Patterns and Types for Querying XML Documents

Giuseppe Castagna

CNRS, École Normale Supérieure de Paris, France

Abstract. Among various proposals for primitives for deconstructing XML data two approaches seem to clearly stem from practise: path expressions, widely adopted by the database community, and regular expression patterns, mainly developed and studied in the programming language community. We think that the two approaches are complementary and should be both integrated in languages for XML, and we see in that an opportunity of collaboration between the two communities. With this aim, we give a presentation of regular expression patterns and the type systems they are tightly coupled with. Although this article advocates a construction promoted by the programming language community, we will try to stress some characteristics that the database community, we hope, may find interesting.

1 Introduction

Working on XML trees requires at least two different kinds of language primitives: (i) deconstruction/extraction primitives (usually called patterns or templates) that pinpoint and capture subparts of the XML data, and (ii) iteration primitives, that iterate over XML trees the process of extraction and transformation of data.

For what concerns iteration primitives, there are many quite disparate proposals: in this category one can find such different primitives as the FLWR (i.e., for-let-where-return) expressions of XQuery [7], the select-from-where of Cω [6] and ℂQL [5], the select-where of Lorel [1] and loto-ql [51], the `filter` primitive of XDuce [40, 39], the `xtransform` primitive of ℂDuce [4], the `iterate` primitive of Xtatic [31], while for other languages, for instance XSLT [22], the iterator is hard-coded in the semantics itself of the language.

For what concerns deconstructing primitives, instead, the situation looks clearer since, among various proposals (see the related work section later on), two different and complementary solutions clearly stem from practice: path expressions (usually XPath paths [21], but also the "dot" navigations of Cω or Lorel [1], caterpillar expressions [12] and their "looping" extension [33]) and regular expression patterns [41].

Path expressions are navigational primitives that pinpoint where to capture data subcomponents. XML path expressions (and those of Cω and Lorel in particular) closely resemble the homonimic primitives used by OQL [23] in the context of OODB query languages, with the difference that instead of sets of objects they return sets or sequences of XML elements: more precisely all elements that can be reached by following the paths at issue. These primitives are at the basis of standard languages such as XSLT and XQuery.

More recently, a new kind of deconstruction primitive was proposed: regular expression patterns [41], which extends by regular expressions the pattern matching primitive as popularised by functional languages such as ML and Haskell. Regular expression patterns were first introduced in the XDuce programming language and are becoming more and more popular, since they are being adopted by such quite different languages as ℂDuce [4] (a general purpose extension of the XDuce language) and its query language ℂQL [5], Xtatic [31] (an extension of C#), Scala [54] (a general purpose Java-like object-oriented language that compiles to Java bytecode), XHaskell [45] as well as the extension of Haskell proposed by Broberg *et al.* [11].

The two kinds of primitives are not antagonist, but rather orthogonal and complementary. Path expressions implement a "vertical" exploration of data as they capture elements that may be at different depths, while patterns perform a "horizontal" exploration of data since they are able to perform finer grained decomposition on sequences of elements. The two kinds of primitives are quite useful and they complement each other nicely. Therefore, it would seem natural to integrate both of them in a query or programming language for XML. In spite of this and of several theoretical works on the topic (see the related work section), we are aware of just two running languages in which both primitives are embedded (and, yet, loosely coupled): in ℂQL [5] it is possible to write select-from-where expressions, where regular expression patterns are applied in the from clause to sequences that are returned by XPath-like expressions (see the example at the end of Section 2.3); Gapeyev and Pierce [32] show how it is possible to use regular expression patterns with an all-matches semantics to encode a subset of XPath and use this encoding to add XPath to the Xtatic programming language.

The reason for the lack of study of the integration of these two primitives may be due to the fact that each of them is adopted by a different community: regular patterns are almost confined to the programming language community while XPath expressions are pervasive in the database community.

The goal of this lecture is to give a brief presentation of the regular pattern expressions style together with the type system they are tightly coupled with, that is the *semantic subtyping*-based type systems [19, 29]. We are not promoting the use of these to the detriment of path expressions, since we think that the two approaches should be integrated in the same language and we see in that a great opportunity of collaboration between the database and the programming languages communities. Since the author belongs to latter, this lecture tries to describe the pattern approach addressing some points that, we hope, should be of interest to the database community as well. In particular, after a general overview of regular expression patterns and types (Section 2) in which we show how to embed patterns in a select-from-where expression, we discuss several usages of these semantic subtyping based patterns/types (henceforward, we will often call them "semantic patterns/types"): how to use these patterns and types to give informative error messages (Section 3.2), to dig out errors that are out of reach of previous type checker technologies (Section 3.3) and how the static information they give can be used to define very efficient and highly optimised runtimes (Section 3.4); we show that these patterns permit new logical query optimisations (Section 3.5) and can be used as building blocks to allow the programmer to fine-grainedly define new iterators on data (Section 3.6); finally, the techniques developed for the semantic patterns and types can be used to define optimal data pruning and other optimisation techniques (Section 3.7–3.8)

Related work. In this work we focus on data extraction primitives coming from the *practice* of programming and query languages manipulating XML data. Thus, we restrict our attention to the primitives included in full-featured languages with a stable community of users. There are however many other proposals in the literature for deconstructing, extracting, and querying XML data.

First and foremost there are all the languages developed from logics for unranked trees whose yardstick in term of expressiveness is the Monadic Second Order Logic. The list here would be too long and we invite the interested reader to consult the excellent overview by Leonid Libkin on the subject [44]. In this area we want to single out the work on composition of monadic queries in [26], since it looks as a promising step toward the integration of path and pattern primitives we are promoting in this work: we will say more about it in the conclusion. A second work that we want to distinguish is Neven and Schwentick's ETL [49], where regular expressions over logical formulæ allow both horizontal and vertical exploration of data; but, as the authors themselves remark, the gap with a usable pattern language is very important, especially if one wants to define non-unary queries typical of Hosoya's regular expressions patterns.

Based on logics also are the query languages developed on or inspired to Ambient Logic, a modal logic that can express spatial properties on unordered trees, as well as to other spatial logics. The result is a very interesting mix of path-like and pattern-like primitives (cf. the dot notation and the spatial formulæ with capture variables that can be found in TQL) [24, 13, 16, 14, 15, 17].

In the query language research, we want to signal the work of Papakonstantinou and Vianu [51] where the loto-ql query language is introduced. In loto-ql it is possible to write `select` x `where` p, where p is a pattern in the form of tree which uses regular expressions to navigate both horizontally and vertically in the input tree, and provides bindings of x.

2 A brief introduction to patterns and types for XML

In this section we give a short survey of patterns and types for XML. We start with a presentation of pattern matching as it can be found in functional languages (Section 2.1), followed by a description of "semantic" types and of pattern-based query primitives (Section 2.2); a description of regular expression patterns for XML (Section 2.3) and their formal definition (Section 2.4) follow, and few comments on iterators (Section 2.5) close the section. Since we introduce early in this section new concepts and notations that will be used in the rest of the article, we advise also the knowledgeable reader to consult it.

2.1 Pattern matching in functional languages

Pattern matching is used in functional languages as a convenient way to capture subparts of non-functional[1] values, by binding them to some variables. For instance, imagine that

[1] We intend *non-functional* in a strict sense. So non-functional values are integer and boolean constants, pair of values, record of values, etc., but not λ-abstractions. Similarly a non-functional type is any type that is not an arrow type.

e is an expression denoting a pair and that we want to bind to x and y respectively to the first and second projection of e, so as to use them in some expression e'. Without patterns this is usually done by two `let` expressions:

```
let x = first(e) in
let y = second(e) in e'
```

With patterns this can be obtained by a single let expression:

```
let (x,y) = e in e'
```

The pattern `(x,y)` simply reproduces the form of the expected result of e and variables indicate the parts of the value that are to be captured: the value returned by e is *matched* against the pattern and the result of this matching is a *substitution*; in the specific case, it is the substitution that assigns the first projection of (the result of) e to x and the second one to y.

If we are not interested in capturing all the parts that compose the result of e, then we can use the wildcard "`_`" in correspondence of the parts we want to discard. For instance, in order to capture just the first projection of e, we can use the following pattern:

```
let (x,_) = e in ...
```

which returns the substitution that assigns the result of *first*(e) to x. In general, a pattern has the form of a value in which some sub-occurrences are replaced by variables (these correspond to parts that are to be captured) and other are replaced by "`_`" (these correspond to parts that are to be discarded). A value is then matched against a pattern and if they both have the same structure, then the matching operation returns the substitution of the pattern variables by the corresponding occurrences of the value. If they do not have the same structure the matching operation *fails*. Since a pattern may fail—and here resides the power of pattern matching—it is interesting to try on the same value several different patterns. This is usually done with a `match` expression, where several patterns, separated by `|`, are tried in succession (according to a so-called "first match" policy). For instance:

```
match e with
  | (_,_) -> true
  | _ -> false
```

first checks whether e returns a pair in which case it returns `true`, otherwise it returns `false`. Note that, in some sense, matching is not very different from a type case. Actually, if we carefully define the syntax of our types, in particular if we use the same syntax for constructing types and their values, then the `match` operation *becomes* a type case: let us write (s,t) for the product type of the types s and t (instead of the more common $s \times t$ or $s * t$ notations) and use the wildcard "`_`" to denote the super-type of all types (instead of the more common Top, $\mathbb{1}$, or $\top$ symbols), then the match expression above is indeed a type case (if the result of e is in the product type `(_,_)` —the type of all products—, then return true else if it is of type top—all values have this type—, then return false). We will see the advantages of such a notation later on, for the time being just notice that with such a syntactic convention for types and values, a pattern is a (non-functional) type in which some variables may appear.

