Pattern by Example: type-driven visual programming of XML queries

Véronique Benzaken[1] Giuseppe Castagna[2] Dario Colazzo[1] Cédric Miachon[3]

[1]Université Paris-Sud 11, LRI, Orsay - France [2]CNRS - PPS, Université Paris 7, Paris - France
[3]Courtanet - Paris - France

ABSTRACT

We present Pattern-by-Example (PBE), a graphical language that allows users with little or no knowledge of pattern-matching and functional programming to define complex and optimized queries on XML documents. We demonstrate the key features of PBE by commenting an interactive session and then we present its semantics, by formally defining a translation from PBE graphical queries into ℂQL ones. The advantages of the approach are twofold. First, it generates queries that are provably correct with respect to types: the type of the result is displayed to the user and this constitutes a first and immediate visual check of the semantic correctness of the resulting query. The second advantage is that a semantics formally—thus, unambiguously—defined is an important advancement over some current approaches in which standard usage and learning methods are based on "trial and error" techniques.

Keywords

Visual programming, Database programming languages, Functional programming, Type systems.

1. INTRODUCTION

One of the reasons, if not the main one, of the success of relational databases is the query language SQL. The key features that made SQL *the* standard query language for relational databases are its ease of use, its formal foundation and clear semantics, and its high declarativity. This last point is quite important because both it makes the writing of SQL queries independent from the physical organization of data and, for the same reason, makes SQL queries highly optimizable.

As we discuss in the related work section, a further boost to relational databases was given by the introduction of graphical query languages, such as *Query-by-Example* (QBE). Despite the simplicity of SQL and of the relational model these graphical query languages allowed more persons to access relational databases and in a more user friendly way. This is done without missing most of the advantages of the previous approach since the semantics of these languages is given by a translation into the relational algebra or calculus.

Nowadays there is a clear trend to increasingly use XML to make data available on the Web. Querying data in this format poses the same challenges as for relational data and even amplifies the problems. The arbitrary structural nesting of XML due to its tree-based structure is at the origin of the absence of a clear candidate language to query bases of XML documents. W3C puts forward the XQuery language [4] and other proposals such as XDuce [12] or ℂDuce [1] exist. While XQuery relies on XPath to deconstruct XML trees, and on a `for` operator to iterate over this deconstruction the other rely on pattern-matching for deconstructing values and, in the case of ℂQL, on a`select-from-where` iterator. While XPath is good for a deconstruction that navigates vertically in the document it is not able to perform a fine grained selection on horizontal navigation, that is on sequences of elements. For instance, imagine that we have to select in an XML document `bib.xml` containing a bibliography (see Figure 2 for an instance), all the titles of books published by Addison Wesley after 1991 that have exactly two authors. In XQuery we cannot directly select these titles but we have to stop at books' level, and then perform three subselections one for authors, one for titles and one for prices as in (iterator keywords are underlined)

```
<bib>  for $b  in
document("bib.xml")/bib/book[count(./author)=2]  where
$b/publisher="Addison-Wesley" and $b/@year>1991  return
<book year="$b/@year">$b/title </book>  </bib>
```

It would be better if we could capture in a variable exactly the titles of the books that match the required conditions, that is, that have a specific given form. In functional languages the form of a value can be described by patterns. Patterns then can be used to perform horizontal selection, by matching them against heterogeneous sequences of elements in order to capture only some given subparts. For this reason in a previous work [2] we proposed ℂQL an XML oriented query language that combines the vertical selection capabilities of XPath-like expressions with the horizontal selection capabilities of ℂDuce patterns [1], which are patterns designed for XML elements. In ℂQL the query above is written as

```
<bib>
 select <book year=y> t from
  <book year=y&(1992--*)>[ t::Title Author Author
                      <publisher>"Addison-Wesley" (_\Author)* ]
    in load_xml("bib.xml")/Book
```

ℂQL syntax is an enriched form of the SQL's one: (*i*) in the `select` part we can use fully structured expressions instead of just relations, (*ii*) on the right of a « `in` » in `from` clauses, simple relations (that is, sets of tuples) are replaced by XPath-like expressions that allow vertical navigation to select heterogeneous sequences of elements and (*iii*) rather than simply captured by variables (as in SQL) the extracted sequence is navigated horizontally by patterns that match the sequence elements and capture subparts in variables. In the expression above the pattern on the left of the « `in` » keyword selects all and only the book elements whose attribute year is in the interval $(1992, \infty)$ and that have *exactly* two author subelements followed by a publisher element that contain the string `"Addison-Wesley"`, this followed by any element (the wild-card "_") that is not (the difference sign \) an author (the * denotes a regular expression that indicates that there may be zero or more such elements);[1] of the selected book elements the pattern captures the year in the variable `y` and the title in the variable `t`.

ℂQL not only makes it possible to combine vertical and horizontal navigation but provides a very precise type inference and better logical optimizations which make it more efficient in main memory execution than major implementations of XQuery [2]. However, the use of patterns may be difficult to a basic programmer, especially in advanced (e.g. nested) queries. In this context a graphical interface to define queries is much more necessary than in the SQL case. This is the goal of our work that, mimicking what was done for SQL, will first define a tableau-based graphical representation of queries for XML-documents and then give its semantics via a translation into ℂQL. The rich structure of XML makes the task much more challenging than for the relational model: we do not work on a set of fixed and flat relations; instead the information we extract may have a complex structure. In order to generate the table corresponding to some extracted data our system will heavily rely on the type system. For instance in the query example we gave above, once we have extracted the data on books the graphical interface will use the type system and the given DTD to generate a table that contains a column for the year, another for the authors, a third for the publisher and a last one for the price: the users will then have just to fill the cells with the corresponding conditions and capture variables to complete the query.

Related work

The use of graphical languages for expressing queries is not new in the database field. This is mainly due to the requirement that non-expert users should be able to interact with the database system while not being acquainted with the subtleties of the underlying query language which may be complex to use.

Query-by-Example (QBE) [16] is the first graphical query language for relational databases. It has been developed in the 70's by Zloof at IBM and gave rise to a wide category of commercial graphical languages such as, for example, Paradox or Microsoft Access. The central concept of QBE is the notion of tableaux. A tableau is a graphical interface (a table indeed) allowing the user to express some queries simply by defining specific variables in the table.

In the context of XML, many attempts to define graphical query languages have been proposed: QSByE (Querying Semi-structured data by Example) [11], XQBE [6], Miroweb [5], EQUIX [9], BBQ [14], Pesto [7], QURSED [15], Xcerpt [3] and Xing [10]. Due to space limitations, we shall give the spirit of these approaches rather than giving an exhaustive state of the art. Hence, we choose to present XQBE *XQuery by Example* as it is the most complete language. We refer the reader to [13] for a complete survey.

Unlike QBE, rather than manipulating tableaux, XQBE manipulates XML trees. The purpose was to offer an intuitive interface in order to automatically generate XQuery queries. XQBE offers most of XPath expressive power,[2] permits the definition of nested queries, to build new elements etc. In order to give the reader a flavor of XQBE let us consider the following query which corresponds to query Q_1 of *XML Query Use Cases* [8]. List all books published by "Addison-Wesley" since 1991. This is exactly the query we presented in the introduction without the condition on the number of authors. Thus to define it it suffices to remove in the XQuery expression the predicate on the path. In XQBE such a query is expressed as shown in Figure 1.

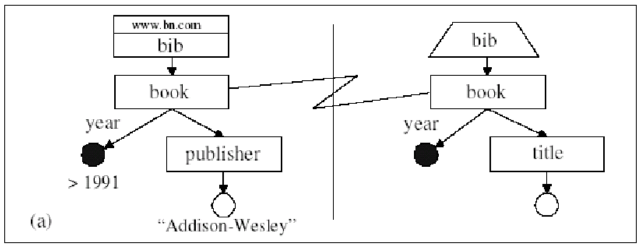


Figure 1: XQBE Q_1

In XQBE, the workspace is divided in two separate zones: the source space (on the left) and the result space (on the right). Each zone contains labeled graphs which represent fragments of the XML document to be processed. XML elements are represented by rectangles annotated by their tag, attributes are represented by black disks together with their names. For instance, on Figure 1 the source zone expresses a query which extracts all books elements `<book>` having an attribute *year* whose value is greater than 1991, and having a child `<publisher>` with value `"Addison-Wesley"`. In the corresponding result space, again the result is described by a tree. For our example, the graph states that the result will consist of all the titles of `<book>` elements which have been selected in the source space (such a binding is materialized by the arc connecting the respective node from source to result space). These will then be encapsulated in a unique fresh element `<bib>` (the trapezoidal shape indicates the fact that the result is considered as new).

Most of graphical query languages for XML use graph-based representations of both documents and queries. Their main limitations are that no semantics is formally assigned to those graphs hence they do not account for correctness proofs of the translation (usually to XQuery) they implement and last they never exploit the underlying type system in order to yield optimized versions of the resulting queries.

Unlike those, (i) we formally assign a semantics to our graphical tableaux-based interface and (ii) formally establish a (partial) correctness proof of the translation to ℂQL.

We will proceed as follows. First we present in Section 2 the system by showing and commenting an interactive session with our prototype. To that end we also introduce ℂQL, since its regular expression types are used as conditions in the graphical interface whose use will result in the generation of a ℂQL expression. The formal development follows in Section 3. In particular we formally introduce the notions of tableau and PBE query and define their semantics by translating PBE queries into ℂQL queries. Since the translation in far from being trivial we define the translation incrementally, by progressively increasing the complexity of

[1]The difference sign is used for the sake of the example but here is completely useless. The DTD of "`bib.xml`" given in Section 2.1 ensures that a publisher element is followed just and exactly by one element of type `Price`. Therefore a single wildcard "_" would have sufficed.

[2]Apart from some functions such as for instance `position()`

the translated queries. This will allow us to point out the most difficult or subtle points of the translation. A partial correctness result of this translation is also given.

Throughout the presentation we use some conventions and syntactic sugar of ℂDuce/ℂQL, most of which are quite intuitive and need no explanation. On the same vein, we just present a very simplified version of the language. Space constraints do not allow us to do a complete treatment, which anyhow would not bring any further insight. The interested reader can consult the documentation available on the ℂDuce web site (`www.cduce.org`) and try the distribution of the full featured language available there too.

2. A GUIDED TOUR

In this section we present a guided tour of PBE (*Pattern by Example*) our graphical query language designed to help non-expert users to write complex queries. PBE uses ℂQL as a back-end since it generates and evaluates optimized ℂQL queries, but other back-ends can be considered. Actually, PBE can be used independently from ℂQL, since its usage only requires the knowledge of the types that ℂQL borrow from ℂDuce, types that are very close to other type systems for XML. However, the presentation of PBE semantics is far simpler in ℂQL, which is the reason why we start this presentation by an overview of ℂQL.

