence>
    </xs:complexType>
  </xs:element>
  <xs:element name="wine">
    <xs:complexType>
      <xs:sequence>
        <xs:element name="name" type="xs:string"/>
        <xs:element name="year"
          type="xs:integer"
          minOccurs="0"/>
        <xs:element ref="blend"
          maxOccurs="unbounded"/>
        <xs:element name="price"
          type="xs:decimal"/>
      </xs:sequence>
        <xs:attribute name="w-id"
          type="xs:ID" use="required"/>
        <xs:attribute name="producer"
          type="xs>IDREF" use="required"/>
      </xs:complexType>
    </xs:element>
  <xs:element name="blend">
    <xs:complexType>
      <xs:sequence>
        <xs:element name="grape"
          type="xs:string"/>
        <xs:element name="percentage"
          type="xs:integer"/>
      </xs:sequence>
    </xs:complexType>
  </xs:element>
  <xs:element name="winery">
    <xs:complexType>
      <xs:sequence>
        <xs:element name="name" type="xs:string"/>
        <xs:element name="owner"
          type="xs:string" maxOccurs="unbounded"/>
        <xs:element name="area"
          type="xs:string"
          minOccurs="0"/>
        <xs:element name="established"
          type="xs:date" minOccurs="0"/>
      </xs:sequence>
        <xs:attribute name="v-id"
          type="xs:ID" use="required"/>
        <xs:attribute name="in-region"
          type="xs>IDREF" use="required"/>
      </xs:complexType>
    </xs:element>
  <xs:element name="region">
    <xs:complexType>
```

```

<xs:sequence>
  <xs:element name="name" type="xs:string"/>
</xs:sequence>
<xs:attribute name="r-id"
  type="xs:ID" use="required"/>
<xs:attribute name="famous-wines"
  type="xs:IDREFS" use="required"/>
</xs:complexType>
</xs:element>
</xs:schema>

```

That is, a cellar contains a list of wines, wineries and regions. A wine is described by a name, a year (optional) and a blend, which is a sequence of grapes together with their percentages. A winery is described by a name, a list of owners, an area (optional) and an establishment date (optional). A region just has a name. Furthermore, there are references from a wine to the winery that produces it, from a winery to the region it is located in, and from a region to all its famous wines.

Using our type system, we obtain the following complex type definitions for representing this schema:

```

cellar = (wines, wineries, regions)
wines = [w-id : wine]
wineries = [v-id : winery]
regions = [r-id : region]
wine = (w-name, year ⊕ Empty, [blend], price, producer)
w-name = String
year = Integer
price = Decimal
producer = v-id
blend = (grape, percentage)
grape = String
percentage = Integer
winery = (v-name, [owner], area ⊕ Empty,
  established ⊕ Empty, in-region)
v-name = String
owner = String
area = String
established = Date
in-region = r-id
region = (r-name, famous-wines)
r-name = String
famous-wines = [w-id]

```

Here w-id, v-id and r-id are labels.

EXAMPLE 2.2 Consider the following XML document that is in accordance with the schema defined in Example 2.1:

```

<cellar
  xmlns:xsi=
    "http://www.w3.org/2001/XMLSchema-instance"
  xsi:noNamespaceSchemaLocation="cellar.xsd">
  <wines>
    <wine w-id="o11" producer="o1">
      <name>Marlborough Riesling</name>
      <year>2003</year>
      <blend>
        <grape>Riesling</grape>
        <percentage>100</percentage>
      </blend>
      <price>13.95</price>
    </wine>
    <wine w-id="o12" producer="o1">
      <name>Marlborough Gewurztraminer</name>
      <year>2000</year>
      <blend>
        <grape>Gewurztraminer</grape>
        <percentage>100</percentage>
      </blend>
      <price>17.95</price>
    </wine>
    <wine w-id="o13" producer="o1">
      <name>Everyday's Favourite</name>
      <blend>

```

```

        <grape>Sauvignon Blanc</grape>
        <percentage>65</percentage>
      </blend>
      <blend>
        <grape>Semillon</grape>
        <percentage>35</percentage>
      </blend>
      <price>5.95</price>
    </wine>
  </wines>
  <wineries>
    <winery v-id="o1" in-region="o2">
      <name>Marlborough Winery</name>
      <owner>Jacques Vine</owner>
      <owner>Claudine Vine</owner>
      <area>231 ha</area>
      <established>1987-01-01</established>
    </winery>
  </wineries>
  <regions>
    <region r-id="o2" famous-wines="o11 o12">
      <name>Marlborough</name>
    </region>
  </regions>
</cellar>

```

This XML document can be represented by the following complex value:

```

( [ (&o11, (Marlborough Riesling, 2003,
  [(Riesling, 100)], 13.95, &o1)),
  (&o12, (Marlborough Gewurztraminer, 2000,
  [(Gewurztraminer, 100)], 17.95, &o1)),
  (&o13, (Everyday's Favourite, ⊥,
  [(Sauvignon Blanc, 65), (Semillon, 35)],
  5.95, &o1))),
  [ (&o1, (Marlborough Winery, [Jacques Vine,
  Claudine Vine], 231 ha, 1987, &o2)),
  [ (&o2, (Marlborough, [&o11, &o12]))] ] )

```

2.2 XQuery in a Nutshell

XQuery is a query language allowing to extract sequences of subtrees and base type values from any number of XML document trees, and to combine them to construct a sequence of trees and basic values (the so-called **items**) forming the result of the query. In practice, most often the result of the query is a sequence consisting of a single tree.

In XQuery, the XML documents serving as input are identified by using the so-called input functions, of which the most commonly used one is `doc`, which accepts a URL corresponding to the location of an XML document as a parameter. For example, `doc("cellar.xml")` would retrieve the `cellar.xml` document from the current directory.

Sequences of subtrees are retrieved by using the so-called path expressions, consisting of one or more **steps** separated by a slash, /, or double slash, //. Each step acts on the sequence of items created by the previous step to form a further sequence, which either forms the output of the path expression (if the step is the last one), or serves as input for further steps. The following query, formed by combining an input function with a path expression, will result in a sequence of `name` elements representing wine names (assuming `cellar.xml` is the XML document introduced in the previous section): `doc("cellar.xml")/cellar/wines/wine/name`.

`doc("cellar.xml")/cellar` results in a sequence consisting of a single `cellar` element, `doc("cellar.xml")/cellar/wines` results in a sequence consisting of a single `wines` element (a subelement of the `cellar` element obtained in the previous step), `doc("cellar.xml")/cellar/wines/wine` will result in a sequence of all `wine` elements from `wines` and so on. Filtering can be applied to restrict which

of the items are to be included in a given step. While `/` retrieves child items (branches immediately connected to the root), `//` forms a sequence consisting of all matching subtrees, at all depths. `/` and `//` are illustrated in examples 2.3 and 2.4, respectively.

As XQuery is a functional language, an XQuery program can be regarded as an expression formed by subexpressions which, at execution time, are evaluated in the order of precedence. The most commonly used type of expressions in XQuery are the so-called FLWOR (for, let, where, order by, return) expressions. In a FLWOR expression, a `for` clause binds each item of a sequence to a variable, and evaluates the rest of the expression with that binding, resulting in as many evaluations as there are items in the sequence. The `for` clause is illustrated in Example 2.4 below.

A `let` clause binds the whole sequence to a variable, and evaluates the rest of the expression just once, with that binding. The `let` clause is illustrated in Examples 2.5, 2.6, and 2.7.

The `where` clause serves as a filter: the rest of the FLWOR expression is executed only if the boolean expression associated with the `where` clause evaluates to true. This is illustrated in Example 2.5, 2.6, and 2.7.

The `order by` clause is used for sorting (we do not discuss it here any further).

Finally, the `return` clause is a constructor, instantiating an item that is to be included as the result of the query. By using `return`, items retrieved from different parts of the same document, or from different documents, can be combined together, resulting in sophisticated joins. As shown in Example 2.7, the constructor formed by using the `return` clause can include subqueries, whose output is incorporated into the sequence created by the constructor.

XML Query can make use of type information from XML Schema documents associated with XML documents inputted by the query by explicitly specifying the type of items to be included in sequences or to be constructed. In addition, parsers are able to analyse and to reject a query based on schema information only, if the query is found to construct items that do not match the declared types for constructor output.

EXAMPLE 2.3 Assume the document in Example 2.2 is stored in `cellar.xml`. Then