Remark 1. *A* pattern *is just a non-functional type where some occurrences may be variables.*

The matching operation is very useful in the definition of functions, as it allows the programmer to define them by cases on the input. For instance, imagine that we encode lists recursively *à la* lisp, that is, either by a nil element for the empty list, or by pairs in which the left projection is the head and the right projection the tail of the list. With our syntax for products and top this corresponds to the recursive definition `List = ‘nil | (_,List)`: a list is either `‘nil` (we use a back-quote to denote constants so to syntactically distinguish them in patterns from variables) or the product of any type and a list. We can now write a tail recursive function[2] that computes the length of a list[3]

```
fun length ((List,Int) -> Int)
  | (‘nil , n) -> n
  | ((_,t), n) -> length(t,n+1)
```

which is declared (see Footnote 3 for notation) to be of type `(List,Int) -> Int`, that is, it takes a pair composed of a list and an integer and returns an integer. More precisely, it takes the list of elements still to be counted and the number of elements already counted (thus `length(a,0)` computes the length of the list `a`). If the list is `‘nil`, then the function returns the integer captured by the pattern variable `n`, otherwise it discards the head of the list (by using a wildcard) and performs a recursive call on the tail, captured in `t`, and on `n+1`. Note that, as shown by the use of ‘nil in the first pattern, patterns can also specify values. When a pattern contains a value v, then it matches only values in which the value v occurs in the same position. Remark 1 is still valid even in the case that values occur in patterns, since we can still consider a pattern as a type with variables: it suffices to consider a value as being the denotation of the singleton type that contains that value.

2.2 Union, intersection, and difference types

In order to type-check `match` expressions, the type-checker must compute unions, intersections, and differences (or, equivalently, negations) of types: let us denote these

[2] A function is *tail recursive* if all recursive calls in its definition occur at the end of its execution flow (more precisely, it is tail recursive if the result of every call is equal to result of its recursive calls): this allows the compiler to optimise the execution of such functions, since it then becomes useless to save and restore the state of recursive calls since the result will be pushed on the top of the stack by the last recursive call.

[3] We use two different syntaxes for functions. The usual notation is standard: for instance, the identity function on integers will be written as `fun id(x:Int):Int = x`. But if we want to feed the arguments of a function directly to a pattern matching, then the name of the function will be immediately followed by the type of the function itself. In this notation the identity for integers is rather written as `fun id(Int->Int) x -> x`. This is the case for the function length that follows, which could be equivalently defined as

```
fun length (x :(List,Int)):Int =
  match x with
    | (‘nil , n) -> n
    | ((_,t), n) -> length(t,n+1)
```

operations by `|` for the union, `&` for the intersection, and \ for the difference. The reason why the type-checker needs to compute them can be better understood if we consider a type as a set of values, more precisely as the set of values that have that type: $t = \{v \mid v$ value of type $t\}$.[4] For instance, the product of the singleton type `‘nil` and of the type `Int`, denoted by `(‘nil,Int)`, will be the set of all pairs in which the first element is the constant `‘nil` and the second element is an integer. Notice that we already implicitly did such an hypothesis at the end of the previous section, when we considered a singleton type as a type *containing* just one value.

As we did for types, it is possible to associate also patterns to sets of values (actually, to types). Specifically, we associate to a pattern p the type $\wr p \int$ defined as the set of values for which the pattern does not fail: $\wr p \int = \{v \mid v \text{ matches pattern } p\}$. Since we use the same syntax for type constructors and value constructors, it results quite straightforward to compute $\wr p \int$: it is the type obtained from p by substituting “`_`” for all occurrences of variables: the occurrences of values are now interpreted as the corresponding singleton types.

Let us check whether the function `length` has the type `(List,Int)` $\rightarrow$ `Int` it declares to have. The function is formed by two branches, each one corresponding to a different pattern. To know the type of the first branch we need to know the set of values (i.e., the type) that can be bound to n; the branch at issue will be selected and executed only for values that are arguments of the function—so that are in `(List,Int)`—and that are accepted by the pattern of the branch—so that are in $\wr$`(‘nil,n)`$\int$ which by definition is equal to `(‘nil,_)`—. Thus, these are the values in the intersection `(List,Int)&(‘nil,_)`. By distributing the intersection on products and noticing that `List&‘nil`= `‘nil` and `Int&_`= `Int`, we deduce that the branch is executed for values in `(‘nil,Int)` and thus n is (bound to values) of type `Int`. The second branch returns a result of type `Int` (the result type declared for the function) provided that the recursive call is well-typed. In order to verify it, we need once more to compute the set of values for which the branch will be executed. These are the arguments of the function, minus the values accepted by the first branch, and intersected with the set of values accepted by the pattern of second branch, that is: `((List,Int)/(‘nil,_)) & ((_,_),_)`. Again, it is easy to see that this type is equal to `((_,List),Int)` and deduce that variable t is of type `List` and the variable n is of type `Int`: since the arguments have the expected types, then the application of the recursive call is well typed. The type of the result of the whole function is the union of the types of the two branches: since both return integers the union is integer. Finally, notice also that the match is *exhaustive*, that is, for every possible value that can be fed to the match, there exists at least one pattern that matches it. This holds true because the set of all arguments of the the function (that is, its domain) is contained in the union of the types accepted by the patterns.

More generally, to deduce the type of an expression (for the sake of simplicity we use a match expression with just two patterns)

$$\texttt{match}\ e\ \texttt{with}\ p_1\texttt{->}e_1\ \texttt{|}\ p_2\texttt{->}e_2$$

[4] Formally, we are not defining the types, we are giving their semantics. So a type “is interpreted as” or “denotes” a set of values. We prefer not to enter in such a distinction here. See [19] for a more formal introduction about these types.

one must: (i) deduce the type t of e, (ii) calculate the type t_1 of e_1 in function of the values in $t\&\lbag p_1 \rbag$, (iii) calculate the type t_2 of e_2 in function of the values in $(t\backslash\lbag p_1 \rbag)\&\lbag p_2 \rbag$ and, finally, (iv) check whether the match is exhaustive that is, $t \leq \lbag p_1 \rbag | \lbag p_2 \rbag$: the type of the expression is then the union $t_1 | t_2$.

The example with `match` clearly shows that for a precise typing of the programs the type-checker needs to compute unions, intersections, and differences of types. Of course, the fact that the type-checker needs to compute unions, intersections, and negations of types does not mean that we need to introduce these operations in the syntax of the types (namely, in the type system): they could be meta-operations whose usage is confined to the type-checker. This is for instance the choice of XDuce (or of XQuery whose types borrow many features from XDuce's ones), where only union types are included in the type system (they are needed to define regular expression types), while intersections and negations are meta-operations computed by the subtyping algorithm.

We defend a choice different from XDuce's one and think that unions, intersections, and differences must be present at type level since, we argue, having these type constructors in the type system is useful for programming[5] This is particularly true for programs that manipulate XML data—as we will see next— in particular if we completely embrace the "pattern as types" analogy of Remark 1. We have seen that in the "pattern as types" viewpoint, the pattern `(`nil,_)` is matched by all values that *have type* `(`nil,_)`. This pattern is built from two very specific types: a singleton type and the "_" type. In order to generalise the approach, instead of using in patterns just singleton and "_" types, let us build patterns using as building blocks generic types. So, to give some examples, let us write patterns such as `(x,Int)` which captures in `x` the first projection of the matched value only if the second projection is an integer; if we want to capture in `y` also the second projection, it suffices to use an intersection: `(x,y&Int)` as to match an intersection a value must match both patterns (the variable to capture and `Int` to check the type); if the second projection of the matched value is an integer, then `(x,y&Int)|(_,x&y)` will capture in `x` the first projection and in `y` the second projection, otherwise both variables will capture the second projection.

