A Core Calculus for XQuery 3.0
Combining Navigational and Pattern Matching Approaches

Giuseppe Castagna[1], Hyeonseung Im[2], Kim Nguyễn[3], and Véronique Benzaken[3]

[1] CNRS, PPS, Univ. Paris Diderot, Sorbonne Paris Cité, Paris, France
[2] Inria, LIG, Univ. Grenoble-Alpes, Grenoble, France
[3] LRI, Université Paris-Sud, Orsay, France

Abstract. XML processing languages can be classified according to whether they extract XML data by paths or patterns. The strengths of one category correspond to the weaknesses of the other. In this work, we propose to bridge the gap between these two classes by considering two languages, one in each class: XQuery (for path-based extraction) and ℂDuce (for pattern-based extraction). To this end, we extend ℂDuce so as it can be seen as a succinct core λ-calculus that captures XQuery 3.0. The extensions we consider essentially allow ℂDuce to implement XPath-like navigational expressions by pattern matching and precisely type them. The elaboration of XQuery 3.0 into the extended ℂDuce provides a formal semantics and a sound static type system for XQuery 3.0 programs.

1 Introduction

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.

XQuery [23] is a declarative language standardized by the W3C that relies heavily on XPath [21,22] as a data extraction primitive. Interestingly, the latest version of XQuery (version 3.0, very recently released [25]) 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 [26]), it says little about the typing of XQuery programs (the XQuery 3.0 recommendation goes as far as saying that static typing is "implementation defined" and hence optional). This is a step back from the XQuery 1.0 Formal Semantics [24] which gives sound (but sometime imprecise) typing rules for XQuery.

In contrast, ℂDuce [4], which is used in production but issued from academic research, is a statically-typed functional language with, in particular, higher-order functions and powerful pattern matching tailored for XML data. Its key characteristic is its type algebra, which is based on *semantic subtyping* [10] and features recursive types, type constructors (product, record, and arrow types)

J. Vitek (Ed.): ESOP 2015, LNCS 9032, pp. 232–256, 2015.
DOI: 10.1007/978-3-662-46669-8_10

XQuery code

```
1  declare function get_links($page, $print) {
2     for $i in $page/descendant::a[not(ancestor::b)]
3     return $print($i)
4  }
5  declare function pretty($link) {
6     typeswitch($link)
7     case $l as element(a)
8        return switch ($l/@class)
9           case "style1"
10             return <a href={$l/@href}><b>{$l/text()}</b></a>
11          default return $l
12    default return $link
13 }
```

ℂDuce code

```
14 let get_links (page: <_>_) (print: <a>_ -> <a>_) : [ <a>_ * ] =
15    match page with
16      <a>_ & x -> [ (print x) ]
17    | < (_\‘b) > l -> (transform l with (i & <_>_) -> get_links i print)
18    | _ -> [ ]
19 let pretty (<a>_ -> <a>_ ; Any\<a>_ -> Any\<a>_)
20    | <a class="style1" href=h ..> l -> <a href=h>[ <b>l ]
21    | x -> x
```

Fig. 1. Document transformation in XQuery 3.0 and ℂDuce

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. Moreover, the ℂDuce type system not only supports *ad-hoc* polymorphism (through overloading and subtyping) but also has recently been extended with parametric polymorphism [5,6].

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. Firstly, *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. Secondly, *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 at the *get_links* function. In XQuery, collecting every “`a`” element of interest is straightforward: it is done by the XPath expression at Line 2:

$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 2–3, 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, the interface of *get_links* is fully specified (Line 14). It is curried and takes two arguments. The first one is *page* of type `<_>_`, which denotes any XML element (`_` denotes a wildcard pattern and is a synonym of the type `Any`, 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 is the type of 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 (in ℂDuce's types, the content of a sequence is described by a regular expression on types). 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 16), 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 tag is *not* `b` ("\" stands for difference, so `_\‘b` matches any value different from `b`)[1], 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 (which is inspired from the example given in §3.16.2 of the XQuery 3.0 recommendation [25]), the XQuery version (Lines 5–13) first performs a "type switch", which tests whether the input *\$link* has label `a`. If so, it extracts the value of the `class` attribute using an XPath expression (Line 8) 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 19–21) 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 (the "`..`" matches possible further attributes), 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

[1] In ℂDuce, one has to use `‘b` in conjunction with \ to denote XML tag `b`.

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>_→<a>_) & (Any\<a>_→Any\<a>_)` is a subtype of `<a>_→<a>_` (an intersection is always smaller than or equal to the types that compose it).

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 16 in the ℂDuce code, there is an implicit assumption that a link cannot occur below another link; the recursion stops at "`a`" elements).

Contributions. The main contribution of the paper is to unify the navigational and pattern matching approaches and to define a formal semantics and type system of XQuery 3.0. Specifically, we extend ℂDuce so as it can be seen as a succinct core λ-calculus that can express XQuery 3.0 programs as follows.

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 [18] 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 zipped values (*i.e.*, values annotated by zippers) but also with *zipped 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.

The last (but not least) step of our work is to define a "normal form" for XQuery 3.0 programs, extending both the original XQuery Core normal form of [24] and its recent adaptation to XQuery 3.0 (dubbed $\mathsf{XQ_H}$) proposed by Benedikt and Vu [3]. In this normal form, navigational (*i.e.*, structural) expressions are well separated from data value expressions (ordering, node identity testing, *etc.*). We then provide a translation from XQuery 3.0 Core to ℂDuce extended with navigational patterns. The encoding provides for free an effective and efficient typechecking algorithm for XQuery 3.0 programs (described in Figure 9 of Section 5.1) 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

$$\begin{array}{lrcl}
\textbf{Pre-values} & w & ::= & c \mid (w,w) \mid \mu f^{(t\to t;\ldots;t\to t)}(x).e \\
\textbf{Zippers} & \delta & ::= & \bullet \mid \mathsf{L}\,(w)_\delta \cdot \delta \mid \mathsf{R}\,(w)_\delta \cdot \delta \\
\textbf{Values} & v & ::= & w \mid (v,v) \mid (w)_\delta \\
\textbf{Expressions} & e & ::= & v \mid x \mid \dot{x} \mid (e,e) \mid (e)_\bullet \mid o(e,\ldots,e) \\
 & & \mid & \mathsf{match}\ e\ \mathsf{with}\ p \to e \,|\, p \to e \\
\textbf{Pre-types} & u & ::= & b \mid c \mid u \times u \mid u \to u \mid u \vee u \mid \neg u \mid \mathbb{0} \\
\textbf{Zipper types} & \tau & ::= & \bullet \mid \top \mid \mathsf{L}\,(u)_\tau \cdot \tau \mid \mathsf{R}\,(u)_\tau \cdot \tau \mid \tau \vee \tau \mid \neg\tau \\
\textbf{Types} & t & ::= & u \mid t \times t \mid t \to t \mid t \vee t \mid \neg t \mid (u)_\tau \\
\textbf{Pre-patterns} & q & ::= & t \mid x \mid \dot{x} \mid (q,q) \mid q|q \mid q \,\&\, q \mid (x := c) \\
\textbf{Zipper patterns} & \varphi & ::= & \tau \mid \mathsf{L}\,p \cdot \varphi \mid \mathsf{R}\,p \cdot \varphi \mid \varphi|\varphi \\
\textbf{Patterns} & p & ::= & q \mid (p,p) \mid p|p \mid p \,\&\, p \mid (q)_\varphi
\end{array}$$

Fig. 2. Syntax of expressions, types, and patterns

definition of XQuery 3.0 and its properties. *A minima*, it is straightforward to use this basis to add overloaded functions to XQuery (*e.g.*, to give a precise type to *pretty*). More crucially, the recent advances on polymorphism for semantic subtyping [5,6,7] 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 the XQuery 3.0 type system.

Plan. Section 2 presents the core typed λ-calculus equipped with zipper-annotated values, accumulators, constructors, recursive functions, and pattern matching. Section 3 gives its semantics, type system, and the expected soundness property. Section 4 turns this core calculus into a full-fledged language using several syntactic constructs and encodings. Section 5 uses this language as a compilation target for XQuery. Lastly, Section 6 compares our work to other related approaches and concludes. Proofs and some technical definitions are given in an online appendix available at `http://www.pps.univ-paris-diderot.fr/~gc/`.