2.1 Presentation of ℂQL

The goal is not to give a full presentation of ℂQL (for that see [2]) but rather to present a minimum set of features that are enough to present PBE. The most important feature are types. PBE and ℂQL use ℂDuce's types, which can be seen as a compact notation for DTDs (actually, for Relax-NG):

$$\begin{array}{lll}
\textbf{Types} & T & ::= \; \mathit{btype} \mid \texttt{[}t\texttt{]} \mid \texttt{<}\mathit{tag}\ \texttt{\{}A\texttt{\}>[}t\texttt{]} \mid \texttt{Any} \mid v \\
 & & \quad\;\; \mid \; T\texttt{|}T \mid T\texttt{\&}T \mid T\setminus T \\
\textbf{RegEx} & t & ::= \; T \mid t\ t \mid t\texttt{|}t \mid t\texttt{?} \mid t\texttt{*} \mid t\texttt{+} \mid \varepsilon \\
\textbf{Attrs} & A & ::= \; a\texttt{=}T A \mid \varepsilon
\end{array}$$

Types are either *type constructors*, that is: basic types (e.g., `Int`, `Bool`, `Char`, ...); heterogeneous sequences types (delimited by square brackets and whose content is described by a type regular expression t); XML elements (that is, tagged sequences whose tag may contain a possibly empty list of attribute type declarations which assign types to attribute names—ranged over by a—); `Any`, the type of all values; v, the singleton type that contains only the value v. Or they are *type combinators*, that is, union, intersection, or difference of types. Regular expression types, ranged over by t, are obtained from types and the empty string (denoted by ε) by juxtaposition, union, and the constructors for optional elements, possibly empty, and nonempty sequences.

We will use some conventions, in particular the underscore "`_`" to denote `Any`, `PCDATA` to denote the regular expression type `Char*`, and `String` to denote the type `[Char*]`. We also use identifiers to denote types (and follow the convention of capitalizing them), as in the following declarations

```
type Bib = <bib>[Book*]
type Book = <book year=String>[Title (Author|Edit)+ Publisher Price]
type Author = <author>[Last First]
type Edit = <editor>[Last First]
type Title = <title>[PCDATA]
type First = <first>[PCDATA]
type Last = <last>[PCDATA]
type Publisher = <publisher>[PCDATA]
type Price = <price>[Int]
```

which defines the types for the bibliography example we will use throughout the paper.

For this paper, ℂQL expressions are variables (ranged over by x, y, ...), constants (e.g. `true`, `1`, `2`, ... ranged over by c), the `select_from_where` expression, the constructors for sequences (a juxtaposition of blank-separated expressions delimited by square brackets), and XML elements (a sequence expression e labeled by a tag and a possibly empty set of attributes), banged expressions `!`e (which "opens" the sequence e so that, for instance, if $e_1, e_2, \ldots, e_n$ are sequences, then `[!`e_1 `!`e_2 ... `!`e_n`]` returns their concatenation), and operators (e.g. `=`, `>`, `max`, `if_then_else`, ...). Values, ranged over by v, are closed expressions that do not contain "select", operators, or banged sub-expressions.

$$\begin{array}{lll}
e & ::= & x \mid c \mid \texttt{[}e\ldots e\texttt{]} \mid \texttt{<}\mathit{tag}\ a\texttt{=}e\ldots a\texttt{=}e\texttt{>}e \mid \texttt{!}e \mid \mathit{op}(e,..,e) \\
 & \mid & \texttt{select}\ e\ \texttt{from}\ p\ \texttt{in}\ e\texttt{,}\ldots\texttt{,}p\ \texttt{in}\ e\ \texttt{where}\ e
\end{array}$$

The expression `select` e^s `from` p_1 `in` e_1,...,p_n `in` e_n `where` e^w deserves explanation. The expression e^w in the `where` clause must be of boolean type, while the expressions e_i's in the `from` clauses must return sequences. Select iterates on these sequences matching each element of e_i against the corresponding pattern p_i. Pattern variables capture subparts of the matching elements and these variables can then be used in e^s or in the successive `from` clauses. The result of a `select` is the sequence of evaluations of the expression e^s in the environments obtained by iterating on the `from` clauses.

Patterns are nothing but types with capture variables. We distinguish two kinds of patterns for capture variables: "simple variables patterns" that have the form of a variable and can occur wherever a type can, and "sequence capture patterns" that have the form x`::`t, can occur wherever a regular expression type can, and capture in x the *sequence* of all values matched by the regular expression t. So in the ℂQL query given in the introduction `y` is a simple capture variable (the intersection of two patterns succeeds only if each pattern succeeds, therefore `y` captures the value of attribute `year` only if this is of type `1992--*`), while `t` captures the sequence of all titles of the book (in this case just one). Differently from union types, that are symmetric, union patterns implement a first match policy: the right pattern is checked only if the left one fails. So, for instance when the pattern `[(x::Author|_)*]` is matched against a sequence it captures in `x` the sequence of all (values of type) authors present in it (if an element is of type `Author`, then it is captured by `x`, otherwise is discarded by matching it against the wildcard "`_`"—i.e. the type `Any`).

We apply the convention to use single quotes to delimit characters and double quotes to delimit strings (which are sequences of characters). For formal and complete definitions of the syntax, the semantics, and the typing of ℂQL the reader can refer to [2].

2.2 A tour of PBE

We demonstrate PBE by querying the document in Figure 2 and assuming that it conforms to the ℂDuce type `Bib` defined by the declarations given in the previous section (from which we omit `Edit` in order to limit the size of figures) that we will have entered in the tab "Data" of our PBE interface, visible in Figures 3–11. [3]

Queries are expressed by means of *tableaux*. Two different kinds of tableaux are presented: *Filter tableaux* and *Con-*

[3]Declarations are generated from a DTD by the program `dtd2cduce`.

```
<bib>
   <book year="1995">
       <title>TCP/IP Illustrated</title>
       <author>
          <last>Stevens</last>
          <first>W.</first>
       </author>
       <publisher>Addison-Wesley</publisher>
       <price>65</price>
   </book>
   <book year="1992">
        <title>Advanced Programming in Unix</title>
        <author>
          <last>Stevens</last>
          <first>W.</first>
        </author>
        <publisher>Addison-Wesley</publisher>
        <price>65</price>
   </book>
   <book year="2000">
        <title>Data on the Web</title>
        <author>
          <last>Abiteboul</last>
          <first>Serge</first>
        </author>
        <author>
          <last>Buneman</last>
          <first>Peter</first>
        </author>
        <author>
          <last>Suciu</last>
          <first>Dan</first>
        </author>
        <publisher>Morgan Kaufmann Publishers </publisher>
        <price>39</price>
   </book>
</bib>
```

Figure 2: reference XML document

struct tableaux. The former are used for extracting information (they are entered in the upper half of the interface), while the latter are used for building the sequence of XML values that constitutes the result of the query (they are entered in the lower half of the interface). PBE tableaux allow for expressing a wide variety of queries. Let us start with a simple query: “return all books in the bibliography”. Assume that the document to be queried is stored in the `doc` (persistent) variable. The filter tableau offers a list of persistent XML documents and the user will choose among them the `doc` variable as shown in the left part of Figure 3.

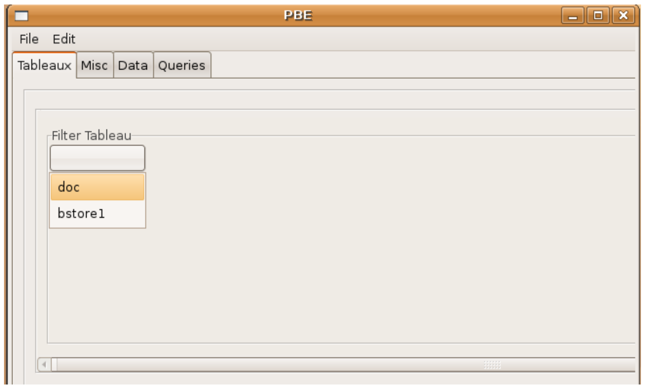


Figure 3: Filter tableau creation

Once the document is selected, PBE displays the filter tableau associated to the type of `doc` (i.e., `Bib`) as shown in Figure 4. The column marked by a `#` symbol represents the tag which can be tested and captured[4] while the fact that the content of `Bib` elements is a sequence of `Book` elements (recall, `Bib = <bib>[Book*]`) is represented by `Book*`. In the row, PBE provides fresh variables `x1`, `x2` to capture the corresponding components and a default (type) constraint `Any` which is always satisfied

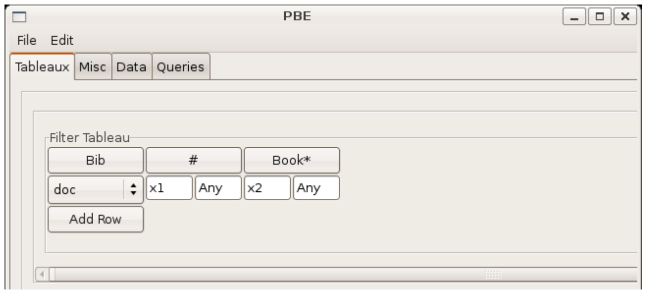


Figure 4: Filter tableau for `doc`

The user who wants to capture all the books of the bibliography `doc` in a variable `books` (Figure 5), has just to declare this variable in the corresponding column (the one labeled by `Book*`). The right part of the cell remains unchanged (`Any`), since we do not need to express further constraints on variable `books`.

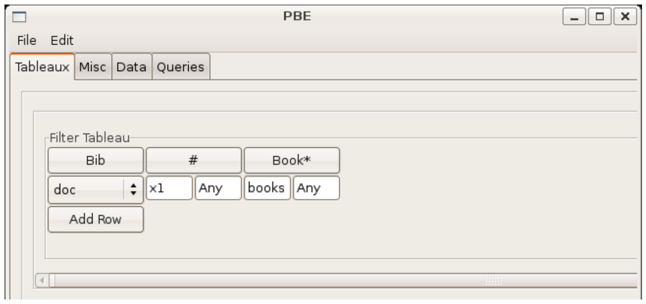
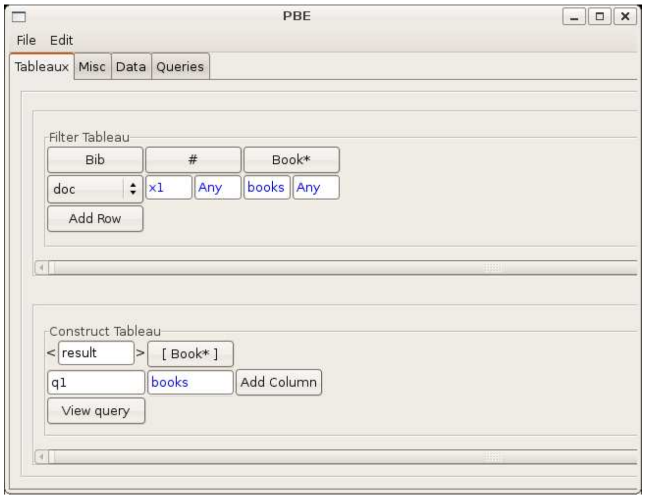


Figure 5: Adding variable `books` in the filter tableau

Getting and, presumably, re-structuring the result is performed by means of a *construct tableau* that is defined in the lower part of the window as illustrated in Figure 6. Construct tableaux are defined by adding new columns and filling the cells by using the variables introduced in the other tableaux. From the content that is filled in a cell, PBE deduces and inserts the type that labels the corresponding column. Not only does the construct tableau indicates how the result is re-structured (here we choose to encapsulate all books in a `<result>` tag) but it also provides a fresh variable `q1` that denotes the query so that it can be later reused (e.g. for defining nested queries).
Clicking on the “View query” button right below a construct tableau, makes PBE compute and display in the “Queries” tab the corresponding ℂQL query and its result (Figure 7). PBE also infers that the type of `q1` is `[<result>[Book*]*]`, an information useful in case `q1` was reused in other queries. As with any other variable, `q1` can be reused by selecting it in the pull down menu of Figure 3 to which it is automatically added at the moment of its definition.