We can then write the pattern `(x & (Car&(Guarantee|(_\Used))))` that captures in `x` all cars that, if they are used have a guarantee (these properties being expressed by the types `Car`, `Guarantee`, and `Used`, the type `_\Used` being equivalent to `¬Used`) and use it to select the wanted items in a catalogue by a select-from-where expression. We have seen at the very beginning of Section 2.1 that patterns can be used instead of variables in let bindings. The idea underlying ℂQL [5] is to do the same with the bindings in the `from` clause of a select-from-where. So if `catalogue` denotes a sequence of items, then we can select from it the cars that if used then have a guarantee, by

```
select x from
   (x & (Car&(Guarantee|(_\Used))) in catalogue
```

As customary, the select-from-where iterates on all elements of `catalogue`; but instead of capturing every element, it captures only those elements that match the pattern, and

[5] From a theoretical point of view the XDuce's choice is justified by the fact that XDuce types are closed with respect to boolean operations. This is no longer true if, as in ℂDuce, one also has function types. However, the point we defend here is that it is useful in practice to have all the boolean operations in the syntax of types, even in the presence of such closure properties.

then binds the pattern variables to their corresponding subparts. These variables are then used in the select and in the subsequent "from" and "where" clauses to form the result. In some sense, the use of patterns in the from clauses corresponds to a syntactic way to force the classic logical optimisation of remounting projections, where here the projection is on the values that match the pattern (we say more about it in Section 3.5). The general form of the select-from-where we will consider here is then the one of ℂQL, namely:

```
select e from
    p in e',...,p in e'
where e''
```

where p are patterns, e' expressions that denote sequences, and e'' a boolean expression. The select iterates on the e' sequences capturing the variables in the patterns only for the elements that match the respective pattern and satisfy the condition e''. Note that the usual select-from-where syntax as found in SQL is the special case of the above where all the patterns are variables. The same of course holds true also for the FLWR expressions of XQuery, which are nothing but a different syntax for the old select expression (the let binding would appear in the e expression following the select).

The select-from-where expressions we just introduced is nothing but a query-oriented syntax for *list comprehensions* [56], which are a convenient way to define a new list in terms of another list. As discussed by Trinder and Wadler [55] a list comprehension is an expression of the form $[e \mid p \leftarrow e'\,; c]$ where e is a list expression, p a pattern, e' a generic expression, and c a boolean condition; it defines the list obtained by evaluating e' in the environment produced by matching an element of the result of e against p provided that in the same environment the condition c holds. It is clear that the expression above is just a different syntax for `select` e' `from` p `in` e `where` c, and that the general case with several from clauses is obtained by nesting list comprehensions in e'.

2.3 Regular Expression Patterns

If we want to use patterns also to manipulate XML data, then the simple and somehow naive approach is to define XML patterns as XML values where capture variables and wildcards may occur. To the best of our knowledge, this was first proposed in the programming language community by XMλ [47] and in the database community by XML-QL [25] (whose query primitive is a simpler version of the pattern-based select-from-where introduced in the previous section). This corresponds to extend the classic "patterns as values with variables (and wildcards)" analogy of functional languages to XML data. However, in the previous sections we introduced a more expressive analogy, the one of "patterns as *types* with capture variables" as stated in Remark 1. Since values denote singleton types, then the latter analogy extends the former one, and so it is in principle more expressive. The gain in expressiveness obtained by using the second analogy becomes clear also in practice as soon as we deal with XML, since the types for XML can be more richly combined than those of classic functional languages. Indeed, XML types are usually defined by regular expressions. This can be best shown by using the paradigmatic example of bibliographies expressed by using the ℂDuce (and ℂQL) type syntax:

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

The declarations above should not pose any problem to the reader familiar with XML, DTD, and XML Schema. The type `Bib` classifies XML-trees rooted at tag `bib` that delimits a possibly empty list of books. These are elements with tag `book`, an attribute `year`, and containing a sequence formed exactly by one element title, followed by either a non empty list of author elements, or a non empty list of editor elements, and ended by an optional element price. Title elements are tagged by `title` and contain a sequence of characters, that is, a string (in XML terminology "parsed character data", PCDATA). The other declarations have similar explanations.

We used the ℂDuce syntax (which slightly differs from the XDuce's one for tags and attributes) foremost because it is the syntax we are most familiar with, but also because ℂDuce currently possesses the richest type and pattern algebras among the languages that use regular expression patterns.

The declarations above give a rather complete presentation of ℂDuce types. There are XML types, that are formed by a tag part and a sequence type (denoted by square brackets). The content of a sequence type is described by a regular expression on types, that is, by the juxtaposition, the application of `*`, `+`, `?` operators, and the union `|` of types. Besides these types there also are all the type constructors we saw earlier in this section, namely: (*i*) values which are considered singleton types, so for instance `"Buneman"` is the type that contains only the string `"Buneman"`, (*ii*) intersection of types, denoted by *s*&*t* that contains all the values that have both type *s* and type *t*, (*iii*) difference "\" of types, so that `<book year=String\"1999">[Title (Author+|Editor+) Price?]` is the type of all books *not* published in 1999, (*iv*) the "_" type, which is the type of all values and is also noted `Any`, and its complement the `Empty` type.

According to Remark 1, patterns are the above types enriched with capture variables. With respect to XMλ's approach of "patterns as values with capture variables", this approach yields regular expression patterns. For instance, `<bib>[(x::Book)*]` is a pattern that captures in `x` the sequence of all books of a bibliography. Indeed, the `*` indicates that the pattern `x::Book` must be applied to every element of the sequence delimited by `<bib>`. When matched against an element, the pattern `x::Book` captures this element in the sequence `x`, provided that the element is of type `Book`.[6] Patterns can then be used in match expressions:

```
 match biblio with   <bib>[ (x::Book)* ] -> x
```

[6] The reader may have noticed that we used both `x&`*t* and `x::`*t*. The double semicolon indicates that the variable `x` captures *sequences* of *t* elements: while the first `x` is of type *t* the second one is of type `[`*t*`*]`. Since inside regular expression patterns, variables capture sequences, then only the latter can be used (see the formal syntax at the beginning of Section 2.4).

This expression matches `biblio` against our pattern and returns `x` as result, thus it makes nothing but stripping the `<bib>` tag from `biblio`. Note that if we knew that `biblio` has type `Bib`, then we could have used the pattern `<bib>[(x::_)*]` (or, equivalently, `<bib>[(x::Any)*]`), since we statically know that all elements have type `Book`.

Besides capture variables there is just one further difference between patterns and types, namely the union operator `|`, which is commutative for types while it obeys a first match policy in patterns. So for instance the following expression returns the sequence of all books published in 1999:

```
match biblio with <bib>[( (x::<book year="1999">_) | _ )*] -> x
```

Again, the pattern `((x::<book year="1999">_) | _ )` is applied to each element of the sequence. This pattern first checks whether the element has the tag `<book year="1999">` whatever its sequence of elements is, and if such is the case it captures it in `x`; otherwise it matches the element against the pattern "`_`", which always succeeds without capturing anything (in this way it discards the element). Note that, if we had instead used `<bib>[ (x::<book year="1999">_)* ]` this pattern would have succeeded only for bibliographies composed only by books published in 1999, and failed otherwise.

As we said in the introduction, an extraction primitive must be coupled with iterator primitives to apply the extraction all over the data. There are several kinds of iterators in ℂDuce, but for the sake of the presentation we will use the one defined in the query sub-language ℂQL, that is the select-from-where defined in the previous section. So, for example, we can define a function that for each book in a sequence extracts all titles, together with relative authors or editors.

```
fun extract(x : [Book*]) : [ [Title (Author+|Editor+)]* ] =
   select (flatten[z y]) from
      <book ..>[ z::Title  y::(Author|Editor)+  _* ] in x
```

The function `extract` takes a possibly empty sequence of books and returns a possibly empty sequence of sequences that start by a title followed by a non-empty uniform sequence of authors or editors. The operator `flatten` takes a sequence of sequences and returns their concatenation, thus `flatten[z y]` is nothing but the concatenation of the sequences `z` and `y`. The select-from-where applies the pattern before the `in` keyword to each element of the sequence `x` and returns `flatten[z y]` for every element matching the pattern. In particular, the pattern captures the title in the sequence variable `z`, the sequence of authors or editors in the sequence variable `y`, and uses "`..`" (the wildcard that matches any set of attributes) to discard the `year` attribute. Had we wanted to return a sequences of pairs (title,price), we would have written

```
fun extract2(x : [Book*]) : [ (Title,Price)* ] =
   select (t,p) from
      <book ..>[ t&Title  _*  p&Price ] in x
```

where we used "`&`" instead of "`::`" to denote that variables capture single elements rather than sequences (see Footnote 6).

Both examples show one of the advantages of using patterns, that is the ability to capture different subparts of a sequence of elements (in the specific case the title and the authors/editors) in a single pass of the sequence.