2 Syntax

We extend the ℂDuce language [4] with zippers *à la* Huet [18]. To ensure the well-foundedness of the definition, we stratify it, introducing first pre-values (which are standard ℂDuce values) and then values, which are pre-values possibly indexed by a zipper; we proceed similarly for types and patterns. The definition is summarized in Figure 2. Henceforth we denote by $\mathcal{V}$ the set of all values and by Ω a special value that represents runtime error and does not inhabit any type. We also denote by $\mathcal{E}$ and $\mathcal{T}$ the set of all expressions and all types, respectively.

2.1 Values and Expressions

Pre-values (ranged over by w) are the usual ℂDuce values without zipper annotations. Constants are ranged over by c and represent integers (`1`, `2`, ...), characters (`'a'`, `'b'`, ...), atoms (`` `nil ``, `` `true ``, `` `false ``, `` `foo ``, ...), *etc.* A value (w, w) represents pairs of pre-values. Our calculus also features recursive functions (hence the μ binder instead of the traditional λ) with explicit, overloaded types (the set of types that index the recursion variable, forming the *interface* of the function). Values (ranged over by v) are pre-values, pairs of values, or pre-values annotated with a *zipper* (ranged over by δ). Zippers are used to record the path covered when traversing a data structure. Since the product is the only construct, we need only three kinds of zippers: the empty one (denoted by $\bullet$) which intuitively denotes the starting point of our navigation, and two zippers $\mathsf{L}\,(w)_\delta \cdot \delta$ and $\mathsf{R}\,(w)_\delta \cdot \delta$ which denote respectively the path to the left and right projection of a pre-value w, which is itself reachable through δ. To ease the writing of several zipper related functions, we chose to record in the zipper the whole "stack" of values we have visited (each tagged with a left or right indication), instead of just keeping the unused component as is usual.

Example 1. Let v be the value $((1,(2,3)))_\bullet$. Its first projection is the value $(1)_{\mathsf{L}\,((1,(2,3)))_\bullet \cdot \bullet}$ and its second projection is the value $((2,3))_{\mathsf{R}\,((1,(2,3)))_\bullet \cdot \bullet}$, the first projection of which being $(2)_{\mathsf{L}\,((2,3))_{\mathsf{R}\,((1,(2,3)))_\bullet \cdot \bullet} \cdot \mathsf{R}\,((1,(2,3)))_\bullet \cdot \bullet}$

As one can see in this example, keeping values in the zipper (instead of pre-values) seems redundant since the same value occurs several times (see how δ is duplicated in the definition of zippers). The reason for this duplication is purely syntactic: it makes the writing of types and patterns that match such values much shorter (intuitively, to go "up" in a zipper, it is only necessary to extract the previous value while keeping it un-annotated —*i.e.*, having $\mathsf{L}\,w \cdot \delta$ in the definition instead of $\mathsf{L}\,(w)_\delta \cdot \delta$— would require a more complex treatment to reconstruct the parent). We also stress that zipped values are meant to be used only for internal representation: the programmer will be allowed to write just pre-values (not values or expressions with zippers) and be able to obtain and manipulate zippers only by applying ℂDuce functions and pattern matching (as defined in the rest of the paper) and never directly.

Expressions include values (as previously defined), variables (ranged over by x, y, ...), accumulators (which are a particular kind of variables, ranged over by $\dot{x}$, $\dot{y}$, ...), and pairs. An expression $(e)_\bullet$ annotates e with the empty zipper $\bullet$. The pattern matching expression is standard (with a first match policy) and will be thoroughly presented in Section 3. Our calculus is parameterized by a set $\mathcal{O}$ of built-in operators ranged over by o. Before describing the use of operators and the set of operators defined in our calculus (in particular the operators for projection and function application), we introduce our type algebra.

2.2 Types

We first recall the ℂDuce type algebra, as defined in [10], where types are interpreted as sets of values and the subtyping relation is semantically defined by using this interpretation (*i.e.*, $[\![t]\!] = \{v \mid \vdash v : t\}$ and $s \leq t \overset{\text{def}}{\iff} [\![s]\!] \subseteq [\![t]\!]$).

Pre-types u (as defined in Figure 2) are the usual ℂDuce types, which are possibly infinite terms with two additional requirements:

1. (regularity) the number of distinct subterms of u is finite;
2. (contractiveness) every infinite branch of u contains an infinite number of occurrences of either product types or function types.

We use b to range over basic types (int, bool, ...). A singleton type c denotes the type that contains only the constant value c. The empty type $\mathbb{0}$ contains no value. Product and function types are standard: $u_1 \times u_2$ contains all the pairs (w_1, w_2) for $w_i \in u_i$, while $u_1 \to u_2$ contains all the (pre-)value functions that when applied to a value in u_1, if such application terminates then it returns a value in u_2. We also include type connectives for union and negation (intersections are encoded below) with their usual set-theoretic interpretation. Infiniteness of pre-types accounts for recursive types and regularity implies that pre-types are finitely representable, for instance, by recursive equations or by the explicit μ-notation. Contractiveness [2] excludes both ill-formed (*i.e.*, unguarded) recursions such as $\mu X.X$ as well as meaningless type definitions such as $\mu X.X \vee X$ or $\mu X.\neg X$ (unions and negations are finite). Finally, subtyping is defined as set-theoretic containment (u_1 is a subtype of u_2, denoted by $u_1 \leq u_2$, if all values in u_1 are also in u_2) and it is decidable in EXPTIME (see [10]).

A *zipper type* τ is a possibly infinite term that is regular as for pre-types and contractive in the sense that every infinite branch of τ must contain an infinite number of occurrences of either left or right projection. The singleton type $\bullet$ is the type of the empty zipper and $\top$ denotes the type of all zippers, while $\mathsf{L}\,(u)_\tau \cdot \tau$ (resp., $\mathsf{R}\,(u)_\tau \cdot \tau$) denotes the type of zippers that encode the left (resp., right) projection of some value of pre-type u. We use $\tau_1 \wedge \tau_2$ to denote $\neg(\neg\tau_1 \vee \neg\tau_2)$.

The type algebra of our core calculus is then defined as pre-types possibly indexed by zipper types. As for pre-types, a *type* t is a possibly infinite term that is both regular and contractive. We write $t \wedge s$ for $\neg(\neg t \vee \neg s)$, $t \setminus s$ for $t \wedge \neg s$, and $\mathbb{1}$ for $\neg\mathbb{0}$; in particular, $\mathbb{1}$ denotes the super-type of all types (it contains all values). We also define the following notations (we use $\equiv$ both for syntactic equivalence and definition of syntactic sugar):

- $\mathbb{1}_{\mathsf{prod}} \equiv \mathbb{1} \times \mathbb{1}$ the super-type of all product types
- $\mathbb{1}_{\mathsf{fun}} \equiv \mathbb{0} \to \mathbb{1}$ the super-type of all arrow types
- $\mathbb{1}_{\mathsf{basic}} \equiv \mathbb{1} \setminus (\mathbb{1}_{\mathsf{prod}} \vee \mathbb{1}_{\mathsf{fun}} \vee (\mathbb{1})_\top)$ the super-type of all basic types
- $\mathbb{1}_{\mathsf{NZ}} \equiv \mu X.(X \times X) \vee (\mathbb{1}_{\mathsf{basic}} \vee \mathbb{1}_{\mathsf{fun}})$ the type of all pre-values (*i.e.*, Not Zipped)

It is straightforward to extend the subtyping relation of pre-types (*i.e.*, the one defined in [10]) to our types: the addition of $(u)_\tau$ corresponds to the addition of a new type constructor (akin to $\to$ and $\times$) to the type algebra. Therefore, it suffices to define the interpretation of the new constructor to complete the definition of the subtyping relation (defined as containment of the interpretations). In particular, $(u)_\tau$ is interpreted as the set of all values $(w)_\delta$ such that $\vdash w : u$ and $\vdash \delta : \tau$ (both typing judgments are defined in Appendix B.1). From this we deduce that $(\mathbb{1})_\top$ (equivalently, $(\mathbb{1}_{\mathsf{NZ}})_\top$) is the type of all (pre-)values decorated with a zipper. The formal definition is more involved (see Appendix A) but the intuition is simple: a type $(u_1)_{\tau_1}$ is a subtype of $(u_2)_{\tau_2}$ if $u_1 \leq u_2$ and τ_2 is a prefix (modulo type equivalence and subtyping) of τ_1. The prefix containment translates the intuition that the more we know about the context surrounding a value, the more numerous are the situations in which it can be safely used. For instance, in XML terms, if we have a function that expects an element whose parent's first child is an integer, then we can safely apply this function to an element whose type indicates that its parent's first child has type (a subtype of) integer *and* that its grandparent is, say, tagged by `a`.