This first example was very simple. We shall now present two more advanced examples that illustrate (*i*) how to program nested queries and (*ii*) what is the use of several rows

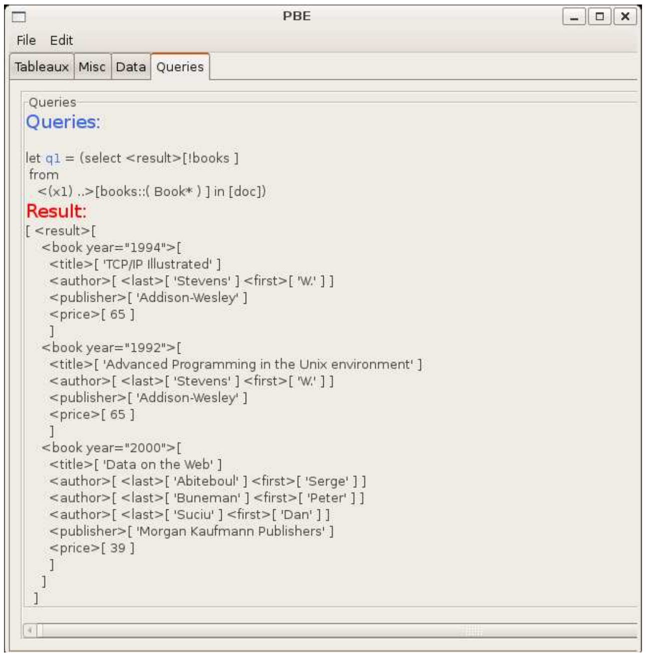
[4]In the full version of ℂQL/ℂDuce XML tags are full fledged expressions that can contain namespaces and have arbitrary complex types such as `type AorB = <(‘a|‘b)>[Any*]`.



Bib	#		Book*	
doc	x1	Any	books	Any

< result >	[Book*]
q1	books

Figure 6: Construct tableau creation for q1



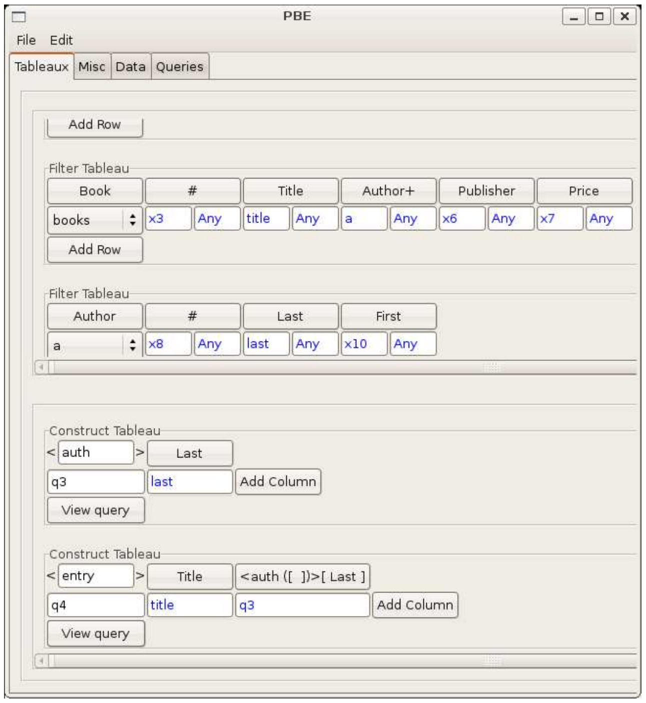
```
Queries:

let q1 = (select <result>[!books ]
 from
  <(x1) ..>[books::( Book* ) ] in [doc])
Result:
[ <result>[
   <book year="1994">[
    <title>[ 'TCP/IP Illustrated' ]
    <author>[ <last>[ 'Stevens' ] <first>[ 'W.' ] ]
    <publisher>[ 'Addison-Wesley' ]
    <price>[ 65 ]
    ]
   <book year="1992">[
    <title>[ 'Advanced Programming in the Unix environment' ]
    <author>[ <last>[ 'Stevens' ] <first>[ 'W.' ] ]
    <publisher>[ 'Addison-Wesley' ]
    <price>[ 65 ]
    ]
   <book year="2000">[
    <title>[ 'Data on the Web' ]
    <author>[ <last>[ 'Abiteboul' ] <first>[ 'Serge' ] ]
    <author>[ <last>[ 'Buneman' ] <first>[ 'Peter' ] ]
    <author>[ <last>[ 'Suciu' ] <first>[ 'Dan' ] ]
    <publisher>[ 'Morgan Kaufmann Publishers' ]
    <price>[ 39 ]
    ]
  ]
 ]
```

Figure 7: ℂQL code and result for q1.

in a filter tableau. Imagine that we want to define a query that returns a sequence of elements tagged by `<entry>` where each such element corresponds to a book of our example bibliography and contains its title element as well as the authors' last name elements encapsulated in a `<auth>` tag. While the plain English semantics is a little bit twisted, the meaning should be quite clearer by looking at how the query is expressed in Figure 8.

The first filter tableau is defined for the `books` variable that was introduced (and automatically added in the pull-down menu) by the previous query, and extracts in `title` and `a` the list of titles (well, just one) and of authors of each book, respectively. This row captures for each book the relationship between its title and its authors. In order to extract for each author in `a` his/her last-name we use a second filter tableau which captures in the variable `last` the corresponding information. To encapsulate each `<last>` element in a tag `<auth>`, we define the construct tableau `q3`.



Book	#		Title		Author+		Publisher		Price	
books	x3	Any	title	Any	a	Any	x6	Any	x7	Any

Author	#		Last		First	
a	x8	Any	last	Any	x10	Any

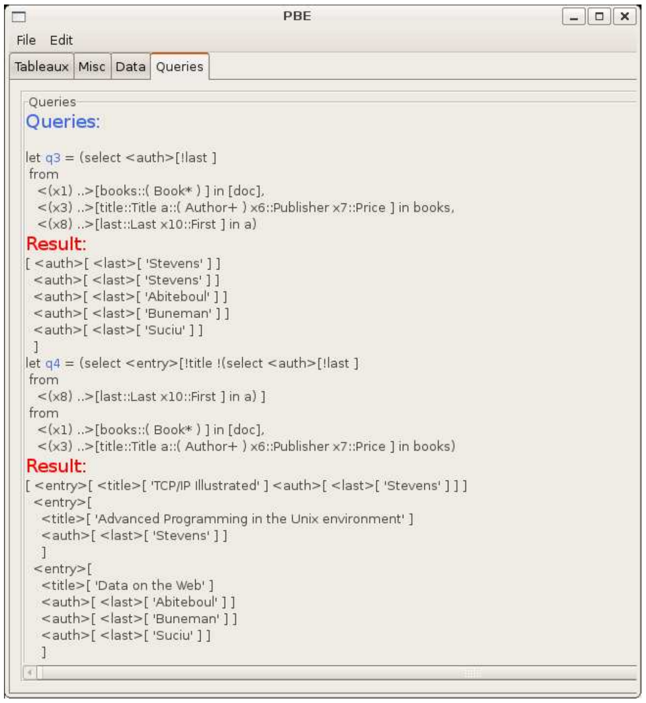
< auth >	Last
q3	last

< entry >	Title	<auth ([ ])>[Last]
q4	title	q3

Figure 8: A nested PBE query

This tableau is then reused in the construct tableau of the query `q4`, in which the title is requested as well as the result of `q3` for this title.

The definitions of the queries `q3` and `q4` and their respective results are shown in Figure 9. When it is executed

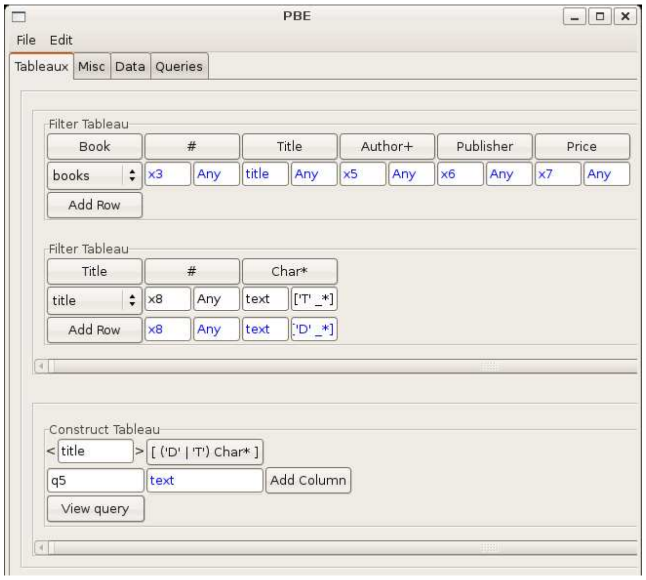


```
Queries:

let q3 = (select <auth>[!last ]
 from
  <(x1) ..>[books::( Book* ) ] in [doc],
  <(x3) ..>[title::Title a::( Author+ ) x6::Publisher x7::Price ] in books,
  <(x8) ..>[last::Last x10::First ] in a)
Result:
[ <auth>[ <last>[ 'Stevens' ] ]
  <auth>[ <last>[ 'Stevens' ] ]
  <auth>[ <last>[ 'Abiteboul' ] ]
  <auth>[ <last>[ 'Buneman' ] ]
  <auth>[ <last>[ 'Suciu' ] ]
  ]
let q4 = (select <entry>[!title !(select <auth>[!last ]
 from
  <(x8) ..>[last::Last x10::First ] in a) ]
 from
  <(x1) ..>[books::( Book* ) ] in [doc],
  <(x3) ..>[title::Title a::( Author+ ) x6::Publisher x7::Price ] in books)
Result:
[ <entry>[ <title>[ 'TCP/IP Illustrated' ] <auth>[ <last>[ 'Stevens' ] ] ]
  <entry>[
   <title>[ 'Advanced Programming in the Unix environment' ]
   <auth>[ <last>[ 'Stevens' ] ]
   ]
  <entry>[
   <title>[ 'Data on the Web' ]
   <auth>[ <last>[ 'Abiteboul' ] ]
   <auth>[ <last>[ 'Buneman' ] ]
   <auth>[ <last>[ 'Suciu' ] ]
   ]
```

Figure 9: ℂQL code for queries q3 and q4

standalone `q3` returns a single list containing all the authors in the bibliography (since in that case `a` is bound to all authors), as shown in the first « Result » section of Figure 9. Instead when it used inside `q4` the query `q3` encapsulates the authors of the book currently selected by the outer iteration. It is important to notice that q3 does not occur in the code for q4. As a matter of fact, it would be wrong to do it, as the code that occurs in q4 at the position of q3 is not the code defined for q3 as a stand-alone query. Indeed when generating the code q4 PBE must generate custom code for the call of q3, that takes into account the environment in which the nested query is evaluated. The technique we use to keep track of the environment in which nested queries are called and to minimize the number of possible patterns needed for expressing the query are formally explained from Section 3.2.3 on.



