Towards a Core Calculus for XQuery 3.0

Combining navigational and pattern-matching approaches

Giuseppe Castagna[1] Hyeonseung Im[2] Kim Nguyễn[2] Véronique Benzaken[2]

[1]CNRS, PPS, Univ Paris Diderot, Sorbonne Paris Cité, Paris, France
[2]LRI, Université Paris-Sud, Orsay, France

Abstract

XML processing languages can be classified according to whether they extract XML data by paths or pattern matching. In the former category one finds XQuery, in the latter XDuce and ℂDuce. The strengths of one category correspond to the weaknesses of the other. In this work, we propose to bridge the gap between two of these languages: XQuery and ℂDuce. To this end, we extend ℂDuce so as it can be seen as a succinct core λ-calculus that captures XQuery 3.0 programs. The extensions we consider essentially allow ℂDuce to implement XPath-like navigation expressions by pattern matching and to precisely type them. The encoding of XQuery 3.0 into the extension of ℂDuce provides a formal semantics and a sound static type system for XQuery 3.0 programs.

1. Motivations

With the establishment of XML as a standard for data representation and exchange, a wealth of XML-oriented programming languages have emerged. They can be classified into two distinct classes according to whether they extract XML data by applying paths or patterns. The strengths of one class correspond to the weaknesses of the other. In this work, we propose to bridge the gap between these classes, and to do so we consider two languages each representing a distinct class: XQuery and ℂDuce.

2. Comparisons between XQuery and ℂDuce

XQuery [7] is a declarative language standardized by the W3C that heavily relies on XPath as a data extraction primitive. Interestingly, the next version of XQuery (version 3.0, currently being drafted [9]) adds several functional traits: type and value case analysis and functions as first-class citizens. However, while the W3C specifies a standard for document types (XML Schema), it says little about the typing of XQuery programs (the XQuery 3.0 draft goes as far as saying that static typing is "implementation defined" and therefore optional). This is a step back from the XQuery 1.0 Formal Semantics [8] which gives sound (but sometime imprecise) typing rules for XQuery.

In contrast, ℂDuce [2] is a programming language used in production but issued from academic research. It is a statically-typed functional language with, in particular, higher-order functions and powerful pattern matching tailored for XML data. A key characteristic of ℂDuce is its type algebra, which is based on *semantic subtyping* [5] and features recursive types, type constructors (product, record, and arrow types), and general Boolean connectives (union, intersection, and negation of types) as well as singleton types. This type algebra is particularly suited to express the types of XML documents and relies on the same foundation as the one that underpins XML Schema: regular tree languages. Finally, the ℂDuce type system supports *ad-hoc* polymorphism (through overloading and subtyping) while parametric polymorphism is not provided yet.

XQuery code:

```
declare function get_links($page, $print)
{
  for $i in $page/descendant::a[not(ancestor::b)]
  return $print($i)
}

declare function pretty($link)
{
  typeswitch($link)
  case $l as element(a)
          return switch ($l/@class)
                 case "style1"
                        return <a href={$l/@href}>
                                 <b>{$l/text()}</b>
                               </a>
                 default return $l
  default return $link
}
```

ℂDuce code:

```
let get_links : <_>_ → (<a>_ → <a>_) → [ <a>_ * ] =
fun page -> fun print ->
 match page with
  <a>_ & x -> [ (print x) ]
| <_\b> l -> (transform l with i) -> get_links i print)
| _ -> [ ]

let pretty : (<a>_ → <a>_) & (Any\<a>_ → Any\<a>_) =
fun link ->
 match link with
  <a class="style1" href=h> l -> <a href=h>[ <b>l ]
| x -> x
```

Figure 1. Document transformation, in XQuery 3.0 and ℂDuce

Figure 1 highlights the key features as well as the shortcomings of both languages by defining the same two functions "*get_links*" and "*pretty*" in each language. The first function "*get_links*" *(i)* takes an XHTML document "*$page*" and a function "*$print*" as input, *(ii)* computes the sequence of all hypertext links (a-labelled elements) of the document that do not occur below a bold element (b-labelled elements), and *(iii)* applies the *print* argument to each link in the sequence, returning the sequence of the results. The second function "*pretty*" takes anything as argument and performs a case analysis. If the argument is a link whose `class` attribute has the value `"style1"`, the output is a link with the same target (`href` attribute) and whose text is embedded in a bold element. Otherwise, the argument is unchanged.