Finally, as for pre-types, the subtyping relation for types is decidable in EXPTIME. This is easily shown by producing a straightforward linear encoding of zipper types and zipper values in pre-types and pre-values, respectively (the encoding is given in Definition 16 in Appendix A).

2.3 Operators and Accumulators

As previously explained, our calculus includes accumulators and is parameterized by a set $\mathcal{O}$ of operators. These have the following formal definitions:

Definition 2 (Operator). *An operator is a 4-tuple* $(o, n_o, \overset{o}{\rightsquigarrow}, \overset{o}{\rightarrow})$ *where* o *is the name (symbol) of the operator,* n_o *is its arity,* $\overset{o}{\rightsquigarrow} \subseteq \mathcal{V}^{n_o} \times \mathcal{E} \cup \{\Omega\}$ *is its reduction relation, and* $\overset{o}{\rightarrow} : \mathcal{T}^{n_o} \to \mathcal{T}$ *is its typing function.*

In other words, an operator is an applicative symbol, equipped with both a dynamic ($\rightsquigarrow$) and a static ($\rightarrow$) semantics. The reason for making $\overset{o}{\rightsquigarrow}$ a relation is to account for non-deterministic operators (*e.g.*, random choice). Note that an operator may fail, thus returning the special value Ω during evaluation.

Definition 3 (Accumulator). *An* accumulator $\dot{x}$ *is a variable equipped with a binary operator* $\mathit{Op}(\dot{x}) \in \mathcal{O}$ *and initial value* $\mathit{Init}(\dot{x}) \in \mathcal{V}$.

2.4 Patterns

Now that we have defined types and operators, we can define patterns. Intuitively, patterns are types with capture variables that are used either to extract subtrees from an input value or to test its "shape". As before, we first recall the definition of standard $\mathbb{C}$Duce patterns (here called pre-patterns), enrich them with accumulators, and then extend the whole with zippers.

A pre-pattern q, as defined in Figure 2, is either a type constraint t, or a capture variable x, or an accumulator $\dot{x}$, or a pair (q_1, q_2), or an alternative $q_1 | q_2$, or a conjunction $q_1 \,\&\, q_2$, or a default case $(x := c)$. It is a possibly infinite term that is regular as for pre-types and contractive in the sense that every infinite branch of q must contain an infinite number of occurrences of pair patterns. Moreover, the subpatterns forming conjunctions must have distinct capture variables and those forming alternatives the same capture variables. A *zipper pattern* φ is a possibly infinite term that is both regular and contractive as for zipper types. Finally, a pattern p is a possibly infinite term with the same requirements as pre-patterns. Besides, the subpatterns q and φ forming a zipper pattern $(q)_\varphi$ must have distinct capture variables. We denote by $\mathrm{Var}(p)$ the set of capture variables occurring in p and by $\mathrm{Acc}(p)$ the set of accumulators occurring in p.

$$E ::= [\,] \mid (E, e) \mid (e, E) \mid (E)_\bullet \mid \mathsf{match}\ E\ \mathsf{with}\ p_1 \to e_1 \,|\, p_2 \to e_2 \mid o(e, ..., E, ..., e)$$

$$\frac{(v_1, \dots, v_{n_o}) \overset{o}{\rightsquigarrow} e}{o(v_1, \dots, v_{n_o}) \rightsquigarrow e} \qquad \frac{\{\dot{x} \mapsto \mathsf{Init}(\dot{x}) \mid \dot{x} \in \mathrm{Acc}(p_1)\}; \square \vdash v/p_1 \rightsquigarrow \sigma, \gamma}{\mathsf{match}\ v\ \mathsf{with}\ p_1 \to e_1 \,|\, p_2 \to e_2 \rightsquigarrow e_1[\sigma;\ \gamma]}$$

$$\frac{\{\dot{x} \mapsto \mathsf{Init}(\dot{x}) \mid \dot{x} \in \mathrm{Acc}(p_1)\}; \square \vdash v/p_1 \rightsquigarrow \Omega \quad \{\dot{x} \mapsto \mathsf{Init}(\dot{x}) \mid \dot{x} \in \mathrm{Acc}(p_2)\}; \square \vdash v/p_2 \rightsquigarrow \sigma, \gamma}{\mathsf{match}\ v\ \mathsf{with}\ p_1 \to e_1 \,|\, p_2 \to e_2 \rightsquigarrow e_2[\sigma;\ \gamma]}$$

$$\frac{e \rightsquigarrow e'}{E[e] \rightsquigarrow E[e']} \qquad \frac{}{e \rightsquigarrow \Omega} \ \left(\begin{array}{l}\text{if no other rule applies}\\ \text{and } e \text{ is not a value}\end{array}\right)$$

Fig. 3. Operational semantics (reduction contexts and rules)

3 Semantics

In this section, the most technical one, we present the operational semantics and the type system of our calculus, and state the expected soundness properties.

3.1 Operational Semantics

We define a call-by-value, small-step operational semantics for our core calculus, using the reduction contexts and reduction rules given in Figure 3, where Ω is a special value representing a runtime error.

Of course, most of the actual semantics is hidden (the careful reader will have noticed that applications and projections are not explicitly included in the syntax of our expressions). Most of the work happens either in the semantics of operators or in the matching v/p of a value v against a pattern p. Such a matching, if it succeeds (*i.e.*, if it does not return Ω), returns two substitutions, one (ranged over by γ) from the capture variables of p to values and the other (ranged over by δ) from the accumulators to values. These two substitutions are simultaneously applied (noted $e_i[\sigma;\gamma]$) to the expression e_i of the pattern p_i that succeeds, according to a first match policy (v/p_2 is evaluated only if v/p_1 fails). Before explaining how to derive the pattern matching judgments "$\_ \vdash v/p \rightsquigarrow \_$" (in particular, the meaning of the context on the LHS of the turnstile "$\vdash$"), we introduce a minimal set of operators: application, projections, zipper erasure, and sequence building (we use $\mathsf{sans\text{-}serif}$ font for concrete operators). We only give their reduction relation and defer their typing relation to Section 3.2.

Function application: the operator $\mathsf{app}(\_,\_)$ implements the usual β-reduction:

$$v, v' \overset{\mathsf{app}}{\rightsquigarrow} e[{}^{v}\!/\!_{f}; {}^{v'}\!/\!_{x}] \qquad \text{if } v = \mu f^{(\dots)}(x).e$$

and $v, v' \overset{\mathsf{app}}{\rightsquigarrow} \Omega$ if v is not a function. As customary, $e[{}^{v}\!/\!_{x}]$ denotes the capture-avoiding substitution of v for x in e, and we write $e_1\ e_2$ for $\mathsf{app}(e_1, e_2)$.

Projection: the operator $\pi_1(\_)$ (resp., $\pi_2(\_)$) implements the usual first (resp., second) projection for pairs:

$$(v_1, v_2) \overset{\pi_i}{\rightsquigarrow} v_i \qquad \text{for } i \in \{1, 2\}$$

The application of the above operators returns Ω if the input is not a pair.