Filter Tableau

Book	#		Title		Author+		Publisher		Price	
books	x3	Any	title	Any	x5	Any	x6	Any	x7	Any

Filter Tableau

Title	#		Char*	
title	x8	Any	text	['T' _*]
	x8	Any	text	['D' _*]

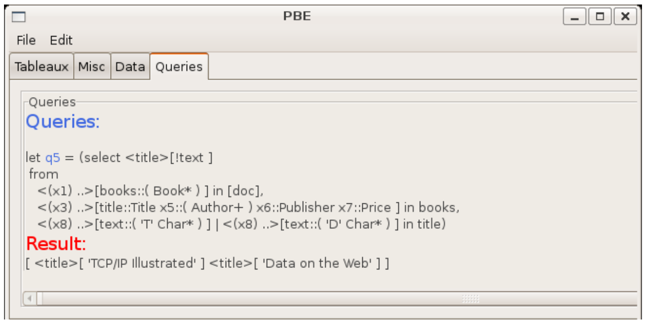
Construct Tableau

<title> [('D' | 'T') Char*]

q5 text

Figure 10: Multiple rows tableau

Our last example illustrates the use of several rows in a filter tableau. Assume that we want to select the books whose title begins either by letter "T" or by letter "D".These constraints are expressed in the ℂQL type algebra respectively as `['T' _*]`, `['D' _*]`. Their "or" is obtained by the tableaux in Figure 10, since in PBE multiple rows are interpreted as union patterns. Note that each row declares the same variables: rows differ only for their constraints (see also Definition 3.2 which enforces this property). It is worth stressing that by using the knowledge of the DTD and the stated constraints of the filter tableau, PBE deduces type : `['D'|'T' Char*]` for the capture variable `text` in the construct tableau. The ℂQL query generated by the system and its result are given on Figure 11.



Queries:

```
let q5 = (select <title>[!text ]
 from
   <(x1) ..>[books::( Book* ) ] in [doc],
   <(x3) ..>[title::Title x5::( Author+ ) x6::Publisher x7::Price ] in books,
   <(x8) ..>[text::( 'T' Char* ) ] | <(x8) ..>[text::( 'D' Char* ) ] in title)
```

Result:

```
[ <title>[ 'TCP/IP Illustrated' ] <title>[ 'Data on the Web' ] ]
```

Figure 11: Result of the multi-row query

3. FORMAL DEVELOPMENT

In this section we give the the formal definition of PBE by first precisely defining its syntax and then stating its semantics via a translation into ℂQL.

3.1 PBE syntax

The syntax of PBE is constituted by three distinct kinds of tableaux, *filter tableaux* and *construct tableaux* that were informally presented in the previous section, and *condition tableaux* (or *condition boxes*). Let us discuss each of them.

3.1.1 Filter tableaux

Filter tableaux are tables in which (i) rows are labeled by already defined variables, (ii) columns are labeled by attribute names, by a hash sign (exactly one column), and/or by type regular expressions and (iii) cells contain fresh variables and regular expression type constraints. For instance, in the previous section we defined the following tableau

Book	#	Title	Author+	Publisher	Price
books	(x_1,t_1)	(x_2,t_2)	(x_3,t_3)	(x_4,t_4)	(x_5,t_5)

which filters the elements that compose the sequence denoted by the variable **books**. The user defines only the content of the row, the rest (that is the number of columns and their labels) are automatically deduced from the type of filtered variable **books**, that is `Book`. But how is that PBE decided to insert a single column labeled `Author+` instead of—equivalently—, say, three columns respectively labeled `Author?`, `Author`, `Author*`? The reason to prefer the former to the latter should be pretty clear: we want to minimize the number of filter columns in order to use as few variables as possible. In order to formalize the way in which this choice is made, we need the definition of *sequence maximal product.*

First notice that every type regular expression t is of the form $R_1R_2\ldots R_n$ (with $n{\geq}1$) where R_i's are type regular expressions different from the juxtaposition. Let us call $R_1\ldots R_n$ the *expanded form* of t. Notice also that every R_i in an expanded form is of the form $t_R\circ$ (where $\circ$ is either `*`, `+`, `?`, or the empty string—in the latter case t_R is either a regular expression union or a type): we call t_R the *base* of R. Finally, we write $T_1 \simeq T_2$ if and only if T_1 and T_2 denote the same type (e.g. `[(A|B) C]`$\simeq$`[(A C)|(B C)]`; see [1] for definition).

Definition 3.1. *Let $R_1\ldots R_n$ be a type regular expression in its expanded form and let us denote the base of R_i by t_{R_i}. $R_1\ldots R_n$ is a* sequence maximal product *if* `[`t_{R_i}`]` $\not\simeq$ `[`$t_{R_{i+1}}$`]` *for* $i = 1...(n{-}1)$.

For example, « `B* B+ C B` » is not a maximal product since the first two elements have the same base. There exists a naive algorithm to transform every type regular expression into a maximal product and consisting in merging consecutive expressions with the same base (e.g., « t`*` t » becomes t`+` and « `B* B+ C B` » becomes « `B+ C B` »). Therefore, henceforward we consider all type regular expressions be maximal products. Notice, however, that this is just a syntactic property with no semantic implication. It heavily depends on way the user wrote DTD's for data: for instance, « `(A|B)* (A*C+|B*C+)` » is a maximal product although « `(A|B)* C+` » would be a smarter denotation.

Definition 3.2. *Let T be an XML type, a* filter tableau *associated to T is:*

T	#	a_1	$\cdots$	a_k	R_1	$\cdots$	R_n
y	(x_0, t_0^1)	(x_1, t_1^1)	$\cdots$	(x_k, t_k^1)	(x_{k+1}, t_{k+1}^1)	$\cdots$	(x_{k+n}, t_{k+n}^1)
$\vdots$	$\vdots$	$\vdots$		$\vdots$	$\vdots$		$\vdots$
y	(x_0, t_0^m)	(x_1, t_1^m)	$\cdots$	(x_k, t_k^m)	(x_{k+1}, t_{k+1}^m)	$\cdots$	(x_{k+n}, t_{k+n}^m)

where

1. *y is a variable of type* `[`T`*]` *or a persistent root of type T,*
2. *T =* `<`*tag* `{`a_1`=`$T_1 \dots a_k$`=`T_k`}>[`$R_1 \ \dots \ R_n$`]`*,*
3. *$R_1 \ \dots \ R_n$ is a maximal product,*
4. *x_j are fresh variables ($j = 0 \dots k+n$),*
5. *t_j^i are regular expression types ($i = 1..m, j = 0..k+n$).*

Henceforth we will mainly work on what we call (improperly in the case of filter tables) rows of a tableau and we use the following compact notation to denote the (set of) row(s) of a filter tableau

$$FT(y|tag|k|(x_0, \vec{t_0})|(x_1, \vec{t_1}) \dots (x_k, \vec{t_k})|(x_{k+1}, \vec{t}_{k+1}) \dots (x_{k+n}, \vec{t}_{k+n}))$$

where *tag* is the tag of the XML type associated to y, k the number of its attributes and each $\vec{t_i}$ represent the vector $t_i^1, \dots, t_i^m$

3.1.2 Construct tableaux

A *construct tableau* is a single row table that defines the structure of the result of a query. The user specifies the tag in which the result must be encapsulated and adds as many columns as (subsequences of) elements in the result. Each element is specified by filling the cell in the corresponding column with a variable whose type will determine the label of the column. For instance, the construct tableau of Figure 10 is:

`<title>`	`[ ('D'|'T') Char* ]`
q5	*text*

In general, users can define not only the tag of the result but also its attributes, which yields the definition:

Definition 3.3. *If $x_1, ..., x_{k+n}$ are variables, $a_1, ..., a_k$ are attribute names and tag is an expression denoting a tag, then they define the following* construct tableau

tag	a_1	$\cdots$	a_k	R_1	$\cdots$	R_n
y	x_1	$\cdots$	x_k	x_{k+1}	$\cdots$	x_{k+n}

where R_i is the (regexp) type of x_{k+i} ($i = 1 \dots n-k$) and y a fresh variable of type `[(<`*tag* `{`a_1`=`$t_1 \dots a_k$`=`t_k`}>[`$R_1 \dots R_n$`])*]`.

As we did for filter tableaux we introduce a compact notation to denote a row of construct tableau, that is

$$CT(y|tag|k|(a_1, x_1) \dots (a_k, x_k)|x_{k+1} \dots x_{k+n}),$$

where k is the number of attributes.

3.1.3 Condition Box

PBE condition boxes are the same as in QBE, that is, they are used to specify constraints. In particular, condition boxes are useful for declaring join conditions between two variables. Condition boxes are of the form as shown on the side, that is they are single column tables whose rows contain a ℂQL expression of boolean type. Usually these expressions are applications of operators to variables,

`CONDITION BOX`
e_1
$\vdots$
e_n

such as the equality of two variables x=y (a typical condition used for joins) or to a variable and constants, such as y>5. As we did for filter and construct tableaux we introduce some special notation to record rows of condition boxes. For the sake of the presentation we consider just a very special case of conditions formed by the application of a binary boolean operator to either variables or values. Then a row of a condition box containing expression $e_1 \, \mathsf{op} \, e_2$ will be represented as $CB(\mathsf{op}, e_1, e_2)$.

3.1.4 PBE Queries

Definition 3.4. *A* PBE query *is defined by a non-empty set of persistent roots, a finite set of filter tableaux, a finite non-empty set of construct tableaux, and an optional condition box.*

In order to be well defined every free variable used in a query must be either a persistent root or defined elsewhere. Notice also that in the result of a query (i.e. in a construct tableau) we do not let the user specify general expressions but just variables (it is a design choice); therefore we also require that no persistent root appears free in a construct tableau, since this would be the same as specifying a constant. In order to formally state when a PBE query is correctly defined we need to introduce the notions of free and declared variables of a tableau

Definition 3.5. *Let f, c, and d denote the following three generic objects: $f = FT(y|tag|k|(x_0, \vec{t_0})|(x_1, \vec{t_1}) \dots (x_k, \vec{t_k})|$ $(x_{k+1}, \vec{t}_{k+1}) \dots (x_{k+n}, \vec{t}_{k+n}))$, $c = CT(y|tag|k|(a_1, x_1) \dots$ $(a_k, x_k)|x_{k+1} \dots x_{k+n})$, and $d = CB(\mathsf{op}, e_1, e_2)$. The free and declared variables of these objects respectively are*

$\mathsf{fv}(f)$	$=$	$\{y\}$	$\mathsf{dv}(f)$	$=$	$\{x_0 \dots x_{k+n}\}$
$\mathsf{fv}(c)$	$=$	$\{x_1 \dots x_{k+n}\}$	$\mathsf{dv}(c)$	$=$	$\{y\}$
$\mathsf{fv}(d)$	$=$	$\mathsf{var}(e_1) \cup \mathsf{var}(e_2)$	$\mathsf{dv}(d)$	$=$	$\varnothing$

where var *denotes the function that returns the free variables of a ℂQL expression.*

If $\mathscr{O}$ is a set of objects, then we denote by $\mathsf{fv}(\mathscr{O})$ and $\mathsf{dv}(\mathscr{O})$ the union of the respective sets of free and declared variables of its objects.