The select-from-where expression is enough to encode XPath-like navigation expressions. In ℂDuce/ ℂQL one can use the expression e/t, which is syntactic sugar for `flatten(select x from <_ ..>[(x::`t`|_)*] in `e`)` and returns all children of type t of elements of the sequence e. We can compose these expressions to obtain an XPath-like expression. For instance, if `bibs` denotes a sequence of bibliographies (i.e., it is of type `[Bib*]`), then `bibs/Book/(Author|Editor)` returns the sequence of all authors and editors appearing in the bibliographies. In order to match more closely the semantics of XPath, we will often write `bibs/<book ..>_/(<author ..>_|<editor ..>_)`, since this kind of expression checks only tags while, in principle, the previous path expression checks the whole type of the elements (not just the tag). As a matter of fact, by using the static type of `bibs`, ℂDuce compiles the first expression into the second one, as we will explain in Section 3.4. Similarly, the expression e`/@`id is syntactic sugar for `select x from <_ `id`=x ..> in `e, which returns all the values of the attribute id occurring in the children of elements in e. One can combine select-from-where and path expressions and write queries such as

```
select name from
  <book year="2005">[Title  a::Author+  <price>p] in bibs/Book,
  name in a/<last>(String\"anonymous")
where int_of(p) << 100
```

which returns the sequence of last name elements of the authors of all the books in `bibs` published this year for which a list of authors is given, a price lower than 100 is specified, and the last name is different from "anonymous". The reader is invited to verify that the query would be far more cumbersome if we had to write it either using only patterns or, as it would be for XQuery, using only paths (in this latter case we also would need many more nested loops). A pattern-based encoding of XPath completely different from the one presented here has been proposed by Gapeyev and Pierce [32].

2.4 Pattern and type algebras

Patterns and types presented in the previous sections are summarised below:

Types
$t ::= b \mid t|t \mid t\&t \mid t\backslash t \mid (t,t) \mid$ `<`t ℓ=t...ℓ=t`>`t $\mid$ `[`R`]` $\mid$ `Empty` $\mid$ `Any`

Type regular expressions
$R ::= t \mid R\,R \mid R|R \mid R* \mid R+ \mid R?$

Patterns
$p ::= x \mid t \mid p\&p \mid p\,|\,p \mid (p,p) \mid$ `<`p ℓ=p ... ℓ=p`>` p $\mid$ `[`r`]`

Pattern regular expressions
$r ::= p \mid x::r \mid r\,|\,r \mid r\,r \mid r+ \mid r* \mid r?$

where b ranges over basic types, that is `Char`, `Int`, etc., as well as singleton types (denoted by values). As a matter of fact, most of the syntax above is just syntactic sugar. These types and patterns can indeed be expressed in a much more compact system, composed only the very simple constructors we started from: basic and product types and their boolean combination. In this short section we will deal with more type-theoretic aspects and give an outline of the fact that all the patterns and types above can be expressed in the system defined as follows.

Definition 1. *A* type *is a possibly infinite term produced by the following grammar:*

$$t \ ::= \ b \mid (t_1,t_2) \mid t_1|t_2 \mid t_1\&t_2 \mid \neg t \mid \mathbb{0} \mid \mathbb{1}$$

with two additional requirements:

1. *(regularity) the term must be a regular tree (it has only a finite number of distinct sub-terms);*
2. *(contractivity) every infinite branch must contain an infinite number of pair nodes* (t_1,t_2).

A pattern *is a type in which (possibly infinitely many) occurrences of finitely many capture variables may appear anywhere provided that*

1. *no variable occurs under a negation,*
2. *patterns forming an intersection have distinct sets of occurring variables,*
3. *patterns forming an union have the same sets of occurring variables.* □

In the definition b ranges again overs basic types and $\mathbb{0}$ and $\mathbb{1}$ respectively represent the `Empty` and `Any` types. The infiniteness of types/patterns accounts for recursive types/patterns. Of course these types must be machine representable, therefore we impose a condition of regularity (in practice, this means that we can define types by recursive equations, using at most as many equations as distinct subtrees). The contractivity condition rules out meaningless terms such as $X = \neg X$ (that is, an infinite unary tree where all nodes are labelled by $\neg$). Both conditions are standard when dealing with recursive types (e.g. see [2]). Also pretty standard are the conditions on the capture variables for patterns: it is meaningless to capture subparts that do not match (one rather captures parts that match the negation); in intersections both patterns must be matched so they have to assign distinct variables, while in union patterns just one pattern will be matched so always the same variables must be assigned whichever alternatives is chosen.

Definition 1 formalises the intuition given in Remark 1 and explains why in the introduction we announced that patterns and types we were going to present were closely connected.

These types and patterns (which are the *semantic subtyping* based ones hinted at in the introduction) are enough to encode all the regular expression types and patterns we used in Section 2.3 (actually, they can do much more than that): sequences can be encoded *à la* Lisp by pairs, pairs can also be used to encode XML types, while regular expression types are encoded by recursive patterns. For instance, if we do not consider attributes, the type

```
type Book = <book>[Title (Author+|Editor+) Price?]
```

can be encoded as $Book = (\text{`}book, (Title, X|Y))$, $X = (Author, X|(Price, \text{`}nil)|\text{`}nil)$ and $Y = (Editor, Y|(Price, \text{`}nil)|\text{`}nil)$, where *‘book* and *‘nil* are singleton (basic) types. More details about the encoding, such as the use of non-linear capture variables to match sequences and the use of record patterns to match attributes, as well as the formal definition of pattern matching are given elsewhere [4].

The core syntax of the semantic types and patterns is very simple and this turns out to be quite useful in the formal treatment of the system. The other characteristic that makes life easier is that the boolean combinators on types are interpreted set theoretically: types are sets of values and intersection, union, and difference of types are interpreted as the corresponding operators on these sets (this is the essence of the semantic subtyping approach). Although this simplifies significantly both the theoretical development (types can be transformed by using the classical set-theoretic laws, e.g. De Morgans's, etc.) and the practice (e.g. for a programmer it is easier to understand subtyping in terms of set containment, than in terms of an axiomatisation), the development of the underlying theory is quite complex (see for instance [19, 37, 35, 34, 38, 30, 29, 39]). Fortunately, this complexity is hidden from the programmer: all (s)he has to know is that types are set of values and subtyping is set inclusion. Such theoretical complexity is the counterpart of the expressiveness of the system; expressiveness that is manifest in the simplicity of the query language: value constructors (constants, pairs, and XML values), operators for basic types (e.g. arithmetic and boolean operators), the `flatten` operator, and the pattern-based select-from-where (i.e., list comprehensions) constitute the complete definition of the ℂQL query language [5]. We are in the presence of few primitives that permit to query complex data in XML format: of course the power comes from the use of patterns in the select-from-where expressions.

2.5 Iterators

In the introduction we said that in order to manipulate XML data besides extraction primitives we also need iterators. Therefore, let us spend a final word about them. In the previous section we used just one iterator, the select-from-where expression. This iterator is very simple but not very expressive: it cannot transform complex trees but just query them (it returns sequences not whole trees) and it applies just one pattern to each element of the scanned sequences (while we have seen that the power of pattern matching resides in the possibility of trying several alternative patterns on the same element). Of course, a select-from-where is meant to be so: it is a query primitive, not a transformation primitive. Therefore it was designed to be simple and not very expressive in order to be easily optimisable (see Section 3.5). But if we want to define concisely more complex transformations, then the language has to provide more powerful built-in operators. For instance, the ℂDuce language provides three different iterators: the `map` constructor, whose syntax is `map` e `with` p_1`->`e_1 | ... | p_n`->`e_n, which applies the specified matching alternatives to each element of the sequence e and returns the sequence of results; the `transform` constructor which acts like map but filters out elements that are not matched; the `xtransform` constructor which performs the same operation but on trees, leaving unmatched subtrees unmodified. Of course, the same behaviour could be obtained by programming these operators by using functions but, as we explain in Section 3.6, we would not obtain the same precision of type checking as we obtain by hard-coding them as primitive constructions of the language.

3 Eight good reasons to use regular expression patterns and types in query languages

As its title clearly states, in this section we try to advocate the use of regular expression patterns and of the union, intersection, and negation types we introduced in the previous section. A first reason to be interested in this approach is that its theoretical core is very compact (which does not mean "simple"): we have seen that an expressive query language can be simply obtained by adding list comprehensions to the types and patterns of Definition 1. Rather than concentrating on the theoretical aspects, in the rest of this work we will focus on more practical issues. We already said that we are not promoting the use of regular expressions to the detriment of path expressions, but we would like to invite the database and programming language community to share their skills to find a suitable way to integrate the two mechanisms. Since the author belongs to latter community this paper has described up to now the pattern approach and will now try to address some points which, hopefully, should interest the database community as well: we apologise in advance for the naiveties that might slip in such a *démarche*.

3.1 Classic usage

The most obvious usages of the type system presented here are those typical of every type system: e.g. static detection of type errors, partial correctness, and database schema specification. In this respect, semantic types do not differ significantly from other type systems and we will no spend much time on this aspect.