Zipper erasure: given a zipper-annotated value, it is sometimes necessary to remove the zipper (*e.g.*, to embed this value into a new data structure). This is achieved by the following remove $\mathsf{rm}(\_)$ and deep remove $\mathsf{drm}(\_)$ operators:

$$\begin{array}{lll} (w)_\delta & \overset{\mathsf{rm}}{\rightsquigarrow} & w \\ v & \overset{\mathsf{rm}}{\rightsquigarrow} & v \quad \text{if } v \not\equiv (w)_\delta \end{array} \qquad \begin{array}{lll} w & \overset{\mathsf{drm}}{\rightsquigarrow} & w \\ (w)_\delta & \overset{\mathsf{drm}}{\rightsquigarrow} & w \\ (v_1, v_2) & \overset{\mathsf{drm}}{\rightsquigarrow} & (\mathsf{drm}(v_1), \mathsf{drm}(v_2)) \end{array}$$

The former operator only erases the top-level zipper (if any), while the latter erases all zippers occurring in its input.

Sequence building: given a sequence (encoded *à la* Lisp) and an element, we define the operators $\mathsf{cons}(\_)$ and $\mathsf{snoc}(\_)$ that insert an input value at the beginning and at the end of the input sequence:

$$v, v' \overset{\mathsf{cons}}{\rightsquigarrow} (v, v') \qquad \begin{array}{lll} v, \texttt{`nil} & \overset{\mathsf{snoc}}{\rightsquigarrow} & (v, \texttt{`nil}) \\ v, (v', v'') & \overset{\mathsf{snoc}}{\rightsquigarrow} & (v', \mathsf{snoc}(v, v'')) \end{array}$$

The applications of these operators yield Ω on other inputs.

To complete our presentation of the operational semantics, it remains to describe the semantics of pattern matching. Intuitively, when matching a value v against a pattern p, subparts of p are recursively applied to corresponding subparts of v until a base case is reached (which is always the case since all values are finite). As usual, when a pattern variable is confronted with a subvalue, the binding is stored as a substitution. We supplement this usual behavior of pattern matching with two novel features. First, we add *accumulators*, that is, special variables in which results are accumulated during the recursive matching. The reason for keeping these two kinds of variables distinct is explained in Section 3.2 and is related to type inference for patterns. Second, we parameterize pattern matching by a zipper of the current value so that it can properly update the zipper when navigating the value (which should be of the pair form).

These novelties are reflected by the semantics of pattern matching, which is given by the judgment $\sigma; \delta^? \vdash v/p \rightsquigarrow \sigma', \gamma$, where v is a value, p a pattern, γ a mapping from $\mathrm{Var}(p)$ to values, and σ and σ' are mappings from accumulators to values. $\delta^?$ is an optional zipper value, which is either δ or a none value $\square$ (we consider $(v)_\square$ to be v). The judgment "returns" the result of matching the value v against the pattern p (noted v/p), that is, two substitutions: γ for capture variables and σ' for accumulators. Since the semantics is given compositionally, the matching may happen on a subpart of an "outer" matched value. Therefore, the judgment records on the LHS of the turnstile the context of the outer value explored so far: σ stores the values already accumulated during the matching, while $\delta^?$ tracks the possible zipper of the outer value (or it is $\square$ if the outer value has no zipper). The context is "initialized" in the two rules of the operational semantics of match in Figure 3, by setting each accumulator of the pattern to its initial value (function $\mathsf{Init}()$) and the outer zipper to $\square$.

Judgments for pattern matching are derived by the rules given in Figure 4. The rules pat-acc, pat-pair-zip, and zpat-* are novel, as they extend pattern matching with accumulators and zippers, while the others are derived from [4,9].

$$\frac{(\vdash v : t)}{\sigma;\delta^? \vdash v/t \rightsquigarrow \sigma, \varnothing}\ \textsf{pat-type} \qquad \frac{}{\sigma;\delta^? \vdash v/\dot{x} \rightsquigarrow \sigma[\,{}^{\mathsf{Op}(\dot{x})(v_{\delta^?},\sigma(\dot{x}))}/_{\dot{x}}\,], \varnothing}\ \textsf{pat-acc}$$

$$\frac{}{\sigma;\delta^? \vdash v/x \rightsquigarrow \sigma, \{x \mapsto v_{\delta^?}\}}\ \textsf{pat-var} \qquad \frac{}{\sigma;\delta^? \vdash v/(x := c) \rightsquigarrow \sigma, \{x \mapsto c\}}\ \textsf{pat-def}$$

$$\frac{\sigma;\square \vdash v_1/p_1 \rightsquigarrow \sigma', \gamma_1 \quad \sigma';\square \vdash v_2/p_2 \rightsquigarrow \sigma'', \gamma_2}{\sigma;\square \vdash (v_1, v_2)/(p_1, p_2) \rightsquigarrow \sigma'', \gamma_1 \oplus \gamma_2}\ \textsf{pat-pair}$$

$$\frac{\sigma;\mathsf{L}\,(w_1, w_2)_\delta \cdot \delta \vdash w_1/p_1 \rightsquigarrow \sigma', \gamma_1 \quad \sigma';\mathsf{R}\,(w_1, w_2)_\delta \cdot \delta \vdash w_2/p_2 \rightsquigarrow \sigma'', \gamma_2}{\sigma;\delta \vdash (w_1, w_2)/(p_1, p_2) \rightsquigarrow \sigma'', \gamma_1 \oplus \gamma_2}\ \textsf{pat-pair-zip}$$

$$\frac{\sigma;\delta^? \vdash v/p_1 \rightsquigarrow \sigma', \gamma}{\sigma;\delta^? \vdash v/p_1 \mid p_2 \rightsquigarrow \sigma', \gamma}\ \textsf{pat-or1} \qquad \frac{\sigma;\delta^? \vdash v/p_1 \rightsquigarrow \Omega \quad \sigma;\delta^? \vdash v/p_2 \rightsquigarrow \sigma', \gamma}{\sigma;\delta^? \vdash v/p_1 \mid p_2 \rightsquigarrow \sigma', \gamma}\ \textsf{pat-or2}$$

$$\frac{\sigma;\delta^? \vdash v/p_1 \rightsquigarrow \sigma', \gamma_1 \quad \sigma';\delta^? \vdash v/p_2 \rightsquigarrow \sigma'', \gamma_2}{\sigma;\delta^? \vdash v/p_1 \,\&\, p_2 \rightsquigarrow \sigma'', \gamma_1 \oplus \gamma_2}\ \textsf{pat-and}$$

$$\frac{\sigma;\delta \vdash w/q \rightsquigarrow \sigma', \gamma_1 \quad \sigma' \vdash \delta/\varphi \rightsquigarrow \sigma'', \gamma_2}{\sigma;\square \vdash (w)_\delta/(q)_\varphi \rightsquigarrow \sigma'', \gamma_1 \oplus \gamma_2}\ \textsf{pat-zip} \qquad \frac{(\vdash \delta : \tau)}{\sigma \vdash \delta/\tau \rightsquigarrow \sigma, \varnothing}\ \textsf{zpat-type}$$

$$\frac{\begin{array}{c}\sigma;\square \vdash (w)_\delta/p \rightsquigarrow \sigma', \gamma_1 \\ \sigma' \vdash \delta/\varphi \rightsquigarrow \sigma'', \gamma_2 \quad \gamma = \gamma_1 \oplus \gamma_2\end{array}}{\sigma \vdash \mathsf{L}\,(w)_\delta \cdot \delta/\mathsf{L}\,p \cdot \varphi \rightsquigarrow \sigma'', \gamma}\ \textsf{zpat-left} \qquad \frac{\begin{array}{c}\sigma;\square \vdash (w)_\delta/p \rightsquigarrow \sigma', \gamma_1 \\ \sigma' \vdash \delta/\varphi \rightsquigarrow \sigma'', \gamma_2 \quad \gamma = \gamma_1 \oplus \gamma_2\end{array}}{\sigma \vdash \mathsf{R}\,(w)_\delta \cdot \delta/\mathsf{R}\,p \cdot \varphi \rightsquigarrow \sigma'', \gamma}\ \textsf{zpat-right}$$

$$\frac{\sigma \vdash \delta/\varphi_1 \rightsquigarrow \sigma', \gamma}{\sigma \vdash \delta/\varphi_1 \mid \varphi_2 \rightsquigarrow \sigma', \gamma}\ \textsf{zpat-or1} \qquad \frac{\sigma \vdash \delta/\varphi_1 \rightsquigarrow \Omega \quad \sigma \vdash \delta/\varphi_2 \rightsquigarrow \sigma', \gamma}{\sigma \vdash \delta/\varphi_1 \mid \varphi_2 \rightsquigarrow \sigma', \gamma}\ \textsf{zpat-or2}$$