Definition 3.6. *For a given PBE query let us denote by $\mathscr{P}$ the set of its persistent roots, by $\mathscr{F}$ the set of all rows of its filter tableaux, by $\mathscr{C}$ the set of all rows of its condition tableaux and by Θ the rows of a possible condition box. The query is* well defined *if and only if*

1. $\mathsf{fv}(\mathscr{F}) \cup \mathsf{fv}(\mathscr{C}) \cup \mathsf{fv}(\Theta) \subseteq \mathsf{dv}(\mathscr{F}) \cup \mathsf{dv}(\mathscr{C}) \cup \mathscr{P}$
2. $\mathsf{fv}(\mathscr{C}) \cap \mathscr{P} = \varnothing$

Note that the freshness conditions in tableaux definitions ensure that every variable is declared in one and only one tableau row that it univocally identifies.

3.2 Semantics

The semantics of PBE is defined via an (effective) translation from PBE queries (more precisely, from variables denoting PBE queries) to ℂQL queries. The translation is defined in form of inference rules. For the sake of presentation, the translation is introduced gradually in several steps: first, we define a naive translation for unnested queries without condition box. Then, we observe that the definition creates some redundancies and modify the translation to avoid them. Next we add nested queries, that is, PBE queries with several interrelated construct tableaux and, finally, the condition box.

3.2.1 Unnested queries without condition

Let $\mathscr{P}$, $\mathscr{F}$, $\mathscr{C}$, and Θ be defined as in Definition 3.6. We start by considering the case in which both Θ and $\mathsf{fv}(\mathscr{C}) \cap \mathsf{dv}(\mathscr{C})$ are empty (no condition and no nesting).

$$\frac{\begin{array}{c} CT(x|tag|k|(a_1,x_1)\dots(a_k,x_k)|x_{k+1}\dots x_{k+n}) \in \mathscr{C} \\ \mathscr{F} \vdash_f x_i \to l_i \quad i=1\dots k{+}n \end{array}}{\begin{array}{l} \mathscr{F},\mathscr{C} \vdash_s x \to \texttt{select <}tag\ a_1\texttt{=}x_1\ \dots\ a_k\texttt{=}x_k\texttt{>} \\ \qquad\qquad \texttt{[!}x_{k+1}\dots\texttt{!}x_{k+n}\texttt{] from } l_1,\dots,l_{k+n} \end{array}} \ (R2)$$

$$\frac{\exists f \in \mathscr{F}, x \in \mathsf{dv}(f) \quad y \in \mathsf{fv}(f) \cap \mathscr{P}}{\mathscr{F} \vdash_f x \to \mathsf{pattern}(f)\ \texttt{in [}y\texttt{]}} \ (F3)$$

$$\frac{\exists f{\in}\mathscr{F},\ x{\in}\mathsf{dv}(f)\ y{\in}\mathsf{fv}(f)\ y{\notin}\mathscr{P} \quad \mathscr{F}\backslash f \vdash_f y \to l}{\mathscr{F} \vdash_f x \to l\ \ \texttt{,}\ \ \mathsf{pattern}(f)\ \texttt{in}\ y} \ (F4)$$

$$\frac{x \notin \mathsf{dv}(\mathscr{C})}{\mathscr{F},\mathscr{C} \vdash_s x \to \Omega} \ (R6) \qquad \frac{x \notin \mathsf{dv}(\mathscr{F})}{\mathscr{F} \vdash_f x \to \Omega} \ (F2)$$

Figure 12: Naive translation of unnested queries without condition.

The inference rules are given in Figure 12. The main judgment is $\mathscr{F},\mathscr{C} \vdash_s x \to e$ which translates a variable x identifying a query—that is, a variable declared by a row in $\mathscr{C}$—into a $\mathbb{C}$QL query e. This is done in rule *R2* which straightforwardly generates the `select` clause (just notice that element variables are banged since they denote sequences) and relies on a new form of judgment to generate the `from` clauses. A judgment $\mathscr{F} \vdash_f x \to l$ generates a list l of `from` clauses of the form «p `in` e», where p is a $\mathbb{C}$QL pattern and e is a $\mathbb{C}$QL expression whose form is either `[`y`]` or y. As we assume that there are no nested queries, then all variables free in $\mathscr{C}$ must be declared by one (and only one) row in $\mathscr{F}$ (recall that these variables cannot be persistent roots). For this reason we just need two rules to generate the `from` clauses: we use *F3* when *the* free variable of the $\mathscr{F}$-row at issue is a persistent root (in which case we can stop the search since the variable is completely defined); we use *F4* when the free variable of the $\mathscr{F}$-row at issue is a capture variable defined in some other row (in which case we have to find this row and recall the judgment $\vdash_f$ under an environment $\mathscr{F}$ from which this row is removed—to avoid loops—in order to generate the clauses l that define this variable: these clauses must precede the definition of the variable, of course). Finally the pattern corresponding to a filter tableau row is generated by the function $\mathsf{pattern}()$ which has the following definition.

Definition 3.7. *Let f be a filter tableau row of the form $FT(y|tag|k|(x_0,\vec{t_0})|(x_1,\vec{t_1})..(x_k,\vec{t_k})|(x_{k+1},\vec{t}_{k+1})..(x_{k+n},\vec{t}_{k+n}))$, where y is of type either* `<`s_0`{`a_1`=`s_1..a_k`=`s_k`}>[`R_1..R_n`]` *(i.e., y is a persistent root), or* `[<`s_0`{`a_1`=`s_1..a_k`=`s_k`}>[`R_1..R_n`]*]` *(i.e., y is a capture variable), and m denotes the arity of the various $\vec{t_i}$'s. Then $\mathsf{pattern}(f) = p_1 | \dots | p_m$ where, for j=1..m, p_j is defined as:*

$$\texttt{<(}x_0\texttt{\&}t_0^j\texttt{\&}s_0^j\texttt{)}\ \ a_1\texttt{=}x_1\texttt{\&}t_1^j\texttt{\&}s_1^j \dots a_k\texttt{=}x_k\texttt{\&}t_k^j\texttt{\&}s_k^j\texttt{>[} \\ x_{k+1}\texttt{::}s_{k+1}^j \dots x_{k+n}\texttt{::}s_{k+n}^j\texttt{]}$$

where for $i = 1..n$

$$s_{i+k}^j = \begin{cases} t_{i+k}^j\texttt{\&}R_i & \textit{if } R_i \textit{ is a type} \\ t_{i+k}^j\texttt{\&[}R_i\texttt{]} & \textit{otherwise} \end{cases}$$

The j-th row of a filter table generates the pattern p_j composing a union pattern. In each p_j, if x_i is a variable that captures an attribute, then the pattern associated to x_i is a_i=x_i&t_i^j. Otherwise we use regular expressions and the pattern is x_{i+k} :: s_{i+k}^j. The s_{i+k}^j is different according to the form of the regular expression type R_i. In the case R_i is a type (e.g. the type regular expression `Title`), then $s_{i+k}^j = t_{i+k}^j$&R_i, otherwise (e.g. the type regular expression `Book*`, which is not a type) $s_{i+k}^j = t_{i+k}^j$`&[`R_i`]`.

Finally, rules *R6* and *F2* explicitly manage the case of ill-defined PBE queries by generating an error, denoted by Ω.

Let us follow the translation on a PBE query q that groups the title and the price of each book in `doc` under a new tag `<result>` and is defined as follows

Bib	#	Book*
doc	$(x_0, \_)$	$(bks, \_)$

Book	#	Title	Author+	Publisher	Price
bks	$(x_1, \_)$	$(tls, \_)$	$(x_2, \_)$	$(x_3, \_)$	$(prc, \_)$

<result>	Title	Price
q	*tls*	*prc*

Formally $\mathscr{C} = \{CT(q|\texttt{result}|0|\ |tls\ \ prc)\}$, $\mathscr{F} = \{FT(doc|\texttt{bib}|0|(x_0,\texttt{Any})|\ |(bks,\texttt{Any})), FT(bks|\texttt{book}|0|(x_1,\texttt{Any})|(tls,\texttt{Any})(x_2,\texttt{Any})(x_3,\texttt{Any})(prc,\texttt{Any}))\}$, $\Theta = \varnothing$.

Rule *R2* is evaluated first since there exists a row in $\mathscr{C}$ which declares the query q. Thus we have:

. $\mathscr{C},\mathscr{F} \vdash_s q \to$ `select <result> [ !tls !prc ] from` l_1,l_2

Since *tls* is based on the variable *bks* which is not a persistent root, then for the computation of l_1 corresponding to *tls* we apply rule *F4*, which gives:

$\mathscr{F} \vdash_f tls \to$ l_3`,<(x1)>[tls::Title x2::Author+ x3::Publisher prc::Price] in bks`

To compute l_3 we repeat the operation on *bks* which being based on the persistent root *doc* triggers *F3*:

$\mathscr{F} \vdash_f bks \to$ `<(x0)> [ bks::Book* ] in [doc]`

Thus l_1 denotes the list:

```
<(x0)>[ bks::Book* ] in [doc],
<(x1)>[ tls::Title x2::Author+ x3::Publisher prc::Price] in bks
```

and the same computation gives for l_2:

```
<(x0)>[ books::Book* ] in [doc],
<(x1)>[ tls::Title x2::Author+ x3::Publisher prc::Price] in bks
```

In conclusion the rules of Figure 12 translate the PBE query q into the following $\mathbb{C}$QL query:

```
select <result> [ !tls !prc ] from
  <(x0)>[bks::Book* ] in [doc],
  <(x1)>[tls::Title x2::Author+ x3::Publisher prc::Price] in bks,
  <(x0)>[bks::Book* ] in [doc],
  <(x1)>[tls::Title x2::Author+ x3::Publisher prc::Price] in bks
```

It is clear that half of the lines in the `from` clauses are useless. This redundancy is due to the fact that the rules compute several times the clauses that define the variables *tls* and *prc*. To avoid this duplication we add a new memoization environment that records the set of variables already defined during the deduction, as we show in the next section.

3.2.2 Redundancy elimination for unnested queries without condition

The rules in Figure 13 define a modification of the previous translation that eliminates the redundancy we pointed out, by using in the $\vdash_f$-judgments a new environment Σ that stores the variables occurring in patterns returned by $\mathsf{pattern}()$.