The only point that is worth noticing is that union, intersection, and difference types, form quite a natural specification language to express schema constraints. This looks particularly interesting from the database perspective, in particular for the definition of different views. Notwithstanding that the specification of complex views requires complex queries, union intersection and negation types constitute a powerful specification language for simple views. Defining views by restriction or extension looks like a natural application of boolean combinators of types. To give a naive example define the following types

```
type WithPrice = <_ ..>[_* Price _*]
type ThisYear = <_ year="2005">_
```

The first is the type of every element (whatever its tag and attributes are) that has at least a child element of type `Price`, the second types every element (whatever its tag and its content is) that has an attribute year equal to "2005". We can then use the type `<bib>[((Biblio&ThisYear)\WithPrice)*]` to specify a view of our bibliography containing only those books published in 2005 that do not have a price element.

3.2 Informative error messages

The use of boolean combinators for types is quite useful in producing informative error messages at compile time. When type checking fails it is always because the type-checker was expecting an expression of some type s and found instead an expression of a type t that is not a subtype of s. Showing the two types s and t is not always informative enough to help the programmer to find the error, especially in case of XML data where s and t can be quite complex (just think of the type describing XHTML documents). Thanks to boolean combinators of types we can compute the difference of these two types, $t\backslash s$, inductively generate a sample value belonging to this difference, and return it to the programmer. This value is a witness that the program is ill-typed, and the generation of just enough of the sample value to outline the error usually allows the programmer to rapidly localise the problem.

To give a practical example of this fact, imagine we want to define a function that returns the list of books of a given year, stripped of the Editors and Price elements. Consider the following solution:

```
fun onlyAuthors (year :Int , books :[Book*]) :[Book*] =
   select <book year=y> (flatten[ t a ]) from
      <book year=y>[ (t::Title | a::Author | _)+] in books
   where int_of(y) = year
```

The idea is that for each book the pattern captures the year in `y`, the title in the sequence variable `t`, and the sequence of authors in `a`. Then, the expression preceding the from clauses rebuilds the book by concatenating the sequences stored in `t` and `a`, provided that the year is the one specified at the argument of the function. The function above is not well-typed and the ℂDuce compiler returns the following error message

```
Error at chars 81-95:
   select <book year=y> ( flatten[ t a ]) from
This expression should have type:
[ Title Editor+ | Title Editor+ Price | Title Author+ | Title Author+ Price ]
but its inferred type is:
[ Title Author+ | Title ]
which is not a subtype, as shown by the sample:
[ <title>[ ] ]
```

The sample value at the end of the message shows at once the origin of the problem: the expression `flatten[ t a ]` outlined in the error message (i.e., the expression at chars 81-95) may return a sequence that contains just a title, but no author or editor. This allows the programmer to understand that the problem is that `a` may denote the empty sequence (the case in which a book specifies a list of editors) and, according to the intended semantics of the program, make her/him correct the error by modifying either the return type of the function (i.e., `[(<book year=String>[Title Author*])*]`), or the pattern (typically, a pattern like `<book year=y>[ t::Title a::Author+ _*]`).

Of course in such a simple example the expected and inferred types would have been informative enough: it is easy to see that the former type in the error message is equivalent to `[Title (Author|Editor)+ Price?]` while the latter is `[Title Author*]` and hence to arrive to the same conclusion. But in practice types are seldom so simple and from our experience in programming with ℂDuce we have found that sample values in error messages play an essential role in helping the programmer to rapidly spot where bugs lie. We invite the reader to verify this claim by trying the ℂDuce online interpreter at `www.cduce.org`.

3.3 Error mining

Patterns and types are powerful enough to spot some subtle errors that elude current type checking technology. Suppose we had programmed the function `extract` of Section 2.3 as follows

```
fun extract(x : [Book*]) : [ [Title (Author+|Editor+)]* ] =
   select (flatten[z y]) from
      <book ..>[ z::Title  y::(<author>_|<edtor>_)+  _* ] in x
```

Note that despite the typo we outlined in bold in the program, the function above is well-typed: no typing rule is violated and the pattern is not a useless one since it can still match authors. However, all the books with editors would be filtered out from the result. Since there are cases in which the pattern matches, a possible static emptiness check of the result (as, for instance, recommended in Section 4, "Static Type Analysis" subsection of the XQuery 1.0 and XPath 2.0 Formal Semantics[7]) of would not uncover the error. Such an error can only be detected by examining the result and verifying that no book with editors appear. This kind of error is not the exclusive resort of patterns, but can happen also with paths. For instance, if we want to extract each title together with the relative price, from our bibliographic collection `bibs` we can write

```
bibs/<book ..>_/(<title>_|<prize>_)
```

which contains an error, as `prize` occurs instead of `price`. But since the result is not always empty no warning is raised. Again, the error is hidden by the fact that the pattern is partially correct: it does find some match, even if, locally, `<prize>_` never matches, hence is incorrect. Once more, as price is optional, by looking at the query output, when seeing only titles, we do not know whether prices are not present in that database or something else went wrong.

These errors can be roughly characterised as the presence of dead code in extraction primitives, that is, the presence of subcomponents (of the patterns or paths) that have no chance to match data. The presence of such errors is very likely in writing programs that process typed XML data, since programmers tend to specify only the part of the schema that is strictly necessary to recover desired data. To that end they make extensive usage of wildcards and alternations that are an important (but not exclusive) source of this kind of errors.

The consequence of these errors is that some desired data may end up not contributing to partial and/or final results, without having the possibility of becoming aware of this problem at compile time. So, this problem may be visible only by carefully observing the results of the programs. This makes error detection quite difficult and the subsequent debugging very hard. And it is made even harder by the fact that, as argued in [18], such errors are not just created by typos—as shown here—but they may be of more conceptual nature.

It has been shown [18] that the errors of this kind can be formally characterised and statically detected by using the set-theoretic operators of the types and patterns we presented here. In particular given a type t and a pattern p, it is not difficult to

[7] See `http://www.w3.org/TR/2005/WD-xquery-semantics-20050915/#processing_static`.

characterise the parts of p which are used for at least one value v in t (and hence the dead parts that are never used). This is done by applying a rewriting system to the pair (t,p) which decomposes the matching problem for each subcomponent of p, by applying the set-theoretic properties of the semantic types and patterns. So for instance $(t,p_1|p_2)$ is rewritten into (t,p_1) and $(t\mathbf{\&}\neg\llparenthesis p_1\rrparenthesis,p_2)$; the set of sub-patterns of p that may be used when matching values of type t is formed by all patterns p' such that (t,p) rewrites in zero or more steps into (t',p') and $t'\mathbf{\&}\llparenthesis p'\rrparenthesis = \mathbb{0}$.

Finally, the implementation of such a technique in the current type-checkers of, among others, Xtatic, ℂDuce, and XDuce, does not produce any noticeable overhead, since the rewriting can be performed by the normal type inference process itself. Further details are available elsewhere [18].

3.4 Efficient execution

The benefits of semantic patterns/types are not confined to the static aspects of XML programming. On the contrary they are the key ingredient that makes languages as ℂDuce and Xtatic outperform the fastest XML processors. The idea is quite simple: by using the static type information and the set-theoretic properties of the semantic patterns/types one can compile data queries (e.g. the patterns) so that they perform a minimum number of checks. For instance, if we look in an XML tree for some given tag and the type of the tree tells us that this tag cannot occur in the left subtree, then we will skip the exploration of this subtree and explore only the right one. As a more concrete example consider the following definitions (see Footnote 3 for notation)

```
type A = <a>[A*]
type B = <b>[B*]

fun check( A|B -> Int ) A -> 1 |  B -> 0
```

The type `A` types all the XML trees where only the `<a>` tags occurs, the type `B` does the same for `<b>`, while the function `check` returns either 1 or 0 according to whether its argument is of type `A` or `B`. A naive compilation schema would yield the following behaviour for the function: first check whether the first pattern matches the argument, by checking that all the elements of the argument are `<a>`; if this fails, try the second branch and do all these tests again with `<b>`. The argument may be run through completely several times. There are many useless tests: since we statically know that the argument is forcedly either of type `A` or of type `B`, then the check of the root tag is enough. It is thus possible to use the static type information to compile pattern matching so that it not only avoids backtracking but it also avoids checking whole parts of the matched value. In practice `check` will be compiled as

```
fun check( A|B -> Int ) <a>_ -> 1 |  _ -> 0
```

As a second example consider the query at the end of Section 2.3. By using the information that `bibs` has static type `[Bib*]`, it will be compiled as:

```
select name from
   <_ year="2005">[ _  a::Author+  <price>p ] in bibs/_,
   name in a/<last>(_\"anonymous")
where int_of(p) << 100
```

While in both cases the solutions are easy to find, in general computing the optimal solution requires fully exploiting intersections and differences of types. These are used to reduce the problem of generating an optimal test to that of deciding to which summand of a union of pairwise disjoint types the values of a given static type belong to. To find the solution, the algorithm—whose description is outside the scope of this paper (see the references below)—descends deep in the static type starting from its root and accumulates enough information to stop the process as soon as possible. The information is accumulated by generating at each step of the descent a new union of pairwise distinct types, each type corresponding to a different branching of the decision procedure.