```
<wines>
{
  doc("cellar.xml")/cellar/wines/wine/name
}
</wines>
```

is a simple query that will select the names of wines. For our example document the result would be

```
<wines>
  <name>Marlborough Riesling</name>
  <name>Marlborough Gewurztraminer</name>
  <name>Everyday's Favourite</name>
</wines>
```

EXAMPLE 2.4 The query

```
<wine-makers>
{
  for $N in doc("cellar.xml")//owner
  return
    <name>{ $N/text() }</name>
}
</wine-makers>
```

returns the names of winery owners.

EXAMPLE 2.5 The following is a query with a more interesting `where`-clause, which returns the names of Riesling wines:

```
<Rieslings>
{
  for $W in doc("cellar.xml")/cellar/wines/wine
  let $N := $W/name, $B := $W/blend
  where $B/grape/text() = "Riesling"
    and $B/percentage/text() = 100
  return <name>{ $N/text() }</name>
}
</Rieslings>
```

EXAMPLE 2.6 The following query, which contains selection conditions on the paths, will produce a list of wines with their producers:

```
<wines>
{
  let $db := doc("cellar.xml")
  for $W in $db//wine, $V in $db//winery
  let $P := $W/@producer, $N := $W/name,
    $M := $V/name, $I := $V/@v-id
  where $I = $P
  return
    <wine>
      <product>{ $N/text() }</product>
      <producer>{ $M/text() }</producer>
    </wine>
}
</wines>
```

EXAMPLE 2.7 The following is an example of a nested query:

```
<wines>
{
  let $db := doc("cellar.xml")
  for $N in $db//wine/name
  return
    <wine>
      { $N }
      {
        for $W in $db//wine
        where $W/name = $N
        return $W/year
      }
    </wine>
}
</wines>
```

The query lists the names of all wines, adding to each name the corresponding production year, when such information is available.

3 RTA: A Rational Tree Query Algebra

Following a basic idea in (Schewe 2001) we use a query algebra with operations “induced” from the type system plus a join-operation. For our purposes here it is more convenient to consider products instead of joins.

In doing so, let $\mathbb{1}$ denote a trivial type with only one value in its domain. We use a unique “forget”-operation $\text{triv} : t \rightarrow \mathbb{1}$ for each type t . Assume further a boolean type $BOOL$ with constant values \mathbf{T} and \mathbf{F} . Thus, we may consider the operations $\wedge : BOOL \times BOOL \rightarrow BOOL$ (conjunction), $\neg : BOOL \rightarrow BOOL$ (negation) and $\Rightarrow : BOOL \times BOOL \rightarrow BOOL$ (implication).

For record types we consider *projection* $\pi_i : (t_1, \dots, t_n) \rightarrow t_i$ and *product* $o_1 \times \dots \times o_n : t \rightarrow (t_1, \dots, t_n)$ for given operations $o_i : t \rightarrow t_i$. As usual, we write π_{i_1, \dots, i_k} as a shortcut for $\pi_{i_1} \times \dots \times \pi_{i_k}$.

For union types we use the canonical embeddings $\iota_i : t_i \rightarrow t_1 \oplus \dots \oplus t_n$. Other operations on union types take the form $o_1 + \dots + o_n : t_1 \oplus \dots \oplus t_n \rightarrow t$ for given operations $o_i : t_i \rightarrow t$.

For list types we may consider $\hat{\ } (concatenation)$, the constant $\mathbf{empty} : \mathbb{1} \rightarrow [t]$ and the *singleton* operation $\mathbf{single} : t \rightarrow [t]$ with well known semantics.

It should be noted here that document order is preserved through the use of lists. The ordering of the elements in the lists conforms to the ordering of the elements in the queried xml document (or conforms to specific re-ordering in the query itself) throughout the execution process.

3.1 Structural Recursion

In addition, we consider structural recursion $\mathbf{src}[e, g, \sqcup] : [t] \rightarrow t'$ with a value e of type t' , an operation $g : t \rightarrow t'$ and an operation $\sqcup : (t', t') \rightarrow t'$, which is defined as follows:

$$\begin{aligned} \mathbf{src}[e, g, \sqcup]([]) &= e \\ \mathbf{src}[e, g, \sqcup](\langle x \rangle) &= g(x) \\ \mathbf{src}[e, g, \sqcup](X \hat{\ } Y) &= \mathbf{src}[e, g, \sqcup](X) \sqcup \mathbf{src}[e, g, \sqcup](Y) \end{aligned}$$

Let us illustrate structural recursion by some more or less standard examples. First consider an operation $f : t \rightarrow t'$. We want to raise f to an operation $\mathbf{map}(f) : [t] \rightarrow [t']$ by applying f to each element of a list. Obviously, we have $\mathbf{map}(f) = \mathbf{src}([], \mathbf{single} \circ f, \hat{\ })$.

Next consider an operation $\varphi : t \rightarrow \mathit{BOOL}$, i.e. a predicate. We want to define an operation $\mathbf{filter}(\varphi) : [t] \rightarrow [t]$, which maps a given list to the sublist of all elements satisfying the predicate φ . For this we may write $\mathbf{filter}(\varphi) =$

$$\mathbf{src}([], \mathbf{if_else} \circ (\varphi \times \mathbf{single} \times (\mathbf{empty} \circ \mathbf{triv})), \hat{\ })$$

with the operation $\mathbf{if_else} : (\mathit{BOOL}, t, t) \rightarrow t$ with $(\mathbf{T}, x, y) \mapsto x$ and $(\mathbf{F}, x, y) \mapsto y$.

As a third example assume that t is a type, on which addition $+$: $(t, t) \rightarrow t$ is defined. Then $\mathbf{src}[0, \mathit{id}, +]$ with the identity id on t defines the sum of the elements in a list.

3.2 Querying XML with RTA

Let us simply illustrate now how RTA can be applied to query XML. We will use the queries from the previous section and write equivalent queries in RTA.

EXAMPLE 3.1 Let us consider first the query in Example 2.3. In this case we basically have to analyse a path expression. For this assume that v_{in} is the complex value in Example 2.2, i.e. it represents the corresponding XML document `cellar.xml`.

The construct `doc(cellar.xml)/cellar` creates a list with the whole document as its only entry, which corresponds to applying the RTA-operation \mathbf{single} to v_{in} . Then `/wines` selects the first successor of the root. As v_{in} is a triple, this corresponds to applying $\mathbf{map}(\pi_1)$ to $[v_{\text{in}}]$. This gives

$$\begin{aligned} \mathbf{map}(\pi_1)([v_{\text{in}}]) &= \mathbf{src}([], \mathbf{single} \circ \pi_1, \hat{\})([v_{\text{in}}]) \\ &= \mathbf{single} \circ \pi_1(v_{\text{in}}) \\ &= [\pi_1(v_{\text{in}})] \end{aligned}$$

The effect of `/wines` in the XQuery path expression is to produce only the list of wines, i.e. $\pi_1(v_{\text{in}})$. This can be achieved by another application of structural recursion, in this case $\mathbf{src}([], \mathit{id}, \hat{\ })$. This gives

$$\mathbf{src}([], \mathit{id}, \hat{\})([\pi_1(v_{\text{in}})]) = \pi_1(v_{\text{in}})$$

as desired. Finally, the effect of `/name` in the XQuery path expression is first to throw away the

identifiers for wines, which can be achieved by applying π_2 , then taking the first component, i.e. to apply π_1 . Thus, the last step is the application of $\mathbf{map}(\pi_1 \circ \pi_2)$.

In summary, the query in Example 2.3 corresponds to the query

$$\mathbf{map}(\pi_1 \circ \pi_2) \circ \mathbf{src}([], \mathit{id}, \hat{\ }) \circ \mathbf{map}(\pi_1) \circ \mathbf{single}.$$

Applied to v_{in} we obtain the list value `[Marlborough Riesling, Marlborough Gewurztraminer, Everyday's Favourite]` as desired.

Example 3.1 already gives valuable hints, how a translation of XQuery into RTA might work. Basically, we follow the execution model for XQuery, which works on lists and applies operations to the elements of the list. So, basically each step corresponds to some structural recursion operation.

Example 3.1 also indicates the expected advantage from the translation into RTA, as we were able to simplify the algebraic query. This is a first step towards query "optimisation".

However, in Example 3.1 we used only explicit path expressions. The next example handles a query, in which we have to search for the path. We will see that this constitutes a much more complicated application of structural recursion.

EXAMPLE 3.2 Let us now consider the query in Example 2.4. As in the previous example we first have to apply \mathbf{single} to v_{in} to achieve the same effect as `doc(cellar.xml)` in the XQuery path expression. However, the follow-on RTA-operation has to capture the effect of `//owner`, which can be done by structural recursion. That is, we apply $\mathbf{src}([], h, \hat{\ })$ to $[v_{\text{in}}]$ with an operation h that searches for successors with the name `owner`.

The application of this operation h to some x can be defined recursively as follows:

```

if type(x) = owner
then [x]
else
  if type(x) = (t1, ..., tn)
  then h(π1(x))  $\hat{\ } \dots \hat{\ } h(\pi_n(x))$ 
  else
    if type(x) = [t]
    then src([], h,  $\hat{\ })(x)$ 
    else
      if type(x) = t1  $\oplus \dots \oplus$  tn
      then h(π2(x))
      else []
    endif
  endif
endif
endif

```

Finally, we can neglect the return-clause, as it is just a renaming of tags, which do not appear in our anonymised complex values.

In summary, the corresponding RTA-query is $\mathbf{src}([], h, \hat{\ }) \circ \mathbf{single}$ with

$$\begin{aligned} h &= \mathbf{if_then} \circ (\varphi_1 \times \mathbf{single} \times h_1) \\ h_1 &= \mathbf{if_then} \circ (\varphi_2 \times (\hat{\ } \circ (h \circ \pi_1 \times \hat{\ } \circ (\dots \\ &\quad (h \circ \pi_{n-1} \times h \circ \pi_n) \dots))) \times h_2) \\ h_2 &= \mathbf{if_then} \circ (\varphi_3 \times \mathbf{src}([], h, \hat{\ }) \circ h_3) \\ h_3 &= \mathbf{if_then} \circ (\varphi_4 \times h \circ \pi_2 \times \mathbf{empty}) \end{aligned}$$

and the obvious Boolean operations

$$\begin{aligned}\varphi_1(x) &\equiv \text{type}(x) = \text{owner} \\ \varphi_2(x) &\equiv \text{type}(x) = (t_1, \dots, t_n) \\ \varphi_3(x) &\equiv \text{type}(x) = [t] \\ \varphi_4(x) &\equiv \text{type}(x) = t_1 \oplus \dots \oplus t_n\end{aligned}$$

Example 3.2 shows that some of the operations used within RTA-queries require complex definitions. It is not difficult to see that the other examples of queries from the previous section require analogous techniques. We will present a general solution for the translation in Section 4.

3.3 Multi-Dimensional Extension

Let us finally mention a “multi-dimensional” extension of structural recursion, but let us restrict for simplicity to the binary case. That is, we define an operation $\text{src2}[f, g, h] : ([t_1], [t_2]) \rightarrow t$ with parameters $f : [t_2] \rightarrow t$, $g : (t_1, [t_2]) \rightarrow t$, and $h : (t, t) \rightarrow t$. Similarly to the “one-dimensional” case we define

$$\begin{aligned}\text{src2}[f, g, h]([], L_2) &= f(L_2) \\ \text{src2}[f, g, h]([x], L_2) &= g(x, L_2) \\ \text{src2}[f, g, h](X \cap Y, L_2) &= h(\text{src2}[f, g, h](X, L_2), \\ &\quad \text{src2}[f, g, h](Y, L_2))\end{aligned}$$

This can be used to define the product of lists (both of type $[t]$) as

$$L_1 \times L_2 = \text{src2}([], g, \wedge)(L_1, L_2),$$

where $[]$ is treated as a constant function, and g is defined by

$$g(x, L_2) = \text{src}([], \text{single} \circ (x \times \text{id}), \wedge)(L_2).$$

4 Linguistic Reflection in Translating XQuery to RTA

Our goal is to translate XQuery into RTA. For this recall that XQuery is basically a functional language, so each query corresponds to a sequence of function calls. For instance, for the simple query in Example 2.3 we would first evaluate $\langle \text{wines} \rangle$ by simply printing it, then evaluate the expression $\{ \text{doc}(\text{cellar.xml})/\dots \}$, finally evaluate $\langle / \text{wines} \rangle$, which again amounts only to a simple print. Therefore, we concentrate on expressions of the form $\{ \dots \}$ with the dots standing for a FLWOR-expression.

4.1 The Basic Translation Model

XQuery works on lists, and each part of a query corresponds to some function that is executed on all elements of the list. As we assume to be given a FLWOR-expression, we first look at the **for**-construct. In its simple form it has the form

for $\$X$ **in** $\langle \text{path-expression} \rangle$,

so we have to evaluate the path expression first:

- If $\text{doc}(\text{xxx.xml})$ appears in the path expression, then xxx.xml is some input document, which is represented by some complex value, say v_{in} . As we have already seen in Examples 3.1 and 3.2, the input-function doc simply corresponds to the RTA-operation **single**.

- If p/name appears in the path expression, we first translate p , say the result is $\text{trans}(p)$. Then $/\text{name}$ gives rise to an application of structural recursion, say $\text{src}([], g/\text{name}, \wedge)$. Thus, the translation of p/name is

$$\text{trans}(p/\text{name}) = \text{src}([], g/\text{name}, \wedge)(\text{trans}(p)).$$

The difficult part is then to determine the operation g/name . Note that all applications of structural recursion in Example 3.1 refer to this step.

- If $p//\text{name}$ appears in the path expression, we proceed analogously. That is, $//\text{name}$ gives rise to an application of structural recursion $\text{src}([], g//\text{name}, \wedge)$, and the translation of $p//\text{name}$ is

$$\text{trans}(p//\text{name}) = \text{src}([], g//\text{name}, \wedge)(\text{trans}(p)).$$

Note that the structural recursion in Example 3.2 refers to this step. It also indicates how to define $g = g//\text{name}$ in general:

$$\begin{aligned}g &= \text{if_then} \circ (\varphi_1 \times \text{single} \times h_1) \\ h_1 &= \text{if_then} \circ (\varphi_2 \times (\wedge \circ (g \circ \pi_1 \times \wedge \circ (\dots \\ &\quad (g \circ \pi_{n-1} \times g \circ \pi_n) \dots))) \times h_2) \\ h_2 &= \text{if_then} \circ (\varphi_3 \times \text{src}([], g, \wedge) \circ h_3) \\ h_3 &= \text{if_then} \circ (\varphi_4 \times g \circ \pi_2 \times \text{empty})\end{aligned}$$

with the Boolean operations

$$\begin{aligned}\varphi_1(x) &\equiv \text{type}(x) = \text{name} \\ \varphi_2(x) &\equiv \text{type}(x) = (t_1, \dots, t_n) \\ \varphi_3(x) &\equiv \text{type}(x) = [t] \\ \varphi_4(x) &\equiv \text{type}(x) = t_1 \oplus \dots \oplus t_n\end{aligned}$$

The crucial remaining part is to take care of the Boolean operations, as these require type-checks.

- If $p[\text{test}]$ appears in the path expression, we first translate p . Furthermore, test will be translated into a Boolean condition ψ , and we can combine both using structural recursion, in this case a **filter**-operation, i.e.

$$\text{trans}(p[\text{test}]) = \text{filter}(\psi)(\text{trans}(p)).$$

If there is more than one condition in the **for**-clause, say

for $\$X_1$ **in** $\langle \text{path-expression}_1 \rangle, \dots$
 $\$X_n$ **in** $\langle \text{path-expression}_n \rangle$,

each path expression will be translated separately resulting in RTA-operations o_1, \dots, o_n , each producing some list, say L_i . Then we have to combine these lists into one list containing all tuple combinations, i.e. we obtain $L_1 \times \dots \times L_n$.

The following **let**-clause simply binds further variables depending on the list resulting from evaluating the **for**-clause. As this may again require evaluating a path expression, we proceed analogously to translating the **for**-clause.

EXAMPLE 4.1 Look at the query in Example 2.5. Analogous to Example 3.1 the **for**-clause will be translated to the operation

$$\text{src}([], \text{id}, \wedge) \circ \text{map}(\pi_1) \circ \text{single},$$

which will be applied to v_{in} . Now the first part of the **let**-clause corresponds to $\text{map}(\pi_1 \circ \pi_2)$ as already seen in Example 3.1. Similarly, the second part of the **let**-clause corresponds to $\text{map}(\text{first} \circ \pi_3 \circ \pi_2)$ with an operation **first** that selects the first element of a list.

However, we do not want to replace the $\$W$ -values by the $\$N$ -values or the $\$B$ -values, but keep all three, so the **let**-clause defines the operation