The rules *F3* and *F4*, besides returning the list of clauses l,

$$\frac{\begin{array}{c}CT(x|tag|k|(a_1,x_1)...(a_k,x_k)|x_{k+1}...x_{k+n}) \in \mathscr{C}\\ \mathscr{F},\Sigma_{i-1} \vdash_f x_i \longrightarrow (l_i,\Sigma_i) \quad \Sigma_0{=}\varnothing \quad i{=}1..k{+}n\end{array}}{\begin{array}{l}\mathscr{F},\mathscr{C} \vdash_s x \longrightarrow \texttt{select <}tag\ a_1\texttt{=}x_1\ldots a_k\texttt{=}x_k\texttt{>}\\ \qquad\qquad \texttt{[!}x_{k+1}\ldots\texttt{!}x_{k+n}\texttt{] from } l_1,\ldots,l_{k+n}\end{array}} \ (R2)$$

$$\frac{x \in \Sigma}{\mathscr{F},\Sigma \vdash_f x \longrightarrow (\varnothing,\Sigma)} \ (F1)$$

$$\frac{x \notin \Sigma \quad \exists f \in \mathscr{F}, x \in \mathsf{dv}(f) \quad y \in \mathsf{fv}(f) \cap \mathscr{P}}{\mathscr{F},\Sigma \vdash_f x \longrightarrow (\mathsf{pattern}(f)\ \texttt{in}\ \texttt{[}y\texttt{]}, \Sigma \cup \mathsf{dv}(f))} \ (F3)$$

$$\frac{\begin{array}{l}x \notin \Sigma \quad \exists f \in \mathscr{F}, x \in \mathsf{dv}(f) \quad y \in \mathsf{fv}(f) \quad y \notin \mathscr{P}\\ \mathscr{F}\backslash f, \Sigma \cup \mathsf{dv}(f) \vdash_f y \longrightarrow (l_i,\Sigma')\end{array}}{\mathscr{F},\Sigma \vdash_f x \longrightarrow (l_i\ \ \texttt{,}\ \mathsf{pattern}(f)\ \texttt{in}\ y, \Sigma')} \ (F4)$$

$$\frac{x \notin \mathsf{dv}(\mathscr{C})}{\mathscr{F},\mathscr{C} \vdash_s x \longrightarrow \Omega}(R6) \qquad \frac{x \notin \Sigma \cup \mathsf{dv}(\mathscr{F})}{\mathscr{F},\Sigma \vdash_f x \longrightarrow \Omega}(F2)$$

Figure 13: Memoization for unnested queries without condition.

they now also return a new environment Σ that that enriches the current one with the variables defined in l.

The overall recording of the defined variables is performed in the rule *R2* by the premises $\mathscr{F},\Sigma_{i-1} \vdash_f x_i \rightarrow (l_i,\Sigma_i)$ where the Σ_i's are used as accumulators. Each Σ_i indeed contains all variables defined in the preceding environments, that is in any Σ_k, such as $k < i$ (where $\Sigma_0 = \varnothing$). The last environment Σ_n will then contain all the defined variables.

The elimination of redundancy is then crucially performed by the new rule *F1* which returns an empty set of `from` clauses in the case where the variable to be sought is already defined—that is, it belongs to Σ—: in this case there is no clause l to add in the construction of the query as all definitions are already present. Rule *F2* is straightforwardly modified.

By applying these rules to the example of the previous section we obtain the following $\mathbb{C}$QL query

```
select <result> [ !tls !prc ] from
  <(x0)>[bks::Book* ] in [doc],
  <(x1)>[tls::Title x2::Author+ x3::Publisher prc::Price] in bks
```

which is indeed the one we expected.

3.2.3 Nested queries without condition

We extend the previous translation to account for nested queries, that is, queries whose construct tableaux declare variables free in other construct tableaux ($\mathsf{fv}(\mathscr{C}) \cap \mathsf{dv}(\mathscr{C}) \neq \varnothing$).

Intuitively, when during the translation of a query we meet a variable, we must check whether this variable is declared in a filter tableau (it is in $\mathsf{dv}(\mathscr{F})$) or in a construct tableau (it is in $\mathsf{dv}(\mathscr{C})$). In the former case we must proceed as before, that is, insert the variable as it is in the `select` expression and generate the `from` clauses that define it. In the latter case, instead of inserting the variable in the `select` expression we have to insert the query generated by recursively calling the translation.

This is done by modifying the *R*-rules for $\vdash_s$ (the *F*-rules, which are for $\vdash_f$-judgments, do not change) as shown in Figure 14. In particular this is done in rule *R2* which for each x_i (independently from whether it is in $\mathsf{dv}(\mathscr{F})$ or in $\mathsf{dv}(\mathscr{C})$) calls for its translation (premises $\mathscr{F},\mathscr{C} \vdash_s x_i \rightarrow e_i$). If the variable is declared in a filter tableau, this results in calling

$$\frac{x \in \mathsf{dv}(\mathscr{F})}{\mathscr{F},\mathscr{C} \vdash_s x \longrightarrow x} \ (R1)$$

$$\frac{\begin{array}{c}CT(x|tag|k|(a_1,x_1)...(a_k,x_k)|x_{k+1}...x_{k+n}) \in \mathscr{C}\\ \{x_{j_1},\ldots,x_{j_m}\} = \mathsf{dv}(\mathscr{F}) \cap \{x_1,\ldots,x_{k+n}\}\\ \mathscr{F},\mathscr{C} \vdash_s x_i \longrightarrow e_i \quad i{=}1..k{+}n\\ \mathscr{F},\Sigma_{h-1} \vdash_f x_{j_h} \longrightarrow (l_h,\Sigma_h) \quad h{=}1..m \quad \Sigma_0{=}\varnothing\end{array}}{\begin{array}{l}\mathscr{F},\mathscr{C} \vdash_s x \longrightarrow \texttt{select <}tag\ a_1\texttt{=}e_1\ldots a_k\texttt{=}e_k\texttt{>}\\ \qquad\qquad \texttt{[!}e_{k+1}\ldots\texttt{!}e_{k+n}\texttt{] from } l_1,\ldots,l_m\end{array}} \ (R2)$$

$$\frac{x \notin \mathsf{dv}(\mathscr{F}) \cup \mathsf{dv}(\mathscr{C})}{\mathscr{F},\mathscr{C} \vdash_s x \longrightarrow \Omega} \ (R6)$$

(*F1*), (*F2*), (*F3*), (*F4*) as in Fig. 13

Figure 14: Translation rules for nested queries without condition.

the new rule *R1* which returns the variable (now considered as a $\mathbb{C}$QL expression), otherwise the rule *R2* is called on the new variable and the corresponding $\mathbb{C}$QL expression generated. The rule also generates the `from` clauses for the variables that are in $\mathsf{dv}(\mathscr{F})$, by the same technique as before. The rule *R6* is modified since variables free in a construct tableau may now be defined in another construct tableau (this modification is not necessary for *F2*).

Bib	#	Book*
doc	$(x_0, \_)$	$(bks, \_)$

Book	#	Title	Author+	Publisher	Price
bks	$(x_1, \_)$	$(tls, \_)$	$(a, \_)$	$(x_2, \_)$	$(x_3, \_)$

Author	#	Last	First
a	$(x_4, \_)$	$(ln, \_)$	$(fn, \_)$

`<auth>`	Last	First
p	*ln*	*fn*

`<result>`	Title	`<auth>[Last First]`
q	*tls*	*p*

Figure 15: Return titles and authors in a new element `<result>`, where the tag `auth` replaces the tag `author`.

Let us apply the translation to the tableaux of Figure 15 which contains nested construct tableaux:

$$\mathscr{C} = \{CT(q|\texttt{result}|0|\ |tls\ p)\ CT(p|\texttt{auth}|0|\ |ln\ \ fn)\}.$$

To translate the query q we apply *R2* and in particular evaluate $\mathscr{F},\mathscr{C} \vdash_s tls \rightarrow e'$ and $\mathscr{F},\mathscr{C} \vdash_s p \rightarrow e''$. Since *tls* is defined in $\mathscr{F}$, then e' is the $\mathbb{C}$QL variable `tls`. This, with the call of $\vdash_f$ to generate the definitions for *tls* yields:

```
select <result>[!tls !e'' ] from
   <(x0)>[bks::Book*  ] in [doc],
   <(x1)>[tls::Title a::Author+ x2::Publisher x3::Price] in bks
```

where e'' is the result of the evaluation of the query p. This being a variable defined in $\mathscr{C}$ fires the rule *R2*. Since the row defining p only contains variables defined in $\mathscr{F}$, then the translation is as in the previous section, yielding:

```
select <result>[!tls
                !select <auth>[ !ln !fn]
                   from  <(x0)>[books::Book*] in [doc],
                         <(x1)>[tls::Title a::Author+
                                x2::Publisher x3::Price] in bks
                         <(x4)>[ln::Last fn::First] in a
               ]
from <(x0)>[bks::Book*  ] in [doc],
     <(x1)>[tls::Title a::Author+ x2::Publisher x3::Price] in bks
```

We notice that a new form of redundancy appears as the clauses for `x0` and `x1` are uselessly computed twice. This is due to the fact that the work done for translating the inner query was already done when computing the translation of the outer query. The solution is as before, that is, we memoize the variables already met by the translation, with the difference that the variables to be stored are now defined in $\mathscr{C}$ and the environment that stores them is added to $\vdash_s$-judgments.

3.2.4 *Redundancy elimination for nested queries without condition*

We need to modify only the R-rules, whose judgments specify now a environment Σ both as input and as output. These two Σ's respectively store and return all the variables defined in the construct tableau being translated, so that these variables are taken into account (when generating `from` clauses) just once. F-rules instead need no modification, even though these rules (in particular *F2*) now work on richer Σ's that convey more information.

$$\frac{x \in \mathsf{dv}(\mathscr{F})}{\mathscr{F},\mathscr{C},\Sigma \vdash_s x \to (\Sigma, x)}\ (R1) \qquad \frac{x \notin \mathsf{dv}(\mathscr{F}) \cup \mathsf{dv}(\mathscr{C})}{\mathscr{F},\mathscr{C},\Sigma \vdash_s x \to \Omega}\ (R6)$$

$$\frac{\begin{array}{c} CT(x|tag|k|(a_1,x_1)...(a_k,x_k)|x_{k+1}...x_{k+n}) \in \mathscr{C} \\ \{x_{j_1},\dots,x_{j_m}\} = \mathsf{dv}(\mathscr{F}) \cap \{x_1,\dots,x_{k+n}\} \\ \mathscr{F},\Sigma_{h-1} \vdash_f x_{j_h} \to (l_h,\Sigma_h) \quad h=1\dots m \\ \mathscr{F},\mathscr{C},\Sigma_m \vdash_s x_i \to (\Sigma'_i, e_i) \quad i=1\dots k{+}n \end{array}}{\begin{array}{l}\mathscr{F},\mathscr{C},\Sigma_0 \vdash_s x \to (\Sigma_m, \texttt{select <}tag\ a_1{=}e_1..a_k{=}e_k\texttt{>} \\ \qquad\qquad \texttt{[!}e_{k+1}..\texttt{!}e_{k+n}\texttt{] from } l_1..l_m)\end{array}}\ (R2)$$

(*F1*), (*F2*), (*F3*), (*F4*) as in Fig. 13

Figure 16: Memoization for nested queries without condition.

In particular, *R1* and *R2* are straightforwardly extended (by adding the context environment and, for *R1*, returning it unmodified). *R2* first generates all the `from` clauses needed at the top level, and then it translates possibly nested queries under the environment Σ_m which records all the variable defined in the generation of the top-level `from` clauses. The rules in Figure 16 translate the tableaux of Figure 15 into the following (expected) query:

```
select <result>[!tls
               !select <auth>[ !ln !fn]
                  from <(x4)>[ln::Last fn::First] in a
               ]
from <(x0)>[bks::Book*  ] in [doc],
     <(x1)>[tls::Title a::Author+ x2::Publisher x3::Price] in bks
```

The rules in Figure 16 are not complete, though. A rule is still missing. The problem is that if in rule *R2* $\Sigma_0 = \Sigma_m$ holds, then the various sub-calls to the F-rules would not generate any clause, thus yielding an empty `from` part (and a syntax error). This in particular happens when all clauses needed for the definition of the variables free in some construct tableau were already generated. To see an instance of the problem, it suffices to replace in Figure 15 the first construct tableau (the one that defines the **p** variable), by the following one.