This algorithm was first defined and implemented for ℂDuce and it is outlined in [4] (whose extended version contain a more detailed description). The tree-automata theory underlying has been formally described [28] and generalised [30]. Levin and Pierce have adapt this technique to Xtatic and extend it with heuristics (their work is included in these proceedings [43]).

We just want to stress that this compilation schema is semantic with respect to types, in the sense that the produced code does not depend on the syntax of the types that appear in patterns, but only on their interpretation as sets of values. Therefore there is no need to simplify types—for instance by applying any of the many type equivalences—before producing code, since such simplifications are all "internalised" in the compilation schema itself.

The practical benefits of this compilation schema have been shown [5] by using XMark [52] and the XQuery Use Cases [20] to benchmark ℂDuce/ ℂQL against Qizx [27] and Qexo [9] two of most efficient XQuery processors (these are several orders of magnitude faster than the reference implementation of XQuery, Galax [3]). The results show that in main memory processing ℂQL is on the average noticeably faster than Qizx and Qexo, especially when computing intensive queries such as joins. Furthermore, since the execution times of ℂQL benchmarks always include the type-checking phase, this also shows that the semantic types presented here are algorithmically tractable in practice.

3.5 Logical optimisation of pattern-based queries

We already remarked at the end of Section 2.2 that the usual select-from-where as found in SQL and the for-expressions of XQuery are both special cases of our pattern-based select-from-where expressions, in which all patterns are variables. Hence, all classic logical optimisations defined for the former apply also to the pattern-based case. However, the use of patterns introduces a new class of pattern-specific optimisations [5] that being orthogonal to the classical optimisations bring a further gain of performance. These optimisations essentially try to transform the from clauses so as to capture in a single pattern as much information as possible. This can be obtained essentially in three ways: (*i*) by merging into a single pattern two different patterns that work on a common sequence, (*ii*) by transforming parts of the where clauses into patterns, and (*iii*) by transforming path expressions into nested pattern-based selections and then merging the different selects before applying the previous optimisations. As an example consider

```
select <book>[t]  from
   b in bibs/<book>_ ,
   p in [b]/<price>_ ,
   t in [b]/<title>_ ,
   y in [b]/@year
where (p = <price>"69.99") and (y="1990")
```

which is a query written in a XQuery style (the from clauses use path expressions on right of the `in` keyword and single variables on its left), and that returns all the titles of books published in 1990 whose price is "69.99" (this essentially is the query Q1 of the XQuery Use Cases). After applying pattern-specific optimisations it will be transformed into

```
select <book>[t]  from
   <bib>[b::Book*] in bibs,
   <book year="1990">[ t&Title  _+  <price>"69.99" ] in b
```

which is intuitively better performing since it computes less nested loops.

The benchmarks mentioned above [5] show that these pattern-specific optimisations in most cases bring a gain in performance, and in no case degrading it.

3.6 Pattern matches as building blocks for iterators

In the introduction we said that in order to work with XML data one needs two different primitives: deconstructors and iterators. Although patterns belong to the first class of primitives, thanks to an idea of Haruo Hosoya, they are useful to define iterators, as well. More precisely, they allow the programmer to define her/his own iterators. This is very important in the context of XML processing for two reasons: (i) the complex structure of data makes virtually impossible for a language to provide a set of iterators covering, in a satisfactory way, all possible cases[8] and (ii) an iterator programmed using the existing primitives of the language would be far less precisely typed than the same built-in operator and would thus require a massive usage of casting operations.

We have seen that by defining regular expressions over patterns we can perform data extraction along sequences of elements. But patterns play a passive role with respect to the elements of the sequence: they can capture (part of) them but do not compute any transformation. Haruo Hosoya noticed that if instead of using patterns as basic blocks of regular expressions one uses pattern matching branches of the form "*p* -> *e*", then it is possible to define powerful iterators that he dubs *filters* [36] and that are included in the recent versions of XDuce. The idea is that as regular expressions over patterns describe the way the patterns are matched against the elements of a sequence, in the same way regular expressions over match branches "*p* -> *e*" describe the way to

[8] No formal expressiveness concern here: just programming experience where the need of a new iterator that would fit and solve the current problem appears over and over.

apply the transformation described by the branch to the elements of a sequence provided that they match p. For instance, the filter[9] `filter[(x&Int -> x+1)*]` applied to `[1 2 3]` returns `[2 3 4]`, while `filter[( x&Int->x+1 | x&Bool->not(x) )*]` transforms `[1 ‘true ‘true 2]` into `[2 ‘false ‘false 3]`. To show a filter in our paradigmatic example consider the following `translate` filter

```
type Titre = <titre>[PCDATA]
type Auteur = <auteur edt=("oui"|"non")>[Last First]

let translate = filter[
      ( <title> x -> <titre> x
      | <author> x -> <auteur edt="non"> x
      | <editor> x -> <auteur edt="oui"> x
      | x -> x )*]
```

which transforms title elements to their French translation, author and editor elements into “auteur” elements, and leaves other elements unchanged. This filter can then be applied to the content of a book element to obtain new elements which will have type `<book year=String>[Titre Auteur+ Price]`.

More generally, filters provide a unique mechanism to implement several primitives. For instance, `match` e `with` p_1`->`e_1 `|` ... `|` p_n`->`e_n is just syntactic sugar for the application `filter[`p_1`->`e_1 `|` ... `|` p_n`->`e_n`]([`e`])`, while a filter such as `filter[(`p_1`->`e_1 `|` ... `|` p_n`->`e_n`)*](`e`)` corresponds to the `transform` iterator of CDuce we hinted at in Section 2.5. Filters can also encode the `xtransform` iterator of Section 2.5 (this is a little clumsier, since it requires the use of recursive filters). The typing of XDuce filters is less precise than the one of map, transform and xtransform, but in exchange filters can do more as they can process several elements at a time while map, transform and xtransform can just process a single element per iteration (see [36] for details).

We already explained that the reason why we need to give the programmer the possibility to define iterators is that built-in iterators cannot cover all the possible cases and that iterations implemented via functions would not be typed precisely enough. To see why consider the filter `translate` we defined before, and notice that the type-checker must be able to deduce that when this filter is applied to a sequence of type `[Title (Author+|Editor+) Price?]`, then the result has type `[Titre Auteur+ Price?]`, while when the same filter is applied to a sequence of type `[Author* Editor]` the type-checker must deduce a result type `[Auteur+]`. This kind of polymorphism goes beyond the possibilities of parametric polymorphism of, say, ML or System F, which can be applied only to homogeneous lists. Here instead the result type is obtained by performing an abstract execution of the iterator on the type of the input. In practice, what the type-checker does is to execute the iterator on the DTD of the input in order to precisely map the transformation of each element of the input tree in the resulting output tree. This justifies the use of a specific syntax for defining iterators, since this sub-language instructs the type-checker to perform this abstract execution, makes it possible, and ensures its termination.

The expressive power of Hosoya’s filters is limited, as they rely on regular expressions. The kind of processing that these filters permits is, roughly, that of the map oper-

[9] We use for filters a syntax slightly different from the one of XDuce.

ator in functional languages. But, for instance, they are not able to express the function that reverses a list. Also, type inference is less precise than that of map, transform, and xtransform, and it is further penalised in the presence of recursion. To obviate all these problems Kim Nguyễn has proposed a more radical approach [50]. He proceeds along the ideas of Haruo Hosoya and takes as basic building blocks the pattern matching branches, but instead of building his filters by defining regular expressions on these building blocks, Nguyễn's filters are obtained by applying the grammar of Definition 1 to them. His filters are then regular trees generated by the following grammar:

$$f \ ::= \ e \ | \ p \rightarrow f \ | \ (f,f) \ | \ f|f \ | \ f;f$$

where e ranges over expressions and p over patterns. Their semantics is quite natural. When a filter formed by just an expression is applied, then the expression is executed. If the filter $p \rightarrow f$ is applied to e, then the filter f applied to e is executed in the environment obtained by matching p against the result of e (failure of this matching makes the whole filter fail); if (f_1, f_2) is applied to a pair, then each f_i is applied to the corresponding element of the pair and the pair of the results is returned; the alternation applies the first filter to the argument and if this fails it applies the second one; finally, the sequencing applies the second filter to the result of the application of the first filter to the argument. Note that with respect to the grammar in Definition 1, sequencing plays the role of the intersection. Furthermore, as in Definition 1 there cannot be any capture variable under a negation, so there is no negation filter in Nguyễn's filters. A limited form of recursion ensures the termination of the type-checking and of the execution of Nguyễn's filters.