$$\frac{(\text{otherwise})}{\sigma;\delta^? \vdash v/p \rightsquigarrow \Omega}\ \textsf{pat-error} \qquad \frac{(\text{otherwise})}{\sigma \vdash \delta/\varphi \rightsquigarrow \Omega}\ \textsf{zpat-error}$$

$$\begin{array}{lll}\text{where } \gamma_1 \oplus \gamma_2 & \overset{\text{def}}{=} & \{x \mapsto \gamma_1(x) \mid x \in \mathsf{dom}(\gamma_1) \backslash \mathsf{dom}(\gamma_2)\} \\ & \cup & \{x \mapsto \gamma_2(x) \mid x \in \mathsf{dom}(\gamma_2) \backslash \mathsf{dom}(\gamma_1)\} \\ & \cup & \{x \mapsto (\gamma_1(x), \gamma_2(x)) \mid x \in \mathsf{dom}(\gamma_1) \cap \mathsf{dom}(\gamma_2)\}\end{array}$$

Fig. 4. Pattern matching

There are three base cases for matching: testing the input value against a type (rule pat-type), updating the environment σ for accumulators (rule pat-acc), or producing a substitution γ for capture variables (rules pat-var and pat-def). Matching a pattern (p_1, p_2) only succeeds if the input is a pair and the matching of each subpattern against the corresponding subvalue succeeds (rule pat-pair). Furthermore, if the value being matched was below a zipper (*i.e.*, the current zipper context is a δ and not—as in pat-pair— $\square$), we update the current zipper context (rule pat-pair-zip); notice that in this case the matched value must be a pair of pre-values since zipped values cannot be nested. An alternative pattern $p_1 \mid p_2$ first tries to match the pattern p_1 and if it fails, tries the pattern p_2 (rules pat-or1 and pat-or2). The matching of a conjunction pattern $p_1 \,\&\, p_2$ succeeds if and only if the matching of both patterns succeeds (rule pat-and). For a zipper constraint $(q)_\varphi$, the matching succeeds if and only if the input value is annotated by a zipper, *e.g.*, $(w)_\delta$, and both the matching of w with q and δ with φ succeed (rule pat-zip). It requires the zipper context to be $\square$ since we do not allow nested zipped values. When matching w with q, we record the zipper δ into the context so that it can be updated (in the rule pat-pair-zip) while navigating the value.

The matching of a zipper pattern φ against a zipper δ (judgments $\sigma \vdash \delta/\varphi \rightsquigarrow \sigma', \gamma$ derived by the zpat-* rules) is straightforward: it succeeds if both φ and δ are built using the same constructor (either L or R) and the componentwise matching succeeds (rules zpat-left and zpat-right). If the zipper pattern is a zipper type, the matching tests the input zipper against the zipper type (rule zpat-type), and alternative zipper patterns $\varphi_1 | \varphi_2$ follow the same first match policy as alternative patterns. If none of the rules is applicable, the matching fails (rules pat-error and zpat-error). Note that initially the environment σ contains $\mathsf{Init}(\dot{x})$ for each accumulator $\dot{x}$ in $\mathrm{Acc}(p)$ (rules for match in Figure 3).

Intuitively, γ is built when returning from the recursive descent in p, while σ is built using a *fold*-like computation. It is the typing of such fold-like computations that justifies the addition of accumulators (instead of relying on plain functions). But before presenting the type system of the language, we illustrate the behavior of pattern matching by some examples.

Example 4. Let $v \equiv (2, (\texttt{`true}, (3, \texttt{`nil})))$, $\mathsf{Init}(\dot{x}) = \texttt{`nil}$, $\mathsf{Op}(\dot{x}) = \mathsf{cons}$, and $\sigma \equiv \{\dot{x} \mapsto \texttt{`nil}\}$. Then, we have the following matchings:

1. $\sigma; \square \vdash v/(\mathsf{int}, (x, \_)) \rightsquigarrow \varnothing, \{x \mapsto \texttt{`true}\}$
2. $\sigma; \square \vdash v/\mu X.((x \,\&\, \mathsf{int} \,|\, \_, X) \,|\, (x := \texttt{`nil})) \rightsquigarrow \varnothing, \{x \mapsto (2, (3, \texttt{`nil}))\}$
3. $\sigma; \square \vdash v/\mu X.((\dot{x}, X) \,|\, \texttt{`nil}) \rightsquigarrow \{\dot{x} \mapsto (3, (\texttt{`true}, (2, \texttt{`nil})))\}, \varnothing$

In the first case, the input v (the sequence `[2 `true 3]` encoded *à la* Lisp) is matched against a pattern that checks if the first element has type int (rule pat-type), binds the second element to x (rule pat-var), and ignores the rest of the list (rule pat-type, since the anonymous variable "_" is just an alias for $\mathbb{1}$).

The second case is more involved since the pattern is recursively defined. Because of the first match policy of rule pat-or1, the product part of the pattern is matched recursively until the atom `` `nil`` is reached. When that is the case, the variable x is bound to a default value `` `nil``. When returning from this recursive matching, since x occurs both on the left and on the right of the product (in $x \,\&\, \mathsf{int}$ and in X itself), a pair of the binding found in each part is formed (third set in the definition of $\oplus$ in Figure 4), thus yielding a mapping $\{x \mapsto (3, \texttt{`nil})\}$. Returning again from the recursive call, only the "_" part of the pattern matches the input `` `true`` (since it is not of type int, the intersection test fails). Therefore, the binding for this step is only the binding for the right part (second case of the definition of $\oplus$). Lastly, when reaching the top-level pair, $x \,\&\, \mathsf{int}$ matches 2 and a pair is formed from this binding and the one found in the recursive call, yielding the final binding $\{x \mapsto (2, (3, \texttt{`nil}))\}$.

The third case is more intuitive. The pattern just recurses the input value, calling the accumulation function for $\dot{x}$ along the way for each value against which it is confronted. Since the operator associated with $\dot{x}$ is cons (which builds a pair of its two arguments) and the initial value is `` `nil``, this has the effect of computing the reversal of the list.

Note the key difference between the second and third case. In both cases, the structure of the pattern (and the input) dictates the traversal, but in the second case, it also dictates *how* the binding is built (if v was a tree and not a list, the binding for x would also be a tree in the second case). In the third case, the way the binding is built is defined by the semantics of the operator and independent of the input. This allows us to reverse sequences or flatten tree structures, both of which are operations that escape the expressiveness of regular tree languages/regular patterns, but which are both necessary to encode XPath.

3.2 Type System

The main difficulty in devising the type system is to type pattern matching and, more specifically, to infer the types of the accumulators occurring in patterns.

Definition 5 (Accepted input of an operator). *The* accepted input *of an operator* $(o, n, \overset{o}{\rightsquigarrow}, \overset{o}{\rightarrow})$ *is the set* $\mathbb{I}(o)$*, defined as:*

$$\mathbb{I}(o) = \{(v_1, ..., v_n) \in \mathcal{V}^n \mid (((v_1, ..., v_n) \overset{o}{\rightsquigarrow} e) \wedge (e \rightsquigarrow^* v)) \Rightarrow v \neq \Omega\}$$

Definition 6 (Exact input). *An operator o has an* exact input *if and only if* $\mathbb{I}(o)$ *is (the interpretation of) a type.*

We can now state a first soundness theorem, which characterizes the set of all values that make a given pattern succeed:

Theorem 7 (Accepted types). *Let p be a pattern such that for every $\dot{x}$ in* $\mathrm{Acc}(p)$*,* $\mathit{Op}(\dot{x})$ *has an exact input. Then, the set of all values v such that* $\{\dot{x} \mapsto \mathit{Init}(\dot{x}) \mid \dot{x} \in \mathrm{Acc}(p)\}; \square \vdash v/p \not\rightsquigarrow \Omega$ *is a type. We call this set the* accepted type *of p and denote it by* $\wr p \int$.