$$\text{map}((\text{id} \times \pi_1 \times (\text{first} \circ \pi_3)) \circ \pi_2).$$

The remaining clauses in FLWOR expressions are easy to handle. A **where**-clause defines a **filter**-operation. The greatest difficulty is to determine the Boolean operation, which may again involve the translation of a path expression. An **order**-clause corresponds to applying a sorting-operation, which can be expressed by structural recursion. Finally, the **return**-clause only constructs a value, so the only difficulty that may occur is that this construction involves evaluating another query.

EXAMPLE 4.2 Let us continue our previous example, as Example 2.5 contains a **where**-clause. The list resulting from the application of the operation in Example 4.1, which represents the combination of the **for**- and **let**-clause, contains triples, where the first component corresponds to a wine, the second one to its name and the third one to the first-listed component of its blend. Thus, applying $\pi_1 \circ \pi_3$ to such a triple gives the requested name of the first grape, while the application of $\pi_2 \circ \pi_3$ results in the corresponding percentage.

Thus, the first condition in the **where**-clause corresponds to the Boolean operation $\text{eq} \circ ((\pi_1 \circ \pi_3) \times \text{Riesling})$, in which Riesling is treated as a constant operation. Similarly, the second condition gives the Boolean operation $\text{eq} \circ ((\pi_2 \circ \pi_3) \times 100)$, and thus, the whole **where**-clause corresponds to the RTA-operation $\text{filter}(\varphi)$, where φ is defined by the Boolean operation

$$\wedge \circ ((\text{eq} \circ ((\pi_1 \circ \pi_3) \times \text{Riesling})) \times (\text{eq} \circ ((\pi_2 \circ \pi_3) \times 100))).$$

Finally, let us complete the translation of the query in Example 2.5. Taking all the parts together, we obtain the RTA-operation

$$\begin{aligned} & \text{map}(\pi_2) \circ \text{filter}(\varphi) \circ \\ & \text{map}((\text{id} \times \pi_1 \times (\text{first} \circ \pi_3)) \circ \pi_2) \circ \\ & \text{src}[\ [], \text{id}, \wedge] \circ \text{map}(\pi_1) \circ \text{single} \end{aligned}$$

4.2 Type-Safe Linguistic Reflection

In the previous subsection we have seen that the translation of XQuery can be done by parsing through FLWOR expressions and translating them step-by-step into RTA-operations, most of which happen to be special cases of structural recursion. More than this, all applications of structural recursion have the form $\text{src}[\ [], g, \wedge]$ and the real difficulty is to determine the functions g . For this we identify two cases:

- We obtain a highly recursive operation g that searches through the whole document. Example 3.2 is a prototype for this case.
- We obtain an operation that is determined by the schema. Example 3.1 is a prototype for this case.

As the chances for query optimisation are much higher in the second case – as already seen in Example 3.1 – it will be advantageous to apply this case,

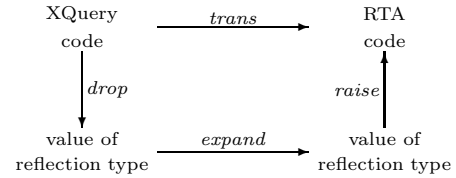


Figure 1: Linguistic Reflection

wherever it is possible. However, this means to explore the schema. As shown in (Stemple et al. 1990) a type-safe way of doing this is to apply linguistic reflection.

Linguistic reflection is the ability of a system to observe and manipulate its own components. This is done by extending the system with extra modules which are created, compiled and linked in by the system itself, either during execution or at compile-time. The language in which the system has been written would, of course, need to provide the ability for the system to behave in this manner.

The general idea is to consider constructs in a query that are used for defining an operation, such as $/\text{name}$ for g/name in the previous subsection, as macros that have to be expanded. This can be done by ignoring that they represent query code, thus *drop* this meaning, and treat them as values of some type. The expansion function will then take this value plus the schema, which is represented as a value of some other type, and create a new value. This new value will finally be *raised* back to an executable operation. This is illustrated by Figure 1.

Therefore, we will define reflection types in the next subsection, and finally illustrate, how the expansion procedure for paths works.

4.3 Reflection Types

In order to represent XSchema schemata we need at least reflection types for types, elements, attributes, and schemata. So we get the reflection type $\text{type}_{\text{rep}} = \text{Xtype}_{\text{rep}} \oplus \text{RTtype}_{\text{rep}}$ indicating that we are using types within XSchema and the rational tree types. For the types that are used to describe XSchema types we then get the following definitions:

$$\begin{aligned} \text{Xtype}_{\text{rep}} &= \text{xs_complex_type}_{\text{rep}} \oplus \text{xs_simple_type}_{\text{rep}} \\ \text{xs_simple_type}_{\text{rep}} &= \text{String} \\ \text{xs_complex_type}_{\text{rep}} &= \text{xs_sequence}_{\text{rep}} \oplus \text{xs_choice}_{\text{rep}} \\ \text{xs_sequence}_{\text{rep}} &= [(\text{name}_{\text{rep}} \oplus \text{xs_element}_{\text{rep}}, \\ & \quad \text{min}, \text{max} \oplus \text{Empty})] \\ \text{xs_choice}_{\text{rep}} &= [(\text{name}_{\text{rep}} \oplus \text{xs_element}_{\text{rep}}, \\ & \quad \text{min}, \text{max} \oplus \text{Empty})] \\ \text{min} &= \text{Integer} \\ \text{max} &= \text{Integer} \\ \text{name}_{\text{rep}} &= \text{String} \\ \text{xs_element}_{\text{rep}} &= (\text{name}_{\text{rep}}, \text{Xtype}_{\text{rep}}, [\text{xs_attribute}_{\text{rep}}]) \\ \text{xs_attribute}_{\text{rep}} &= (\text{name}_{\text{rep}}, \text{xs_simple_type}_{\text{rep}}, \text{use}_{\text{rep}}) \\ \text{use}_{\text{rep}} &= \text{String} \end{aligned}$$

and finally

$$\text{XSchema}_{\text{rep}} = (\text{namespace}_{\text{rep}}, \text{name}_{\text{rep}}, [\text{xs_element}_{\text{rep}}])$$

$$\text{namespace}_{\text{rep}} = \text{String}$$

EXAMPLE 4.3 The value $(1, (1, (1, [(1, \text{"wines"}), 1, (1, 1)], ((1, \text{"wineries"}), 1, (1, 1))), ((1, \text{"regions"}), 1, (1, 1))))$ of type type_{rep} represents the complex type used for the element “cellar” in Example 2.1. The leading 1s indicate that it is a

value of an XSchema type, a complex type, and a sequence type, respectively. Note that the min- and max-values are the defaults.

Analogously, the value $(1, (1, (1, [(1, \text{"region"}), 0, (2, \perp)])))$ represents the complex type for the element "regions" in Example 2.1.

The value $(\text{"http://www.w3.org/2001/XMLSchema"}, \text{"cellar"}, [(\text{"cellar"}, (1, (1, (1, [(1, \text{"wines"}), 1, (1, 1)], ((1, \text{"wineries"}), 1, (1, 1)), ((1, \text{"regions"}), 1, (1, 1)])))]), \dots])$ of type $\text{XSchema}_{\text{rep}}$ represents the schema in Example 2.1. Here the dots stand for representations of all the elements used in the schema.

The value $(\text{"wine"}, (1, (1, [((2, (\text{"name"}, (2, \text{"String"}), [])), 1, (1, 1)), ((2, (\text{"year"}, (2, \text{"Integer"}), [])), 0, (1, 1)), ((1, \text{"blend"}), 1, (2, \perp)), ((2, (\text{"price"}, (2, \text{"Decimal"}), [])), 1, (1, 1))])), [(\text{"w-id"}, \text{"ID"}, \text{"required"}), (\text{"producer"}, \text{"IDREF"}, \text{"required"})])$ of type $\text{xs.element}_{\text{rep}}$ represents the element specification for "wine" in Example 2.1. It would of course be part of the value representing the schema.