As simple as they are, Nguyễn's filters have many of the sought properties: they are expressive enough to encode list reversal, while the encodings of map, transform, and xtransform have same precise typing as in ℂDuce. The algorithmic properties of their type system are (at the moment of writing) still a matter of study.

3.7 Type and pattern-based data pruning for memory usage optimisation

XML data projection (or pruning) is one of the main optimisation techniques recently adopted in the context of main-memory XML query-engines [46, 10]. The underlying idea is very simple but useful at the same time. In short, given a query q over a document d, subtrees of d not necessary for evaluating q are pruned, thus obtaining a smaller document d'. Then q is executed over d', hence avoiding the need to allocate and process nodes that will never be reached by navigational specifications in q. As has been shown [46, 10], in general, XML navigation specifications expressed in queries tend to be very selective. Hence, significant improvements due to pruning can be actually obtained, either in terms of query execution time or in term of memory usage (it is worth observing that for main-memory XML query engines, very large documents can not be queried without pruning).

The work we have described in Section 3.4 already provides optimal data pruning for pattern matching. Even if the actual implementation relies on automata, we have seen that it essentially consists in computing a set of equivalent minimal patterns. The "minimality" is given by the presence of "_" wildcards that denote parts of the data that need not to be checked. It is clear that the set of the parts denoted by "`_`" and not in the scope of any capture variable constitutes an optimal set of data to be pruned.

Of course, extending the technique developed for the compilation of patterns to general and more complex queries (i.e. not just consisting of simple pattern matching) requires further work. But this does not seems a far-reached objective: once more set-theoretic operations should came to rescue, as the process should essentially reduce to the computation of the intersections of the various optimal patterns met in the query, and its distribution with respect to the data stored in secondary memory, so that to individuate its optimal pruning set.

3.8 Type-based query optimisation

Let us conclude this overview by a somewhat vague but nevertheless important remark. The type system presented in this work is very precise.

Concerning data description the XML components of Xtatic, XDuce, and ℂDuce's type systems are already more expressive than DTDs and XML Schemas. This holds true from a formal point of view (see [48] for a formal taxonomy) but, above all, in practice, too.[10] For the practical aspect let us take a real life example and consider the Internet Movie Database [42] whose XML Schema cab be found elsewhere [8]: thanks to singleton types and boolean combinators, ℂDuce types can express integrity constraints such as the fact that the `type` attribute of `show` elements can only be either `"Movie"` or `"TV Series"`, and that only in the former case the `show` element can contain a `box_office` element, and only in the latter case it can contain a `season` element.

But the real boost with these new types is when typing queries and transformations, since semantic types are far more precise than the type-systems of current languages for XML. We have already seen in Section 3.6 the stringent requirements for typing iterators: the type-checker must be able to compute the precise modifications performed on each element the iterator meets. For instance the system presented here will deduce for `bibs/Book/(Title|Author|Editor)` the type `[(Title (Author+|Editor+))*]` while, at best, the corresponding XPath expression will be typed by the XQuery type system as `[(Title|Author|Editor)*]`, which is far less precise. A further example of precision of the type system is the typing of pattern variables which is *exact*, in the sense that the type inferred by the type-checker for a pattern variable is exactly the set of values that may be assigned to that variable at run-time. The precision of the type system is also witnessed by the practice, since in languages that use these semantic types and patterns, downward casts are essentially confined to external data, while internal data (that is, data generated by the program) have sufficiently precise types that no cast is needed. Such precision gives a lot of information about the queries at hand and, as in the case of the pruning presented in the previous section, it should and could be possible to use this information for the optimisation of secondary memory queries. DTDs and XML Schemas have already been used to optimise access to XML data in secondary storage (in particular they were used to map XML into relations, e.g., [8, 53]), but for

[10] To tell the truth, some fancier aspects of XML Schema, such as integrity of IDREFs are not captured. But this goes beyond the possibility of any static type system.

semantic types/patterns this is a research direction that, as far as we know, has not yet been explored.

4 Conclusion

With this presentation of regular patterns and types for XML we hope to have convinced the reader that they constitute primitives worthy of consideration. To that end we described various problems that can be solved with the help of regular patterns and types. Apart from the interest of each problem, what really seems important is that regular patterns and types constitute a unique and general framework to understand and solve such a disparate disparate problems. Despite that, we do not believe that regular patterns are *the* solution: as we said repeatedly, we think one should aim at a tight integration, or even a unification, of path and pattern extraction primitives. We have discussed a query language, ℂQL, and hinted at an object-oriented language, Xtatic, in which both primitives coexist (even if one is defined in terms of the other), but the solutions proposed for these languages are far from being satisfactory since the two primitives are essentially disconnected. We aim at a deeper integration of the approaches in which, say, we could build paths over patterns (so as to capture intermediate results) or define regular expression patterns on paths (so as to perform pattern extractions at different depths) . . . or maybe both. In this perspective the work on the composition of monadic queries [26] that we cited in the related work section of the Introduction, looks quite promising. The idea is simple and elegant and consists in concatenating monadic queries so that the set of the nodes resulting from one query are the input nodes of the query that follows it. Actually, the concatenation (noted by a dot and denoting composition) is not performed among single queries but among unions and intersections of queries. This yields the following composition formalism $\phi ::= q \mid q.\phi \mid \phi \wedge \phi \mid \phi \vee \phi$ (where q ranges over particular monadic queries, called *parametrised queries*) that closely resembles to a mix of paths and pattern primitives.

All these considerations concern the extraction primitives, but the need of mixing horizontal and vertical navigation concerns iterators, as well. For instance, Hosoya's filters are inherently characterised by an horizontal behaviour: it is easier to use a path to apply a filter at a given depth, than to program the latter recursively so that it can autonomously perform the vertical navigation. Such a problem may be less felt in Nguyễn's filters, but we are afraid that using them to program a mix of vertical and horizontal iterations would be out of reach of the average programmer.

As a matter of fact, this last consideration probably holds for patterns already: it is true that it is easier to write a path than a pattern, and even if the use of the more sophisticated features of XPath is not for the fainthearted programmer, nevertheless simple queries are simpler in XPath. So if we want the use of regular expression patterns to spread outside the community of functional programmers, it will be necessary to find alternative ways to program them, for instance by developing QBE-style query interfaces. This will be even more urgent for a formalism integrating both pattern and path navigational primitives. It looks as a promising topic for future work.

Acknowledgements. I want to warmly thank Gavin Bierman for several suggestions and for proof-reading the paper. Véronique Benzaken, Peter Buneman, Dario Colazzo, Cédric Miachon, Ioana Manolescu, and Matthieu Objois provided useful feedback on this work. Part of it was prepared while I was visiting Microsoft Research in Cambridge and it greatly benefited of the interesting discussions and stimulating environment that I enjoyed during my stay. Thus let me warmly thank Microsoft Research for its support and the MSR Cambridge Lab for its “unmatchable” hospitality. This work was partially supported by the RNTL project "GraphDuce" and by the ACI project "Transformation Languages for XML: Logics and Applications".

References

1. S. Abiteboul, D. Quass, J. McHugh, J. Widom, and J. Wiener. The Lorel query language for semistructured data. *International Journal on Digital Libraries*, 1(1):68–88, 1997.
2. Roberto M. Amadio and Luca Cardelli. Subtyping recursive types. *ACM Transactions on Programming Languages and Systems*, 15(4), September 1993.
3. Bell-labs. *Galax*. `http://db.bell-labs.com/galax/`.
4. V. Benzaken, G. Castagna, and A. Frisch. CDuce: an XML-friendly general purpose language. In *ICFP '03, 8th ACM International Conference on Functional Programming*, pages 51–63, Uppsala, Sweden, 2003. ACM Press.
5. V. Benzaken, G. Castagna, and C. Miachon. A full pattern-based paradigm for XML query processing. In *PADL 05, 7th International Symposium on Practical Aspects of Declarative Languages*, number 3350 in LNCS, pages 235–252. Springer, January 2005.
6. Gavin Bierman, Erik Meijer, and Wolfram Schulte. The essence of data access in Cω. In *Proc. of ECOOP '2005, European Conference on Object-Oriented Programming*, volume 3586 of *Lecture Notes in Computer Science*. Springer, 2005.
7. 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.
8. P. Bohannon, J. Freire, J. Haritsa, M. Ramanath, P. Roy, and J. Simeon. LegoDB: customizing relational storage for XML documents. In *VLDB '02, 28th Int. Conference on Very Large Databases*, pages 1091–1094, 2002.
9. P. Bothner. Qexo - the GNU Kawa implementation of XQuery. `http://www.gnu.org/software/qexo/`.
10. Stéphane Bressan, Zoé Lacroix, Ying Guang Li, and Anna Maddalena. Prune the XML before you search it: XML transformations for query optimization. *DataX Workshop*, 2004.
11. Niklas Broberg, Andreas Farre, and Josef Svenningsson. Regular expression patterns. In *ICFP '04: 9th ACM SIGPLAN International Conference on Functional programming*, pages 67–78, New York, NY, USA, 2004. ACM Press.
12. A. Brüggemann-Klein and D. Wood. Caterpillars, context, tree automata and tree pattern matching. In *Proceedings of DLT '99: Foundations, Applications and Perspectives*. World Scientific Publishing Co., 2000.
13. C. Calcagno, P. Gardner, and U. Zarfaty. Context logic and tree update. In *POPL '05, 32nd ACM Symposium on Principles of Programming Languages*. ACM Press, 2005.
14. L. Cardelli, P. Gardner, and G. Ghelli. A spatial logic for querying graphs. In *29th International Colloquium on Automata, Languages, and Programming*, volume 2380 of *Lecture Notes in Computer Science*. Springer-Verlag, 2002.
15. L. Cardelli, P. Gardner, and G. Ghelli. Manipulating trees with hidden labels. In *6th International Conference on Foundations of Software Science and Computational Structures*, volume 2620 of *Lecture Notes in Computer Science*. Springer-Verlag, 2003.