`<auth>`	`Author+`
p	a

for which the sole rules of Figure 16 would return

```
select <result>[!tls
               !select <auth>[ !a]
                  from   ]
from <(x0)>[bks::Book*  ] in [doc],
     <(x1)>[tls::Title a::Author+ x2::Publisher x3::Price] in bks
```

whose syntax is incorrect since the grayed `from` clause is empty. To avoid this problem it suffices to add to the rules of Figure 16 the following rule *R4* that for $\Sigma_0 = \Sigma_m$ returns `[`e`]` instead of "`select` e `from _` ":

$$\frac{\begin{array}{c} (\text{if } \Sigma_0 = \Sigma_m) \\ CT(x|tag|k|(a_1,x_1)...(a_k,x_k)|x_{k+1}...x_{k+n}) \in \mathscr{C} \\ \{x_{j_1},\dots,x_{j_m}\} = \mathsf{dv}(\mathscr{F}) \cap \{x_1,\dots,x_{k+n}\} \\ \mathscr{F},\Sigma_{h-1} \vdash_f x_{j_h} \to (l_h,\Sigma_h) \quad h=1\dots m \\ \mathscr{F},\mathscr{C},\Sigma_m \vdash_s x_i \to (\Sigma'_i, e_i) \quad i=1\dots k{+}n \end{array}}{\begin{array}{l}\mathscr{F},\mathscr{C},\Sigma_0 \vdash_s x \to (\Sigma_m, \texttt{[<}tag\ \ a_1{=}e_1\dots a_k{=}e_k\texttt{>} \\ \qquad\qquad\qquad \texttt{[!}e_{k+1}\dots\texttt{!}e_{k+n}\texttt{]])}\end{array}}\ (R4)$$

With this new rule the previous example translates to:

```
select <result>[ !tls ![<auth>[!a]] ]
from <(x0)>[bks::Book*] in [doc],
     <(x1)>[tls::Title a::Author+ x2::Publisher x3::Price] in bks
```

3.2.5 *Nested queries with condition.*

Finally, the most general case, in which $\Theta \neq \varnothing$ needs the new rules *C1*-*C7* of Figure 17. These have as input $\mathscr{F}$, Σ and Θ and generate a $\mathbb{C}$QL condition C that translates the rows that use variables in Σ (that is, variables used by the query being translated). The output also includes the list l of `from` clauses that were created during the construction of C. These clauses are created when Θ uses variables not already treated (hence, not belonging to Σ). Of course, we need to keep track of these variables for subsequent analysis steps, in order to avoid the creation of duplicated `from` clauses. This explains the third output of $\vdash_c$, an environment Σ' that collects all the newly encountered and treated variables.

The first two C-rules handle the base cases where there are no conditions to create, either because Σ is empty and thus the query being translated does not define any new variable (*C1*) or because there are no more condition rows to translate (*C2*). Rule *C3* handles the case where the selected condition uses only one variable x and this variable is not already defined by a `from` clause (i.e., $x \notin \Sigma$) . This means that the condition is not relevant for the query being created, and therefore we may drop this condition-box row and continue with other conditions. Rule *C4* handles the case of one-variable condition where the variable was already treated. Rules *C5* and *C6* are the two-variables counterparts of *C3* and *C4*, respectively (in this sense *C1* is an optimization of *C3* and *C5*). Finally, rule *C7* handles the case of a two-variable condition, where just one of the two variables has not been treated (it is not in Σ). Since one of the two variables is already defined, we have to generate the `from` clauses that define the other one, which is done by the last premise in the rule. We omitted the symmetric cases of *C3*, *C4*, and *C7* in which operands are swapped.

The R-rules are modified as well, in particular by the addition of Θ to the inputs and of the calls to $\vdash_c$ to generate conditions. When these calls do not generate any condition (rules *R2*, *R4*), then the rules work as before. If instead the calls generate a condition C, then this is added to the translation. Rule *R3* adds C as the `where` clause of the generated `select` expression (plus all the generated `from`

$$\frac{x \in \mathsf{dv}(\mathscr{F})}{\mathscr{F}, \mathscr{C}, \Sigma, \Theta \vdash_s x \rightarrow (\Sigma, x)} \;(R1)$$

$$\frac{\begin{array}{c}(\text{if } \Sigma_0 \neq \Sigma_m)\\ CT(x|tag|k|(a_1, x_1)\ldots(a_k, x_k)|x_{k+1}\ldots x_{k+n}) \in \mathscr{C}\\ \{x_{j_1}, \ldots, x_{j_m}\} = \mathsf{dv}(\mathscr{F}) \cap \{x_1, \ldots, x_{k+n}\}\\ \mathscr{F}, \Sigma_{h-1} \vdash_f x_{j_h} \rightarrow (l_h, \Sigma_h) \quad \mathscr{F}, \mathscr{C}, \Sigma_m, \Theta \vdash_s x_i \rightarrow (\Sigma'_i, e_i)\\ \mathscr{F}, \Sigma_m, \Theta \vdash_c (\varnothing, \varnothing, \Sigma_m) \quad i = 1\ldots k{+}n \quad h = 1\ldots m\end{array}}{\begin{array}{r}\mathscr{F}, \mathscr{C}, \Sigma_0, \Theta \vdash_s x \rightarrow (\Sigma_m, \texttt{select <}tag\ a_1\texttt{=}e_1\ldots a_k\texttt{=}e_k\texttt{>}\\ \texttt{[!}e_{k+1}\ldots\texttt{!}e_{k+n}\texttt{] from } l_1, \ldots, l_m)\end{array}} \;(R2)$$

$$\frac{\begin{array}{c}(\text{if } \Sigma_0 \neq \Sigma')\\ CT(x|tag|k|(a_1, x_1)\ldots(a_k, x_k)|x_{k+1}\ldots x_{k+n}) \in \mathscr{C}\\ \{x_{j_1}, \ldots, x_{j_m}\} = \mathsf{dv}(\mathscr{F}) \cap \{x_1, \ldots, x_{k+n}\}\\ \mathscr{F}, \Sigma_{h-1} \vdash_f x_{j_h} \rightarrow (l_h, \Sigma_h) \quad \mathscr{F}, \mathscr{C}, \Sigma_m, \Theta \vdash_s x_i \rightarrow (\Sigma'_i, e_i)\\ \mathscr{F}, \Sigma_m, \Theta \vdash_c (C, l_c, \Sigma') \quad i = 1\ldots k{+}n \quad h = 1\ldots m\end{array}}{\begin{array}{r}\mathscr{F}, \mathscr{C}, \Sigma_0, \Theta \vdash_s x \rightarrow (\Sigma', \texttt{select <}tag\ a_1\texttt{=}e_1..a_k\texttt{=}e_k\texttt{>}\\ \texttt{[!}e_{k+1}..\texttt{!}e_{k+n}\texttt{] from } l_1..l_m\texttt{,}l_c \texttt{ where } C)\end{array}} \;(R3)$$

$$\frac{\begin{array}{c}(\text{if } \Sigma_0 = \Sigma_m)\\ CT(x|tag|k|(a_1, x_1)\ldots(a_k, x_k)|x_{k+1}\ldots x_{k+n}) \in \mathscr{C}\\ \{x_{j_1}, \ldots, x_{j_m}\} = \mathsf{dv}(\mathscr{F}) \cap \{x_1, \ldots, x_{k+n}\}\\ \mathscr{F}, \Sigma_{h-1} \vdash_f x_{j_h} \rightarrow (l_h, \Sigma_h) \quad \mathscr{F}, \mathscr{C}, \Sigma_m, \Theta \vdash_s x_i \rightarrow (\Sigma'_i, e_i)\\ \mathscr{F}, \Sigma_m, \Theta \vdash_c (\varnothing, \varnothing, \Sigma_m) \quad i = 1\ldots k{+}n \quad h = 1\ldots m\end{array}}{\begin{array}{r}\mathscr{F}, \mathscr{C}, \Sigma_0, \Theta \vdash_s x \rightarrow (\Sigma_m, \texttt{[<}tag\ a_1\texttt{=}e_1\ldots a_k\texttt{=}e_k\texttt{>}\\ \texttt{[!}e_{k+1}\ldots\texttt{!}e_{k+n}\texttt{]]})\end{array}} \;(R4)$$

$$\frac{\begin{array}{c}(\text{if } \Sigma_0 = \Sigma_m)\\ CT(x|tag|k|(a_1, x_1)\ldots(a_k, x_k)|x_{k+1}\ldots x_{k+n}) \in \mathscr{C}\\ \{x_{j_1}, \ldots, x_{j_m}\} = \mathsf{dv}(\mathscr{F}) \cap \{x_1, \ldots, x_{k+n}\}\\ \mathscr{F}, \Sigma_{h-1} \vdash_f x_{j_h} \rightarrow (l_h, \Sigma_h) \quad \mathscr{F}, \mathscr{C}, \Sigma_m, \Theta \vdash_s x_i \rightarrow (\Sigma'_i, e_i)\\ \mathscr{F}, \Sigma_m, \Theta \vdash_c (C, \varnothing, \Sigma_m)\\ i = 1\ldots k{+}n \quad h = 1\ldots m\end{array}}{\begin{array}{r}\mathscr{F}, \mathscr{C}, \Sigma_0, \Theta \vdash_s x \rightarrow (\Sigma_m, \texttt{if } C \texttt{ then [<}tag\ a_1\texttt{=}e_1\ldots a_k\texttt{=}e_k\texttt{>}\\ \texttt{[!}e_{k+1}\ldots\texttt{!}e_{k+n}\texttt{]] else []})\end{array}} \;(R5)$$

$$\frac{x \notin \mathsf{dv}(\mathscr{F}) \cup \mathsf{dv}(\mathscr{C})}{\mathscr{F}, \mathscr{C}, \Sigma, \Theta \vdash_s x \rightarrow \Omega} \;(R6)$$

$$\frac{x \in \Sigma}{\mathscr{F}, \Sigma \vdash_f x \rightarrow (\varnothing, \Sigma)} \;(F1)$$

$$\frac{x \notin \Sigma \cup \mathsf{dv}(\mathscr{F})}{\mathscr{F}, \Sigma \vdash_f x \rightarrow \Omega} \;(F2)$$

$$\frac{x \notin \Sigma \quad \exists f \in \mathscr{F}, x \in \mathsf{dv}(f) \quad y \in \mathsf{fv}(f) \cap \mathscr{P}}{\mathscr{F}, \Sigma \vdash_f x \rightarrow (\mathsf{pattern}(f)\ \texttt{in}\ \texttt{[}y\texttt{]}, \Sigma \cup \mathsf{dv}(f))} \;(F3)$$

$$\frac{\begin{array}{l}x \notin \Sigma \quad \exists f \in \mathscr{F}, x \in \mathsf{dv}(f) \quad y \in \mathsf{fv}(f)\\ y \notin \mathscr{P} \quad \mathscr{F}\backslash f, \Sigma \cup \mathsf{dv}(f) \vdash_f y \rightarrow (l_i, \Sigma')\end{array}}{\mathscr{F}, \Sigma \vdash_f x \rightarrow (l_i\ \texttt{,}\ \mathsf{pattern}(f)\ \texttt{in}\ y, \Sigma')} \;(F4)$$