We first look into the "*get_links*" function. In XQuery, collecting all the "a" elements of interest is straightforward: this is done through the XPath expression at Line 3:

\$page`/descendant::a[not(ancestor::b)]`

In a nutshell, an XPath expression is a sequence of steps that *(i)* select sets of nodes along the specified axis (here `descendant` meaning the descendants of the root node of *\$page*), *(ii)* keep only those nodes in the axis that have a particular label (here "a"), and *(iii)* further filter the results according to a Boolean condition (here `not(ancestor::b)` meaning that from a candidate "a" node, the step `ancestor::b` must return an empty result). At Lines 3–4, the "`for...return`" expression binds in turn each element of the result of the XPath expression to the variable *\$i*, evaluates the `return` expression, and concatenates the results. Note that there is no type annotation and that this function would fail at runtime if *\$page* is not an XML element or if *\$print* is not a function.

In clear contrast, in the ℂDuce program[1], the interface of "*get_links*" is fully specified (Line 19). Moreover, it is currified and takes two arguments. The first argument is "*page*" of type `<_>_`, which denotes any XML element (`_` denotes a wildcard pattern and is a synonym of the type $\mathbb{1}$, the type of all values, while `<`s`>`t is the type of an XML element with tag of type s and content of type t). The second argument is *print* of type `<a>_` $\to$ `<a>_`, which denotes functions that take an "a" element (whose content is anything) and return an "a" element. The final output is a value of type `[ <a>_ * ]`, which denotes a possibly empty sequence of "a" elements. The implementation of *get_links* in ℂDuce is quite different from its XQuery counterpart: following the functional idiom, it is defined as a recursive function that traverses its input recursively and performs a case analysis through pattern matching. If the input is an "a" element (Line 22), it binds the input to the capture variable "*x*", evaluates "*print x*", and puts the result in a sequence (denoted by square brackets). If the input is an XML element whose label is *not* "b" ("\" stands for difference, so `_\b` denotes or matches any value different from `b`), it captures the content of the element (a sequence) in "*l*" and applies itself recursively to each element of "*l*" using the `transform ... with` construct whose behavior is the same as XQuery's "`for`". Lastly, if the result is not an element (or it is a "b" element), it stops the recursion and returns the empty sequence.

For the *pretty* function, the XQuery version (Lines 7–18) first performs a "type switch", which tests whether the input "*\$link*" has the label "a". If so, it extracts the value of the `class` attribute using an XPath expression (Line 11) and performs a case analysis on that value. In the case where the attribute is `"style1"`, it re-creates an "a" element (with a nested "b" element) extracting the relevant part of the input using XPath expressions. The ℂDuce version (Lines 26–30) behaves in the same way but collapses all the cases in a single pattern matching. If the input is an "a" element with the desired `class` attribute, it binds the contents of the `href` attribute and the element to the variables *h* and *l*, respectively, and builds the desired output; otherwise, the input is returned unchanged. Interestingly, this function is *overloaded*. Its signature is composed of two arrow types: if the input is an "a" element, so is the output; if the input is something else than an "a" element, so is the output (`&` in types and patterns stands for intersection). Note that it is safe to use the *pretty* function as the second argument of the *get_links* function since `(<a>_`$\to$`<a>_) & (Any\<a>_`$\to$`Any\<a>_)` is a subtype of `<a>_`$\to$`<a>_` (an intersection is always smaller than or equal to the types that compose it).

[1] We took some liberties with ℂDuce's syntax in order to better match XQuery's one. The actual ℂDuce definitions are more compact.

Here we see that the strength of one language is the weakness of the other: ℂDuce provides static typing, a fine-grained type algebra, and a pattern matching construct that cleanly unifies type and value case analysis. XQuery provides through XPath a declarative way to navigate a document, which is more concise and less brittle than using hand-written recursive functions (in particular, at Line 22 in the ℂDuce code, there is an implicit assumption that a link cannot occur below another link; the recursion stops at "a" elements).