16. L. Cardelli and G. Ghelli. Tql: A query language for semistructured data based on the ambient logic. *Mathematical Structures in Computer Science*, 14:285–327, 2004.
17. Luca Cardelli, Philippa Gardner, and Giorgio Ghelli. A spatial logic for querying graphs. In *ICALP '02, 29th International Colloquium on Automata, Languages and Programming*, pages 597–610. Springer-Verlag, 2002.
18. G. Castagna, D. Colazzo, and A. Frisch. Error mining for regular expression patterns. In *ICTCS 2005, Italian Conference on Theoretical Computer Science*, number 3701 in Lecture Notes in Computer Science. Springer, 2005.
19. G. Castagna and A. Frisch. A gentle introduction to semantic subtyping. In Proceedings of *PPDP '05, the 7th ACM SIGPLAN International Symposium on Principles and Practice of Declarative Programming,* ACM Press (full version) and *ICALP '05, 32nd International Colloquium on Automata, Languages and Programming,* Lecture Notes in Computer Science n. 3580, Springer (summary), Lisboa, Portugal, 2005. Joint ICALP-PPDP keynote talk.
20. D. Chamberlin, P. Fankhauser, D. Florescu, M. Marchiori, and J. Robie. XML Query Use Cases. Technical Report 20030822, World Wide Web Consortium, 2003.
21. J. Clark and S. DeRose. *XML Path Language (XPath).* W3C Recommendation, `http://www.w3.org/TR/xpath/`, November 1999.
22. James Clark. *XSL Transformations (XSLT).* W3C Recommendation, `http://www.w3.org/TR/xslt/`, November 1999.
23. Sophie Cluet. Designing OQL: allowing objects to be queried. *Inf. Syst.*, 23(5):279–305, 1998.
24. G. Conforti, G. Ghelli, A. Albano, D. Colazzo, P. Manghi, and C. Sartiani. The query language TQL. In *Proc. of the 5th WebDB, Madison, Wisconsin, USA*, pages 19–24, 2002.
25. A. Deutsch, M. F. Fernandez, D. Florescu, A. Y. Levy, and D. Suciu. "XML-QL: A Query Language for XML". In *WWW The Query Language Workshop (QL)*, 1998.
26. E. Filiot. Composition de requêtes monadiques dans les arbres. Master's thesis, Master Recherche de l'Université des Sciences et Technologies de Lille, 2005.
27. X. Franc. Qizx/open. `http://www.xfra.net/qizxopen`.
28. A. Frisch. Regular tree language recognition with static information. In *Proc. IFIP Conference on Theoretical Computer Science (TCS)*, Toulouse, 2004. Kluwer.
29. A. Frisch, G. Castagna, and V. Benzaken. Semantic Subtyping. In *LICS '02, 17th Annual IEEE Symposium on Logic in Computer Science*, pages 137–146. IEEE Computer Society Press, 2002.
30. Alain Frisch. *Théorie, conception et réalisation d'un langage de programmation fonctionnel adapté à XML.* PhD thesis, Université Paris 7, December 2004.
31. Vladimir Gapeyev and Benjamin C. Pierce. Regular object types. In *ECOOP '03, European Conference on Object-Oriented Programming*, 2003.
32. Vladimir Gapeyev and Benjamin C. Pierce. Paths into patterns. Technical Report MS-CIS-04-25, University of Pennsylvania, October 2004.
33. E. Goris and M. Marx. Looping caterpillars. In *LICS '05, 20th Annual IEEE Symposium on Logic in Computer Science*. IEEE Computer Society Press, 2005.
34. H. Hosoya. *Regular Expression Types for XML.* PhD thesis, The University of Tokyo, 2001.
35. H. Hosoya. Regular expressions pattern matching: a simpler design. Unpublished manuscript, February 2003.
36. H. Hosoya. Regular expression filters for XML. In *Programming Languages Technologies for XML (PLAN-X)*, 2004.
37. H. Hosoya, A. Frisch, and G. Castagna. Parametric polymorphism for XML. In *POPL '05, 32nd ACM Symposium on Principles of Programming Languages*. ACM Press, 2005.
38. H. Hosoya and M. Murata. Validation and boolean operations for attribute-element constraints. In *Programming Language Technologies for XML (PLAN-X)*, 2002.

39. H. Hosoya and B. Pierce. XDuce: A typed XML processing language. *ACM Transactions on Internet Technology*, 3(2):117–148, 2003.
40. H. Hosoya and B.C. Pierce. XDuce: A typed XML processing language. In *WebDB2000, 3rd International Workshop on the Web and Databases*, 2000.
41. H. Hosoya and B.C. Pierce. Regular expression pattern matching for XML. In *POPL '01, 25th ACM Symposium on Principles of Programming Languages*, 2001.
42. Internet Movie Database. `http://imdb.com`.
43. Michael Y. Levin and Benjamin C. Pierce. Type-based optimization for regular patterns. In *DBPL '05, Database Programming Languages*, August 2005.
44. L. Libkin. Logics for unranked trees: an overview. In *ICALP '05, 32nd International Colloquium on Automata, Languages and Programming*, number 3580 in LNCS. Springer, 2005.
45. K. Zhuo Ming Lu and M. Sulzmann. An implementation of subtyping among regular expression types. In *Proc. of APLAS'04*, volume 3302 of *LNCS*, pages 57–73. Springer, 2004.
46. Amélie Marian and Jérôme Siméon. Projecting XML elements. In *29th Int. Conference on Very Large Databases (VLDB '03)*, pages 213–224, 2003.
47. Erik Meijer and Mark Shields. XMλ: A functional language for constructing and manipulating XML documents. (Draft), 1999.
48. M. Murata, D. Lee, and M. Mani. Taxonomy of XML schema languages using formal language theory. In *Extreme Markup Languages*, 2001.
49. F. Neven and T. Schwentick. Expressive and efficient pattern languages for tree-structured data. In *PODS '00: 19th ACM Symposium on Principles of Database Dystems*, pages 145–156. ACM Press, 2000.
50. Kim Nguyễn. Une algèbre de filtrage pour le langage ℂDuce. DEA *Programmation*, Université Paris 11, September 2004. Available at `http://www.lri.fr/~kn/main.pdf`.
51. Yannis Papakonstantinou and Victor Vianu. Dtd inference for views of xml data. In *PODS '00: Proceedings of the nineteenth ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems*, pages 35–46, New York, NY, USA, 2000. ACM Press.
52. Albrecht Schmidt, Florian Waas, Martin L. Kersten, Michael J. Carey, Ioana Manolescu, and Ralph Busse. XMark: A benchmark for XML data management. In *Proceedings of the Int'l. Conference on Very Large Database Management (VLDB)*, pages 974–985, 2002.
53. J. Shanmugasundaram, K. Tufte, C. Zhang, G. He, D. DeWitt, and J. Naughton. Relational databases for querying XML documents: Limitations and opportunities. In *VLDB '99, 25th Int. Conference on Very Large Databases*, pages 302–314, 1999.
54. M. Odersky *et. al.* An overview of the Scala programming language. Technical Report IC/2004/64, École Polytechnique Fédérale de Lausanne, 2004. Latest version at `http://scala.epfl.ch`.
55. P. Trinder and P. Wadler. Improving list comprehension database queries. In *Proc. of TENCON '89, Bombay, India (November 1989), 186-192.*, 1989.
56. P. Wadler. List comprehensions. In *The Implementation of Functional Programming Languages* by S. Peyton Jones (chapter 7). Prentice Hall, 1987. Available on-line at `http://research.microsoft.com/Users/simonpj/Papers/slpj-book-1987`.