We next define the type system for our core calculus, in the form of a judgment $\Gamma \vdash e : t$ which states that in a typing environment Γ (*i.e.*, a mapping from variables and accumulators to types) an expression e has type t. This judgment is derived by the set of rules given in Figure 10 in Appendix. Here, we show only the most important rules, namely those for accumulators and zippers:

$$\frac{}{\Gamma \vdash \dot{x} : \Gamma(\dot{x})} \qquad \frac{\vdash w : t \quad \vdash \delta : \tau \quad t \leq \mathbb{1}_{\mathsf{NZ}}}{\Gamma \vdash (w)_\delta : (t)_\tau} \qquad \frac{\vdash e : t \quad t \leq \mathbb{1}_{\mathsf{NZ}}}{\Gamma \vdash (e)_\bullet : (t)_\bullet}$$

which rely on an auxiliary judgment $\vdash \delta : \tau$ stating that a zipper δ has zipper type τ. The rule for operators is:

$$\frac{\forall i = 1..n_o,\ \Gamma \vdash e_i : t_i \quad t_1, \ldots, t_{n_o} \overset{o}{\rightarrow} t}{\Gamma \vdash o(e_1, \ldots, e_{n_o}) : t} \text{ for } o \in \mathcal{O}$$

which types operators using their associated typing function. Last but not least, the rule to type pattern matching expressions is:

$$\frac{\begin{array}{ll} t \leq \wr p_1 \int \vee \wr p_2 \int & \Gamma \vdash e : t \\ t_1 \equiv t \wedge \wr p_1 \int \quad t_2 \equiv t \wedge \neg \wr p_1 \int & \Gamma_i \equiv \square | t_i / p_i \quad \Gamma'_i \equiv \Sigma_i; \square | t_i /\!\!/ p_i \\ \Sigma_i \equiv \{\dot{x} \mapsto \mathsf{Init}(\dot{x}) \mid \dot{x} \in \mathrm{Acc}(p_i)\} & \Gamma \cup \Gamma_i \cup \Gamma'_i \vdash e_i : t'_i \end{array}}{\Gamma \vdash \mathsf{match}\ e\ \mathsf{with}\ p_1 \rightarrow e_1 \,|\, p_2 \rightarrow e_2 : \bigvee_{\{i \mid t_i \not\simeq \mathbb{0}\}} t'_i} \ (i = 1, 2)$$

This rule requires that the type t of the matched expression is smaller than $\wr p_1 \int \vee \wr p_2 \int$ (*i.e.*, the set of all values accepted by any of the two patterns), that is, that the matching is exhaustive. Then, it accounts for the first match policy by checking e_1 in an environment inferred from values produced by e and that match p_1 ($t_1 \equiv t \wedge \wr p_1 \int$) and by checking e_2 in an environment inferred from values produced by e and that *do not* match p_1 ($t_2 \equiv t \wedge \neg\wr p_1 \int$). If one of these branches is unused (*i.e.*, if $t_i \simeq \mathbb{0}$ where $\simeq$ denotes semantic equivalence, that is, $\leq \cap \geq$), then its type does not contribute to the type of the whole expression (*cf.* §4.1 of [4] to see why, in general, this must not yield an "unused case" error). Each right-hand side e_i is typed in an environment enriched with the types for capture variables (computed by $\square | t_i / p_i$) and the types for accumulators (computed by $\Sigma_i; \square | t_i /\!\!/ p_i$). While the latter is specific to our calculus, the former is standard except it is parameterized by a zipper type as for the semantics of pattern matching (its precise computation is described in [9] and already implemented in the $\mathbb{C}$Duce compiler except the zipper-related part: see Figure 11 in Appendix for the details). As before, we write $\tau^?$ to denote an optional zipper type, *i.e.,* either τ or a none type $\square$, and consider $(t)_\square$ to be t.

To compute the types of the accumulators of a pattern p when matched against a type t, we first initialize an environment Σ by associating each accumulator $\dot{x}$ occurring in p with the singleton type for its initial value $\mathsf{Init}(\dot{x})$ ($\Sigma_i \equiv \{\dot{x} \mapsto \mathsf{Init}(\dot{x}) \mid \dot{x} \in \mathrm{Acc}(p_i)\}$). The type environment is then computed by generating a set of mutually recursive equations where the important ones are (see Figure 12 in Appendix for the complete definition):

$$\begin{array}{lll}
\Sigma; \tau^? | t /\!\!/ \dot{x} & = \Sigma[s/\dot{x}] & \text{if } (t)_{\tau^?}, \Sigma(\dot{x}) \xrightarrow{\mathsf{Op}(\dot{x})} s \\
\Sigma; \tau^? | t /\!\!/ p_1 \,|\, p_2 & = \Sigma; \tau^? | t /\!\!/ p_1 & \text{if } t \leq \wr p_1 \int \\
\Sigma; \tau^? | t /\!\!/ p_1 \,|\, p_2 & = \Sigma; \tau^? | t /\!\!/ p_2 & \text{if } t \leq \neg\wr p_1 \int \\
\Sigma; \tau^? | t /\!\!/ p_1 \,|\, p_2 & = (\Sigma; \tau^? | (t \wedge \wr p_1 \int) /\!\!/ p_1) \bigsqcup (\Sigma_1; \tau^? | (t \wedge \neg\wr p_1 \int) /\!\!/ p_2) & \text{otherwise}
\end{array}$$

When an accumulator $\dot{x}$ is matched against a type t, the type of the accumulator is updated in Σ, by applying the typing function of the operator associated with $\dot{x}$ to the type $(t)_{\tau^?}$ and the type computed thus far for $\dot{x}$, namely $\Sigma(\dot{x})$. The other equations recursively apply the matching on the subcomponents while updating the zipper type argument $\tau^?$ and merge the results using the "$\sqcup$" operation. This operation implements the fact that if an accumulator $\dot{x}$ has type t_1 in a subpart of a pattern p and type t_2 in another subpart (*i.e.*, both subparts match), then the type of $\dot{x}$ is the union $t_1 \vee t_2$.

The equations for computing the type environment for accumulators might be *not* well-founded. Both patterns and types are possibly infinite (regular) terms and therefore one has to guarantee that the set of generated equations is finite. This depends on the typing of the operators used for the accumulators. Before stating the termination condition (as well as the soundness properties of the type system), we give the typing functions for the operators we defined earlier.

Function application: it is typed by computing the minimum type satisfying the following subtyping relation: $s, t \overset{\mathsf{app}}{\to} \mathsf{min}\{t' \mid s \leq t \to t'\}$, provided that $s \leq t \to \mathbb{1}$ (this min always exists and is computable: see [10]).

Projection: to type the first and second projections, we use the property that if $t \leq \mathbb{1} \times \mathbb{1}$, then t can be decomposed in a finite union of product types (we use Π_i to denote the set of the i-th projections of these types: see Lemma 19 in Appendix B for the formal definition): $t \overset{\pi_i}{\to} \bigvee_{s \in \Pi_i(t)} s$, provided that $t \leq \mathbb{1} \times \mathbb{1}$.

Zipper erasure: the top-level erasure $\overset{\mathsf{rm}}{\to}$ simply removes the top-level zipper type annotation, while the deep erasure $\overset{\mathsf{drm}}{\to}$ is typed by recursively removing the zipper annotations from the input type. Their precise definition can be found in Appendix B.4.

Sequence building: it is typed in the following way:

$$\begin{array}{lcl} t_1, \texttt{`nil} & \overset{\mathsf{cons}}{\to} & \mu X.((t_1 \times X) \vee \texttt{`nil}) \\ t_1, \mu X.((t_2 \times X) \vee \texttt{`nil}) & \overset{\mathsf{cons}}{\to} & \mu X.(((t_1 \vee t_2) \times X) \vee \texttt{`nil}) \\ \\ t_1, \texttt{`nil} & \overset{\mathsf{snoc}}{\to} & \mu X.((t_1 \times X) \vee \texttt{`nil}) \\ t_1, \mu X.((t_2 \times X) \vee \texttt{`nil}) & \overset{\mathsf{snoc}}{\to} & \mu X.(((t_1 \vee t_2) \times X) \vee \texttt{`nil}) \end{array}$$

Notice that the output types are approximations: the operator "$\mathsf{cons}(\_)$" is *less* precise than returning a pair of two values since, for instance, it approximates any sequence type by an infinite one (meaning that any information on the length of the sequence is lost) and approximates the type of all the elements by a single type which is the union of all the elements (meaning that the information on the order of elements is lost). As we show next, this loss of precision is instrumental in typing accumulators and therefore pattern matching.