$$\frac{}{\mathscr{F}, \varnothing, \Theta \vdash_c (\varnothing, \varnothing, \varnothing)} \;(C1)$$

$$\frac{}{\mathscr{F}, \Sigma, \varnothing \vdash_c (\varnothing, \varnothing, \Sigma)} \;(C2)$$

$$\frac{\begin{array}{l}r = CB(\mathsf{op}, x, v) \in \Theta \quad x \notin \Sigma\\ \mathscr{F}, \Sigma, \Theta\backslash r \vdash_c (C, l, \Sigma')\end{array}}{\mathscr{F}, \Sigma, \Theta \vdash_c (C, l, \Sigma')} \;(C3)$$

$$\frac{\begin{array}{l}r = CB(\mathsf{op}, x, v) \in \Theta \quad x \in \Sigma\\ \mathscr{F}, \Sigma, \Theta\backslash r \vdash_c (C, l, \Sigma')\end{array}}{\mathscr{F}, \Sigma, \Theta \vdash_c (C\ \texttt{and}\ \texttt{(}x\ \mathsf{op}\ v\texttt{)}, l, \Sigma')} \;(C4)$$

$$\frac{\begin{array}{l}r = CB(\mathsf{op}, x_1, x_2) \in \Theta \quad x_1 \notin \Sigma \quad x_2 \notin \Sigma\\ \mathscr{F}, \Sigma, \Theta\backslash r \vdash_c (C, l, \Sigma')\end{array}}{\mathscr{F}, \Sigma, \Theta \vdash_c (C, l, \Sigma')} \;(C5)$$

$$\frac{\begin{array}{l}r = CB(\mathsf{op}, x_1, x_2) \in \Theta \quad x_1 \in \Sigma \quad x_2 \in \Sigma\\ \mathscr{F}, \Sigma, \Theta\backslash r \vdash_c (C, l, \Sigma')\end{array}}{\mathscr{F}, \Sigma, \Theta \vdash_c (C\ \texttt{and}\ \texttt{(}x_1\ \mathsf{op}\ x_2\texttt{)}, l, \Sigma')} \;(C6)$$

$$\frac{\begin{array}{l}r = CB(\mathsf{op}, x_1, x_2) \in \Theta \quad x_1 \in \Sigma \quad x_2 \notin \Sigma\\ \mathscr{F}, \Sigma, \Theta\backslash r \vdash_c (C, l_1, \Sigma') \quad \mathscr{F}, \Sigma' \vdash_f x_2 \rightarrow (l_2, \Sigma'')\end{array}}{\mathscr{F}, \Sigma, \Theta \vdash_c (C\ \texttt{and (}x_1\ \mathsf{op}\, x_2\texttt{)}\ \texttt{,}\ l_1\texttt{,}l_2\ \texttt{,}\ \Sigma'')} \;(C7)$$

Figure 17: Translation rules for nested queries with condition.

clauses). Rule *R5* handles the special case in which the various sub-calls generates an empty set of `from` clauses (it is the non-empty condition counterpart of rule *R4*) and therefore there is no `select` expression to which stick C as a `where` clause: in this case an `if_then_else` ℂQL operator is used instead.

Bib	#	Book*
doc	$(x_0, \_)$	$(bks, \_)$

Book	#	Title	Author+	Publisher	Price
bks	$(x_1, \_)$	$(tls_1, \_)$	$(x_2, \_)$	$(x_3, \_)$	$(x_4, \_)$

Entries	#	Entry*
bstore2	$(x_5, \_)$	$(reviews, \_)$

Entry	#	Title	Price	Review
reviews	$(x_6, \_)$	$(tls_2, \_)$	$(x_7, \_)$	$(x_8, \_)$

<result>	Title
q	tls_1

CONDITION BOX
tls_1=tls_2

Figure 18: Titles that appear both in doc and in bstore2.

The PBE query of Figure 18 defines the query Q5 of *XML Query Use Cases* [8], which is interesting since it contains a join condition $tls_1 = tls_2$. The generation of the corresponding ℂQL query, relies on rule *C7*, when the `from` clause for tls_1 occurring Θ has been created, but tls_2 has not been defined yet. The result is:

```
select <result>[!tls1
from <(x0)>[bks::Book*] in [doc],
     <(x1)>[tls1::Title x2::Author+ x3::Publisher x4::Price] in bks,
     <(x5)>[reviews::Entry*] in [bstore2],
     <(x6)>[tls2::Title x7::Price x8::Review ] in reviews
where tls1=tls2
```

The translation of well-defined PBE queries always terminates and yields well-typed ℂQL expressions, as stated by the following theorem

Theorem 3.8. *Let $Q = (\mathscr{F}, \mathscr{C}, \mathscr{P}, \Theta)$ be a PBE query. For every $x \in \mathsf{dv}(\mathscr{C})$ there exists a unique e such that the judgment $\mathscr{F}, \mathscr{C}, \varnothing, \Theta \vdash_s x \rightarrow e$ is provable. Furthermore, if Q is well defined, then e is a well-typed* ℂQL *expression (in* *particular, $e \neq \Omega$) up to exhaustiveness of pattern matching.*[5]

3.3 Further design issues

So far the interpretation of tableaux, although technically difficult, is rather uncontroversial: the given semantics implements what one intuitively expects from tableaux. There are however some design choices that are not so obvious and that can be interesting to allow more advanced uses of the language. In particular, should constraints given in some filter tableau for a variable defined in a different filter tableau apply locally or globally? Note that the latter choice is the one done by QBE Also, should we relax the restrictions on variables declared on multiple rows of a filter tableau, accept rows that declare distinct variables, and considered them as intersection patterns? For space reasons the discussion of these two options (which are easily implemented and currently under consideration for inclusion in PBE) are available at `www.cduce.org/paper/pbe.pdf`.

4. CONCLUSION AND FUTURE WORK

PBE is a graphical interface that allows users with little or no knowledge of XPath, XQuery, or ℂQL to define complex and optimized queries on XML documents. The only required skill is to be able to understand XML types written using pretty intuitive and standard conventions of type regular expressions. At road test we found the usage of PBE quite simple and intuitive. Of course this is a subjective view, but PBE has two objective and important advantages with respect to other graphical query languages. The first is that it generates queries that are provably correct with respect to types. The type of the result is displayed to the user and this constitutes a first and immediate visual yardstick to check semantic correctness of the resulting query. The second advantage is that its semantics is formally—thus, unambiguously—defined, and this is an important advancement over some current approaches in which the standard usage and learning methods are based on "trial and error" techniques (a.k.a. "click and hope").

The implementation of PBE developed in OCaml is in alpha-testing and available at `www.lri.fr/~miachon/pbe`. It relies for its graphical part on LablGTK, on the ℂDuce's type engine for computing table entries, and uses ℂQL as back-end. Its kismet is its inclusion in the official ℂDuce distribution (`www.cduce.org`), but before some improvements are still needed. Some are purely ergonomic, such as the possibility of defining DTDs by using tableaux, the early detection of useless filter tableaux rows (see Footnote 5), the elimination of explicit variables by replacing them by "drag-and-drop" techniques. Others are enhancement features: foremost we want to allow the user to split an automatically generated column into several equivalent ones (for instance, if a user wants to capture exactly the second author of a book, (s)he should be allowed to split the `Author+` column of the first filter tableau in Figure 8 into three columns, one for the first author, another for the second author, and a last one for the remaining authors); but we want also devise a way to express unions or complex constraints without the necessity of writing complex type regular expressions in filter tableau rows.

In our future plans there also is the use for PBE of different back-ends, *in primis* XQuery. Such a modification is not straightforward because XPath selections are not as fine grained on sequences as PBE ones. If for instance we query a document of type `<a>[B* C* B*]` and insert a variable in the first column of the corresponding filter tableau, then this variable must be translated into an XPath expression with a non-trivial condition that captures all `B` elements whose left siblings do not include `C` elements.

[5]The definition of well-defined query does not ensure that all the rows of a filter tableau are useful. For instance, every row following a row with all constraints equal to `Any` will never be used. This property can be easily checked at construction time but its definition would have required the introduction of several technical definitions of the ℂDuce type system. We preferred to keep the definition simple, as these errors are statically detected as soon as the query is generated (more precisely, as soon as the `pattern`() funcion is called).

5. REFERENCES

[1] V. Benzaken, G. Castagna, and A. Frisch. CDuce: an XML-friendly general purpose language. In *ICFP '03, 8th ACM Int. Conf. on Functional Programming*, pages 51–63. ACM Press, 2003.

[2] V. Benzaken, G. Castagna, and C. Miachon. A full pattern-based paradigm for XML query processing. In *PADL 05, 7th Int. Symp. on Practical Aspects of Declarative Languages*, number 3350 in LNCS, pages 235–252. Springer, 2005.

[3] S. Berger, F. Bry, S. Schaffert, and Ch. Wieser. Xcerpt and visXcerpt: From pattern-based to visual querying of XML and semistructured data. In *VLDB*, pages 1053–1056, 2003.

[4] S. Boag, D. Chamberlin, M. Fernandez, D. Florescu, J. Robie, J. Siméon, and M. Stefanescu. *XQuery 1.0: An XML Query Language*. W3C Working Draft, `http://www.w3.org/TR/xquery/`, May 2003.

[5] L. Bouganim, T. Chan-Sine-Ying, T-T. Dang-Ngoc, J-L Darroux, G. Gardarin, and F. Sha. Miro web: Integrating multiple data sources through semistructured data types. In *The VLDB Journal*, pages 750–753, 1999.

[6] D. Braga, A. Campi, and S. Ceri. "XQBE (XQuery By Example): A visual interface to the standard XML query language". *TODS*, 30:398–443, 2005.

[7] M. J. Carey, L. M. Haas, V. Maganty, and J. H. Williams. Pesto : An integrated query/browser for object databases. In *VLDB*, pages 203–214, 1996.

[8] D. Chamberlin, P. Fankhauser, D. Florescu, M. Marchiori, and J. Robie. XML Query Use Cases. Technical Report 20030822, World Wide Web Consortium, 2003.

[9] S. Cohen, Y. Kanza, Y. A. Kogan, W. Nutt, Y. Sagiv, and A. Serebrenik. Equix easy querying in XML databases. In *WebDB (Informal Proceedings)*, pages 43–48, 1999.

[10] M. Erwig. Xing: A visual XML query language. *Journal of Visual Languages and Computing*, 14(1):5–45, 2003.

[11] I. Filha, A. Laender, and A. da Silva. Querying Semi-structured Data By Example: The QSByE Interface. In *Workshop on Information Integration on the Web*, 2001.

[12] H. Hosoya and B. Pierce. XDuce: A typed XML processing language. *ACM Transactions on Internet Technology*, 3(2):117–148, 2003.

[13] C. Miachon. *Langages de requêtes pour XML à base de patterns : conception, optimisation et implantation.* PhD thesis, Université Paris Sud, 2006.

[14] K. D. Munroe and Y. Papakonstantinou. BBQ: A visual interface for integrated browsing and querying of XML. In *VLDB*, 2000.

[15] M. Petropoulos, Y. Papakonstantinou, and V. Vassalos. Graphical query interfaces for semistructured data: the QURSED system. *TOIT*, 5(2):390–438, May 2005.

[16] M. Zloof. Query-by-example: A data base language. *IBM Systems Journal*, 16(4):324–343, 1977.