Analogously, for the types used with RTA we obtain the following reflection types:

$$\begin{aligned} \text{RTtype}_{\text{rep}} &= (\text{name}_{\text{rep}}, \text{type_exp}_{\text{rep}}) \\ \text{type_exp}_{\text{rep}} &= \text{base_type}_{\text{rep}} \oplus \text{label}_{\text{rep}} \oplus \\ &\quad \text{record_type}_{\text{rep}} \oplus \text{list_type}_{\text{rep}} \oplus \text{union_type}_{\text{rep}} \\ &\quad \oplus \text{labelled_type}_{\text{rep}} \oplus \text{name}_{\text{rep}} \\ \text{base_type}_{\text{rep}} &= \text{String} \\ \text{label}_{\text{rep}} &= \text{String} \\ \text{record_type}_{\text{rep}} &= [\text{type_exp}_{\text{rep}}] \\ \text{list_type}_{\text{rep}} &= \text{type_exp}_{\text{rep}} \\ \text{union_type}_{\text{rep}} &= [\text{type_exp}_{\text{rep}}] \\ \text{labelled_type}_{\text{rep}} &= (\text{label}_{\text{rep}}, \text{type_exp}_{\text{rep}}) \end{aligned}$$

EXAMPLE 4.4 The type winery from Example 2.1 will be represented by the value $(\text{"winery"}, (3, [(7, \text{"v-name"}), (4, (7, \text{"owner"})), (5, [(7, \text{"area"}), (1, \text{"Empty"})]), (5, [(7, \text{"established"}), (1, \text{"Empty"})]), (7, \text{"in-region"})]))$ of type $\text{RTtype}_{\text{rep}}$.

Similarly, the type in-region is represented by $(\text{"in-region"}, (2, \text{"r-id"}))$, and the type wineries is represented by the value $(\text{"wineries"}, (4, (6, (\text{"w-id"}, (7, \text{"wine"}))))$.

Finally, we also need representation types for the RTA-operations. For this, the following is sufficient:

$$\begin{aligned} \text{Operation}_{\text{rep}} &= \text{base_op}_{\text{rep}} \oplus \text{projection}_{\text{rep}} \\ &\quad \oplus \text{product}_{\text{rep}} \oplus \text{embedding}_{\text{rep}} \\ &\quad \oplus \text{sum}_{\text{rep}} \oplus \text{src}_{\text{rep}} \oplus \text{composition}_{\text{rep}} \\ \text{base_op}_{\text{rep}} &= \text{String} \\ \text{projection}_{\text{rep}} &= \text{Integer} \\ \text{product}_{\text{rep}} &= [\text{Operation}_{\text{rep}}] \\ \text{embedding}_{\text{rep}} &= \text{Integer} \\ \text{sum}_{\text{rep}} &= [\text{Operation}_{\text{rep}}] \\ \text{src}_{\text{rep}} &= \text{Operation}_{\text{rep}} \times \text{Operation}_{\text{rep}} \times \text{Operation}_{\text{rep}} \\ \text{composition}_{\text{rep}} &= [\text{Operation}_{\text{rep}}] \end{aligned}$$

EXAMPLE 4.5 The values $(1, \text{"single"})$ and $(1, \text{"concat"})$ represent the operations **single** and concatenation \frown on lists, respectively. The value $(6, ((1, \text{"empty"}), (2, 3), (1, \text{"concat"})))$ represents the operation $\text{src}[\perp, \pi_3, \frown]$. The value $(7, [(2, 2), (2, 1), (1, \text{"single"})])$ represents the operation $\pi_2 \circ \pi_1 \circ \text{single}$.

4.4 The Expansion Procedure for Paths

Let us now look at the problem of expanding paths, as this turned out to be the core of the translation problem. We have seen above that $/\text{name}$ gives rise to a structural recursion operation $\text{src}[\perp, g/\text{name}, \frown]$, so we have to determine a representation of g/name .

In order to do so, we first determine the RT type that corresponds to an element in the schema using an operation

$$\text{expand_elt_type} : (\text{String}, \text{XSchema}_{\text{rep}}) \rightarrow \text{RTtype}_{\text{rep}},$$

i.e. we associate with an element name and a representation of a schema a rational tree type. This can be achieved by parsing through the schema representation and then applying the rules for type transformation that we used in Section 2. In particular, we blur the differences between subelements and attributes, and we replace references by occurrences of the base type ID :

$$\begin{aligned} \text{expand_elt_type}(n, S) &= \\ &\quad \text{expand_elt_type}'(n, \text{search}(n, \pi_3(S)), S) \\ \text{search}(n, S) &= \\ &\quad \text{if } \pi_1(\text{first}(S)) = n \\ &\quad \text{then } (\pi_2 \times \pi_3)(\text{first}(S)) \\ &\quad \text{else } \text{search}(n, \text{rest}(S)) \\ &\quad \text{endif} \\ \text{expand_elt_type}'(n, (e, L), S) &= \\ &\quad \text{case } \pi_1(\pi_2(e)) = 2 \\ &\quad \quad \text{then } (n, (1, \pi_2(\pi_2(e)))) \\ &\quad \text{case } \pi_1(\pi_2(\pi_2(e))) = 1 \\ &\quad \quad \text{then } (n, (3, \text{check_ID}(\text{parse_seq}(\pi_2(\pi_2(\pi_2(e))) \frown L, S)))) \\ &\quad \text{case } \pi_1(\pi_2(\pi_2(e))) = 2 \\ &\quad \quad \text{then } (n, (5, \text{check_ID}(\text{parse_seq}(\pi_2(\pi_2(\pi_2(e))) \frown L, S)))) \\ &\quad \text{endcase} \\ \text{parse_seq}(L, S) &= \\ &\quad \text{if } L = \perp \\ &\quad \text{then } \perp \\ &\quad \text{elseif } \text{first}(L) = ((1, n'), m, M) \\ &\quad \quad \text{then if } m = 1 \wedge M = (1, 1) \\ &\quad \quad \quad \text{then } [\pi_2(\text{expand_elt_type}(n', S))] \frown \\ &\quad \quad \quad \quad \text{parse_seq}(\text{rest}(L), S) \\ &\quad \quad \quad \text{elseif } m = 0 \wedge M = (1, 1) \\ &\quad \quad \quad \text{then } [(5, [\pi_2(\text{expand_elt_type}(n', S)), \\ &\quad \quad \quad \quad (1, \text{"Empty"})])] \frown \text{parse_seq}(\text{rest}(L), S) \\ &\quad \quad \quad \text{else } [(4, \pi_2(\text{expand_elt_type}(n', S))] \frown \\ &\quad \quad \quad \quad \text{parse_seq}(\text{rest}(L), S) \\ &\quad \quad \quad \text{endif} \\ &\quad \quad \text{elseif } \text{first}(L) = ((2, e), m, M) \\ &\quad \quad \quad \text{then if } m = 1 \wedge M = (1, 1) \\ &\quad \quad \quad \quad \text{then } [\pi_2(\text{expand_elt_type}'(\pi_1(e), (\pi_2(e), \\ &\quad \quad \quad \quad \pi_3(e)), S))] \frown \text{parse_seq}(\text{rest}(L), S) \\ &\quad \quad \quad \quad \text{elseif } m = 0 \wedge M = (1, 1) \\ &\quad \quad \quad \quad \text{then } [(5, [\pi_2(\text{expand_elt_type}'(\pi_1(e), (\pi_2(e), \\ &\quad \quad \quad \quad \pi_3(e)), S)), (1, \text{"Empty"})])] \frown \\ &\quad \quad \quad \quad \quad \text{parse_seq}(\text{rest}(L), S) \\ &\quad \quad \quad \quad \text{else } [(4, \pi_2(\text{expand_elt_type}'(\pi_1(e), (\pi_2(e), \\ &\quad \quad \quad \quad \pi_3(e)), S))] \frown \text{parse_seq}(\text{rest}(L), S) \\ &\quad \quad \quad \quad \text{endif} \\ &\quad \quad \quad \text{elseif } \text{first}(L) = (n', t, u) \\ &\quad \quad \quad \quad \text{then if } t \neq \text{"IDREFS"} \\ &\quad \quad \quad \quad \quad \text{then } [(1, t)] \frown \text{parse_seq}(\text{rest}(L), S) \\ &\quad \quad \quad \quad \quad \text{else } [(4, (1, \text{"ID"})] \frown \text{parse_seq}(\text{rest}(L), S) \\ &\quad \quad \quad \quad \quad \text{endif} \\ &\quad \quad \quad \text{endif} \\ \text{check_ID}(L) &= \\ &\quad \text{if } L = \perp \\ &\quad \text{then } \perp \\ &\quad \text{elseif } \pi_1(\text{first}(L)) = 1 \wedge \pi_2(\text{first}(L)) = \text{"ID"} \\ &\quad \quad \text{then } (3, [\text{first}(L), (3, \text{rest}(L))]) \\ &\quad \quad \text{elseif } \text{first}(\text{check_ID}(\text{rest}(L))) = (1, \text{"ID"}) \\ &\quad \quad \text{then } (3, [(1, \text{"ID"}), (3, [\text{first}(L)] \frown \\ &\quad \quad \quad \pi_2(\text{first}(\text{rest}(\text{check_ID}(\text{rest}(L))))))]) \\ &\quad \quad \text{else } L \\ &\quad \text{endif} \end{aligned}$$

EXAMPLE 4.6 If S represents the schema from Example 2.1, we obtain