Example 8. Consider the matching of a pattern p against a value v of type t:

$$\begin{array}{rcl} p & \equiv & \mu X.((\dot{x}\,\&\,(\texttt{`a}|\texttt{`b}))\,|\,\texttt{`nil}\,|\,(X, X)) \\ v & \equiv & (\texttt{`a}, ((\texttt{`a}, (\texttt{`nil}, (\texttt{`b}, \texttt{`nil}))), (\texttt{`b}, \texttt{`nil}))) \\ t & \equiv & \mu Y.((\texttt{`a} \times (Y \times (\texttt{`b} \times \texttt{`nil}))) \vee \texttt{`nil}) \end{array}$$

where $\mathsf{Op}(\dot{x}) = \mathsf{snoc}$ and $\mathsf{Init}(\dot{x}) = \texttt{`nil}$. We have the following matching and type environment:

$$\begin{array}{l} \{\dot{x} \mapsto \texttt{`nil}\}; \square \vdash v/p \rightsquigarrow \{\dot{x} \mapsto (\texttt{`a}, (\texttt{`a}, (\texttt{`b}, (\texttt{`b}, \texttt{`nil}))))\}, \varnothing \\ \{\dot{x} \mapsto \texttt{`nil}\}; \square \,|\, t /\!\!/ p \ = \ \{\dot{x} \mapsto \mu Z.(((\texttt{`a} \vee \texttt{`b}) \times Z) \vee \texttt{`nil})\} \end{array}$$

Intuitively, with the usual sequence notation (precisely defined in Section 4), v is nothing but the nested sequence $[\,\texttt{`a}\ [\,\texttt{`a}\ [\,]\ \texttt{`b}\,]\ \texttt{`b}\,]$ and pattern matching just flattens the input sequence, binding $\dot{x}$ to $[\,\texttt{`a}\ \texttt{`a}\ \texttt{`b}\ \texttt{`b}\,]$. The type environment for $\dot{x}$ is computed by recursively matching each product type in t with the pattern (X, X), the singleton type $\texttt{`a}$ or $\texttt{`b}$ with $\dot{x}\,\&\,(\texttt{`a}|\texttt{`b})$, and $\texttt{`nil}$ with $\texttt{`nil}$. Since the operator associated with $\dot{x}$ is snoc and the initial type is $\texttt{`nil}$, when $\dot{x}$ is matched against $\texttt{`a}$ for the first time, its type is updated to $\mu Z.((\texttt{`a} \times Z) \vee \texttt{`nil})$. Then, when $\dot{x}$ is matched against $\texttt{`b}$, its type is updated to the final output type which is the encoding of $[\,(\text{`a} \vee \text{`b})*\,]$. Here, the approximation in the typing function for snoc is important because the exact type of $\dot{x}$ is the union for $n{\in}\mathbb{N}$ of $[\,\text{`a}^n\ \text{`b}^n\,]$, that is, the sequences of `a's followed by the same number of `b's, which is beyond the expressivity of regular tree languages.

We conclude this section with statements for type soundness of our calculus (see Appendix C for more details).

Definition 9 (Sound operator). *An operator* $(o, n, \overset{o}{\rightsquigarrow}, \overset{o}{\rightarrow})$ *is* sound *if and only if* $\forall v_1, \ldots, v_{n_o} \in \mathcal{V}$ *such that* $\vdash v_1 : t_1, \ldots, \vdash v_{n_o} : t_{n_o}$*, if* $t_1, \ldots, t_{n_o} \overset{o}{\rightarrow} s$ *and* $v_1, \ldots, v_{n_o} \overset{o}{\rightsquigarrow} e$ *then* $\vdash e : s$.

Theorem 10 (Type preservation). *If all operators in the language are* sound, *then typing is preserved by reduction, that is, if* $e \rightsquigarrow e'$ *and* $\vdash e : t$*, then* $\vdash e' : t$*. In particular,* $e' \neq \Omega$.

Theorem 11. *The operators app,* π_1*,* π_2*, drm, rm, cons, and snoc are sound.*

4 Surface Language

In this section, we define the "surface" language, which extends our core calculus with several constructs:

- Sequence expressions, regular expression types and patterns
- Sequence concatenation and iteration
- XML types, XML document fragment expressions
- XPath-like patterns

While most of these traits are syntactic sugar or straightforward extensions, we took special care in their design so that: *(i)* they cover various aspects of XML programming and *(ii)* they are expressive enough to encode a large fragment of XQuery 3.0.

Sequences: we first add sequences to expressions

$$e \quad ::= \quad \ldots \quad | \quad [\,e \, \cdots \, e\,]$$

where a sequence expression denotes its encoding *à la* Lisp, that is, $[\,e_1 \cdots e_n\,]$ is syntactic sugar for $(e_1, (\ldots, (e_n, \text{`nil})))$.

Regular expression types and patterns: regular expressions over types and patterns are defined as

$$\begin{array}{llllllllllll} \text{(Regexp. over types)} & R & ::= & t & | & R\,|\,R & | & R\,R & | & R* & | & \epsilon \\ \text{(Regexp. over patterns)} & r & ::= & p & | & r\,|\,r & | & r\,r & | & r* & | & \epsilon \end{array}$$

with the usual syntactic sugar: $R? \equiv R\,|\,\epsilon$ and $R+ \equiv R\,R*$ (likewise for regexps on patterns). We then extend the grammar of types and patterns as follows:

$$t \quad ::= \quad \ldots \quad | \quad [\,R\,] \qquad\qquad p \quad ::= \quad \ldots \quad | \quad [\,r\,]$$

Regular expression types are encoded using recursive types (similarly for regular expression patterns). For instance, $[\,\text{int}*\,\text{bool}?\,]$ can be rewritten into the recursive type $\mu X.\text{`nil} \vee (\text{bool} \times \text{`nil}) \vee (\text{int} \times X)$.

Sequence concatenation is added to the language in the form of a binary infix operator _ @ _ defined by:

$$\begin{array}{lcl} \texttt{`nil}, v & \overset{@}{\leadsto} & v \\ (v_1, v_2), v & \overset{@}{\leadsto} & (v_1, v_2 \,@\, v) \end{array} \qquad [R_1],[R_2] \overset{@}{\to} [R_1 R_2]$$

Note that this operator is sound but cannot be used to accumulate in patterns (since it does not guarantee the termination of type environment computation). However, it has an exact typing.

Sequence iteration is added to iterate transformations over sequences without resorting to recursive functions. This is done by a family of "`transform`"-like operators $\mathsf{trs}_{p_1,p_2,e_1,e_2}(\_)$, indexed by the patterns and expressions that form the branches of the transformation (we omit trs's indexes in $\overset{\mathsf{trs}}{\leadsto}$):

$$\begin{array}{lcl} \texttt{`nil} & \overset{\mathsf{trs}}{\leadsto} & \texttt{`nil} \\ (v_1, v_2) & \overset{\mathsf{trs}}{\leadsto} & (\mathsf{match}\ v_1\ \mathsf{with}\ p_1 \to e_1 \mid p_2 \to e_2) \,@\, \mathsf{trs}_{p_1,p_2,e_1,e_2}(v_2) \end{array}$$

Intuitively, the construct "$\mathsf{transform}\ e\ \mathsf{with}\ p_1 \to e_1 \mid p_2 \to e_2$" iterates all the "branches" over each element of the sequence e. Each branch may return a sequence of results which is concatenated to the final result (in particular, a branch may return "`` `nil ``" to delete elements that match a particular pattern).