3. Contributions

We improve *both* XQuery and ℂDuce by showing that (an extended) ℂDuce can be seen as a succinct core λ-calculus that exactly captures XQuery 3.0 programs. To achieve this, we extend ℂDuce in several ways.

First, we allow one to navigate in ℂDuce values, both downward and upward. A natural way to do so in a functional setting is to use *zippers à la* Huet [6] to annotate values. Zippers denote the position in the surrounding tree of the value they annotate, as well as its current path from the root. We extend ℂDuce not only with zipper values (*i.e.*, values annotated by zippers) but also with *zipper types*. By doing so, we show that we can navigate not only in any direction in a document but also in a *precisely typed* way, allowing one to express constraints on the path in which a value is within a document.

Second, we extend ℂDuce pattern matching with accumulating variables that allow us to encode *recursive* XPath axes (such as `descendant` and `ancestor`). It is well known that typing such recursive axes goes well beyond regular tree languages and that approximations in the type system are needed. Rather than giving ad-hoc built-in functions for `descendant` and `ancestor`, we define the notion of *type operators* and parameterize the ℂDuce type system (and dynamic semantics) with these operators. Soundness properties can then be shown in a modular way without hard-coding any specific typing rules in the language. With this addition, XPath navigation can be encoded simply in ℂDuce's pattern matching constructs and it is just a matter of syntactic sugar definition to endow ℂDuce with nice declarative navigational expressions such as those successfully used in XQuery or XSLT.

Finally, on the XQuery side, we extend $\mathsf{XQ_H}$, a core version of XQuery 3.0 proposed by Benedikt and Vu [1], with type case, value case and type annotations on functions. We give an encoding of the extended $\mathsf{XQ_H}$ into ℂDuce. The encoding provides for free an effective and efficient typechecking algorithm for XQuery 3.0 programs as well as a formal and compact specification of their semantics. Even more interestingly, it provides a solid formal basis to start further studies on the definition of XQuery 3.0 and of its properties. At least, it is straightforward to use this basis to add overloaded functions to XQuery. More crucially, we expect that the recent advances on polymorphism for semantic subtyping [3, 4] can be transposed to this basis to provide a polymorphic type system and type inference algorithm both to XQuery 3.0 and to the extended ℂDuce language defined here. Polymorphic types are the missing ingredient to make higher-order functions yield their full potential and to remove any residual justification of the absence of standardization of XQuery 3.0 type system.

An expanded version of this paper, containing all the technical details and comparisons with related work, can be found at the following web page:

`https://www.lri.fr/~im`

References

[1] M. Benedikt and H. Vu. Higher-order functions and structured datatypes. In *WebDB*, pages 43–48, 2012.

[2] V. Benzaken, G. Castagna, and A. Frisch. CDuce: An XML-centric general-purpose language. In *ICFP*, pages 51–63, 2003.

[3] G. Castagna, K. Nguyễn, Z. Xu, H. Im, S. Lenglet, and L. Padovani. Polymorphic functions with set-theoretic types: part 1: syntax, semantics, and evaluation. In *POPL*, pages 5–18, 2014.

[4] G. Castagna and Z. Xu. Set-theoretic foundation of parametric polymorphism and subtyping. In *ICFP*, pages 94–106, 2011.

[5] A. Frisch, G. Castagna, and V. Benzaken. Semantic subtyping: Dealing set-theoretically with function, union, intersection, and negation types. *J. ACM*, 55(4):1–64, 2008.

[6] G. Huet. The Zipper. *J. Funct. Program.*, 7(5):549–554, 1997.

[7] W3C: XML Query. `http://www.w3.org/TR/xquery`, 2010.

[8] XQuery 1.0 and XPath 2.0 Formal Semantics (Second Edition). `http://www.w3.org/TR/xquery-semantics/`, 2010.

[9] W3C: XQuery 3.0. `http://www.w3.org/TR/xquery-3.0`, 2013.