```
expand_elt_type("wine", S) =
  ("wine", (3, [(1, "ID"), (3, [(1, "String"),
    (5, [(1, "Integer"), (1, "Empty")]),
    (4, (3, [(1, "String"), (1, "Integer")])]),
    (1, "Decimal"), (1, "ID")]))))
```

and $\text{expand_elt_type}(\text{"name"}, S) = (\text{"w-name"}, (1, \text{"String"}))$ assuming in the latter case that we add some renaming to avoid name-conflicts.

Now that we got the transformation of types, we can define the expansion of paths, i.e. we get an operation

$$\text{expand_elt} : (\text{String}, \text{XSchema}_{\text{rep}}) \rightarrow \text{Operation}_{\text{rep}}$$

such that $\text{expand_elt}(n, S)$ will be a representation of the operation g/n . In order to define the expansion we need both the types for the element n and its parent, and the position at which the type of n appears inside the type of its parent. We then parse through the parent type and determine the operation g/n according to the given position:

```
expand_elt(n, s) =
  parse_type(π2(expand_elt_type(n, S)),
    π2(first(parents(n, π3(S)))),
    π2(expand_elt_type(
      π1(first(parents(n, π3(S))), S)))
parents(n, L) =
  if L = []
  then []
  else list_match(n, π1(first(L)),
    π2(π2(π2(first(L))), 1) ^ parents(n, rest(L)))
  endif
list_match(n, n', L, i) =
  if L = []
  then []
  else match(n, n', π1(first(L)), i)
    ^ list_match(n, n', rest(L), i + 1)
  endif
match(n, n', e, i) =
  if π1(e) = 1 ∧ π2(e) = n
  then [(n', i)]
  elseif π1(e) = 2 ∧ π1(π2(e)) = n
  then [(n', i)]
  else []
  endif
parse_type(t, i, t') =
  if π1(t') = 3 ∧ first(π2(t')) = (1, "ID")
  then (7, [parse_type(t, i, first(rest(π2(t')))], (2, 2))]
  else case t = t'
    then (1, "id")
    case π1(t') = 3
    then (2, i)
    case π1(t') = 4
    then (1, "id")
    case π1(t') = 5
    then (7, [(1, "if_else"), (3, [(7, [(1, "eq"),
      (3, [(2, 1), (1, i)])]), (1, "single"),
      (7, [(1, "empty"), (1, "triv")])])])])
    endcase
  endif
```

EXAMPLE 4.7 If S represents the schema in Example 2.1 we obtain $\text{expand_elt}(\text{"wine"}, S) = (1, \text{"id"})$, and $\text{expand_elt}(\text{"name"}, S) = (7, [(2, 1), (2, 2)])$.

Similarly, we get an operation expand_latt such that $\text{expand_latt}(n, S)$ will be a representation of the operation $g/@n$. We omit the details.

Finally, let us look at the expansion of paths containing some $//\text{name}$. In this case we get an operation

$$\text{expand_elt}^* : (\text{String}, \text{XSchema}_{\text{rep}}) \rightarrow \text{Operation}_{\text{rep}}$$

such that $\text{expand_elt}^*(n, S)$ will be a representation of the operation $g//n$. We already saw the general structure of this operation, when we discussed the basic model of the translation into RTA, so let us now concentrate on the Boolean operations only. The only condition that involves the element name n is φ_1 . So, let

$$\text{expand_bool1} : (\text{String}, \text{XSchema}_{\text{rep}}) \rightarrow \text{Operation}_{\text{rep}}$$

be such that $\text{expand_bool1}(n, S)$ will be a representation of the operation φ_1 associated with n . Thus, we get:

```
expand_bool1(n, S) =
  (7, [(1, "eq"), (3, [(1, "type"),
    (7, [(1, expand_elt_type(n, S)), (1, "triv")])])])
```

The other operations can be obtained analogously.

5 Conclusion

In this paper we addressed the translation of XQuery to a complex value query algebra. The model underlying this algebra is based on a type system that supports constructors for records, lists and unions as well as optionality and references. This captures the rational tree structures represented by XML documents. The query algebra uses operations defined on this type system, in particular structural recursion for lists.

We demonstrated that the basic execution model of XQuery easily gives rise to nested structural recursion. However, the function parameters involved in these operations require complex definitions resulting from information about the schema. These functions can be generated using compile-time linguistic reflection.

An obvious advantage of this approach is type safety. Furthermore, a translation to a simple query algebra enables the support of algebraic query optimization, the easy implementation of the operations and thus the integration with programming languages, the easy extension to other constructors such as sets and multisets in case the order that comes with the list constructor is considered unnecessary or even undesired. In particular, much of the complexity resulting from path expressions is captured in the translation process.

References

- Abiteboul, S., Buneman, P. & Suciu, D. (2000), *Data on the Web: From Relations to Semistructured Data and XML*, Morgan Kaufmann Publishers.
- Abiteboul, S., Quass, D., McHugh, J., Widom, J. & Wiener, J. (1997), 'The LOREL query language for semi-structured data', *International Journal on Digital Libraries* 1(1), 68–88.
- Buneman, P., Davidson, S., Hillebrand, G. & Suciu, D. (1996), A query language and optimization techniques for unstructured data, in 'Proceedings of the 1996 ACM SIGMOD International Conference on Management of Data', Montréal, Canada, pp. 505–516.

- Chen, Y., Davidson, S. B. & Zheng, Y. (2004), BLAS: An efficient XPath processing system, in G. Weikum, A. C. König & S. Deßloch, eds, 'Proceedings of the 2004 ACM SIGMOD-SIGART-SIGACT Symposium on Principles of Database Systems (SIGMOD 2004)', ACM, Paris, France, pp. 47–58.
- DeHaan, D., Toman, D., Consens, M. P. & Özsu, M. T. (2003), A comprehensive XQuery to SQL translation using dynamic interval encoding, in A. Y. Halevy, Z. G. Ives & A. Doan, eds, 'Proceedings of the 2003 ACM SIGMOD-SIGART-SIGACT Symposium on Principles of Database Systems (SIGMOD 2003)', ACM, San Diego, California, USA, pp. 623–634.
- Deutsch, A., Fernandez, M., Florescu, D., Levy, A. & Suci, D. (1999), 'A query language for XML', *Computer Networks* **31**(11-16), 1155–1169.
- Gottlob, G., Koch, C. & Pichler, R. (2003), The complexity XPath query evaluation, in 'Proceedings of the 22nd ACM SIGMOD-SIGART-SIGACT Symposium on Principles of Database Systems (PoDS 2003)', ACM, San Diego, California, USA, pp. 179–190.
- Katz, H., ed. (2003), *XQuery from the Experts – A Guide to the W3C XML Query Language*, Addison-Wesley.
- Kirchberg, M., Schewe, K.-D. & Tretiakov, A. (2003), A multi-level architecture for distributed object bases, in 'Proceedings of the 5th International Conference on Enterprise Information Systems (ICEIS)', Vol. 1, ICEIS Press, pp. 63–70.
- Koch, C. (2005), On the complexity of nonrecursive xquery and functional query languages on complex values, in 'Principles of Database Systems', ACM.
- Lobin, H. (2001), *Informationsmodellierung in XML und SGML*, Springer-Verlag.
- Marx, M. (2004), Conditional XPath, the first order complete XPath dialect, in 'Proceedings of the 23rd ACM SIGMOD-SIGART-SIGACT Symposium on Principles of Database Systems (PoDS 2004)', ACM, Paris, France, pp. 13–22.
- Paparizos, S., Wu, Y., Lakshmanan, L. V. S. & Jagadish, H. V. (2004), Tree logical classes for efficient evaluation of XQuery, in G. Weikum, A. C. König & S. Deßloch, eds, 'Proceedings of the 2004 ACM SIGMOD-SIGART-SIGACT Symposium on Principles of Database Systems (SIGMOD 2004)', ACM, Paris, France, pp. 71–82.
- Ruecker, S. (2000), A conceptual taxonomy of SGML tags, in X. Zhou, J. Fong, X. Jia, Y. Kambayashi & Y. Zhang, eds, 'WISE 2000, Proceedings of the First International Conference on Web Information Systems Engineering, Volume II (Workshops)', IEEE Computer Society, pp. 2–10.
- Schewe, K.-D. (2001), On the unification of query algebras and their extension to rational tree structures, in M. E. Orlowska & J. F. Roddick, eds, 'Database Technologies – Proceedings of the 12th Australasian Database Conference (ADC 2001)', IEEE Computer Society, Gold Coast, Queensland, Australia, pp. 52–59.
- Siméon, J. & Wadler, P. (2003), The essence of XML, in 'Proceedings of the 30th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL 2003)', ACM.
- Stemple, D. W., Fegaras, L., Sheard, T. & Socorro, A. (1990), Exceeding the limits of polymorphism in database programming languages, in F. Bancilhon, C. Thanos & D. Tsichritzis, eds, 'Advances in Database Technology - EDBT'90', Vol. 416 of *LNCS*, Springer-Verlag, pp. 269–285.
- Tannen, V., Buneman, P. & Wong, L. (1992), Naturally embedded query languages, in J. Biskup & R. Hull, eds, 'Database Theory (ICDT 1992)', Vol. 646 of *LNCS*, Springer-Verlag, pp. 140–154.
- Tatarinov, I., Ives, Z., Halevy, A. & Weld, D. (2001), Updating XML, in 'Proceedings of the 2001 ACM SIGMOD International Conference on Management of Data', Santa Barbara, California, pp. 413–424.
- Wadler, P. (1992), 'Comprehending monads.', *Mathematical Structures in Computer Science* **2**(4), 461–493.
- World Wide Web Consortium (W3C) (2001), 'XML Schema', <http://www.w3c.org/TR/xmlschema-0>, <http://www.w3c.org/TR/xmlschema-1>, <http://www.w3c.org/TR/xmlschema-2>.
- World Wide Web Consortium (W3C) (2004), 'XQuery', <http://www.w3c.org/TR/xquery>.