XML types, patterns, and document fragments: XML types (and thus patterns) can be represented as a pair of the type of the label and a sequence type representing the sequence of children, annotated by the zipper that denotes the position of document fragment of that type. We denote by $<t_1>{t_2}_\tau$ the type $(t_1 \times t_2)_\tau$, where $t_1 \leq \mathbb{1}_{\mathsf{basic}}$, $t_2 \leq [\mathbb{1}*]$, and τ is a zipper type. We simply write $<t_1>t_2$ when $\tau = \top$, that is, when we do not have (or do not require) any information on the zipper type. The invariant that XML values are always given with respect to a zipper must be maintained at the level of expressions. This is ensured by extending the syntax of expressions with the construct:

$$e \ ::= \ \ldots \ \mid \ <e>e$$

where $<e_1>e_2$ is syntactic sugar for $(e_1, \mathsf{drm}(e_2))_\bullet$. The reason for this encoding is best understood with the following example:

Example 12. Consider the code:

```
match v with
  ( <a>[ _ x _* ] )⊤ -> <b>[ x ]
| _ -> <c>[ ]
```

According to our definition of pattern matching, x is bound to the second XML child of v and retains its zipper (in the right-hand side, we could navigate from x up to v or even above if v is not the root). However, when x is embedded into another document fragment, the zipper must be erased so that accessing the element associated with x in the *new* value can create an appropriate zipper (with respect to its new root <b>[...]).

$$
\begin{aligned}
\mathsf{self}_0\{x \mid t\} &\equiv \dot{x} \,\&\, t \mid \_ \\
\mathsf{self}\{x \mid t\} &\equiv (\mathsf{self}_0\{x \mid t\})_{\top} \\
\mathsf{child}\{x \mid t\} &\equiv (\mathsf{<\_>}[\,(\mathsf{self}_0\{x \mid t\})*\,] \mid \_)_{\top} \\
\mathsf{desc\text{-}or\text{-}self}_0\{x \mid t\} &\equiv \mu X.(\mathsf{self}_0\{x \mid t\} \,\&\, \mathsf{<\_>}[\,X*\,]) \mid \_ \\
\mathsf{desc\text{-}or\text{-}self}\{x \mid t\} &\equiv (\mathsf{desc\text{-}or\text{-}self}_0\{x \mid t\})_{\top} \\
\mathsf{desc}\{x \mid t\} &\equiv (\mathsf{<\_>}[\,(\mathsf{desc\text{-}or\text{-}self}_0\{x \mid t\})*\,] \mid \_)_{\top} \\
\mathsf{foll\text{-}sibling}\{x \mid t\} &\equiv (\_)_{\mathsf{L}\,(\_,\,[\,(\mathsf{self}_0\{x \mid t\})*\,])_{\top}\cdot\top} \\
\mathsf{parent}\{y \mid t\} &\equiv (\_)_{\mathsf{L}\,\_\cdot\,\mu X.((\mathsf{R}\,(\dot{y}\,\&\,t \mid \_)_{\top}\cdot(\mathsf{L}\,\_\cdot\top \mid \bullet)) \mid \mathsf{R}\,\_\cdot X)} \mid \_ \\
\mathsf{prec\text{-}sibling}\{y \mid t\} &\equiv (\_)_{\mathsf{L}\,\_\cdot\,\mu X.(\mathsf{R}\,(\dot{y}\,\&\,t,\,\_)_{\top}\cdot X) \mid (\mathsf{R}\,\_\cdot(\mathsf{L}\,\_\cdot\top \mid \bullet))} \mid \_ \\
\mathsf{anc}\{y \mid t\} &\equiv (\_)_{\mathsf{L}\,\_\cdot\,\mu X.\mu Y.((\mathsf{R}\,(\dot{y}\,\&\,t \mid \_)_{\top}\cdot(\mathsf{L}\,\_\cdot X \mid \bullet)) \mid \mathsf{R}\,\_\cdot Y)} \mid \_ \\
\mathsf{anc\text{-}or\text{-}self}\{y \mid t\} &\equiv (\mathsf{self}\{y \mid t\} \,\&\, \mathsf{anc}\{y \mid t\}) \mid \_
\end{aligned}
$$

where $\mathsf{Op}(\dot{x}) = \mathsf{snoc}$, $\mathsf{Init}(\dot{x}) = \text{‘}\mathtt{nil}$, $\mathsf{Op}(\dot{y}) = \mathsf{cons}$, and $\mathsf{Init}(\dot{y}) = \text{‘}\mathtt{nil}$

Fig. 5. Encoding of axis patterns

XPath-like patterns are one of the main motivations for this work. The syntax of patterns is extended as follows:

$$
\begin{array}{lrcl}
\text{(Patterns)} & p & ::= & \ldots \mid \mathit{axis}\{x \mid t\} \\
\text{(Axes)} & \mathit{axis} & ::= & \mathsf{self} \mid \mathsf{child} \mid \mathsf{desc} \mid \mathsf{desc\text{-}or\text{-}self} \mid \mathsf{foll\text{-}sibling} \\
 & & \mid & \mathsf{parent} \mid \mathsf{anc} \mid \mathsf{anc\text{-}or\text{-}self} \mid \mathsf{prec\text{-}sibling}
\end{array}
$$

The semantics of $\mathit{axis}\{x \mid t\}$ is to capture in x all fragments of the matched document along the *axis* that have type t. We show in Appendix D how the remaining two axes (following and preceding) as well as "multi-step" XPath expressions can be compiled into this simpler form. We encode axis patterns directly using recursive patterns and accumulators, as described in Figure 5. First, we remark that each pattern has a default branch "$\ldots \mid \_$" which implements the fact that even if a pattern fails, the value is still accepted, but the default value ‘`nil` of the accumulator is returned. The so-called "downward" axes —self, child, desc-or-self, and desc— are straightforward. For self, the encoding checks that the matched value has type t using the auxiliary pattern self_0, and that the value is annotated with a zipper using the zipper type annotation $(\_)_{\top}$. The child axis is encoded by iterating self_0 on every child element of the matched value. The recursive axis desc-or-self is encoded using the auxiliary pattern $\mathsf{desc\text{-}or\text{-}self}_0$ which matches the root of the current element (using self_0) and is recursively applied to each element of the sequence. Note the double recursion: vertically in the tree using a recursive binder and horizontally at a given level using a star. The non-reflexive variant desc evaluates $\mathsf{desc\text{-}or\text{-}self}_0$ on every child element of the input.

The other axes heavily rely on the binary encoding of XML values and are better explained on an example. Consider the XML document and its binary tree representation given in Figure 6. The following siblings of a node (*e.g.*, `<c>`) are reachable by inspecting the first element of the zipper, which is necessarily an L one. This parent is the pair representing the sequence whose tail is the sequence of following siblings (R_3 and R_2 in the figure). Applying the $\mathsf{self}\{x \mid t\}$ axis on each element of the tail therefore filters the following siblings that are sought (`<d>` and `<e>` in the figure). The parent axis is more involved. Consider

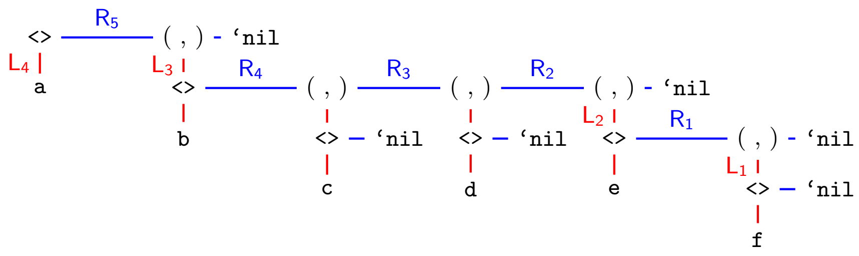

Fig. 6. A binary tree representation of an XML document
`doc = <a>[ <b>[ <c>[ ] <d>[ ] <e>[ <f>[ ] ] ] ]`

for instance node `<e>`. Its parent in the XML tree can be found in the zipper associated with `<e>`. It is the last R component of the zipper before the next L component (in the figure, the zipper of `<e>` starts with L_2, then contains its previous siblings reachable by R_2 and R_3, and lastly its parent reachable by R_4 (which points to node `<b>`). The encoding of the parent axis reproduces this walk using a recursive zipper pattern, whose base case is the last R before the next L, or the last R before the root (which has the empty zipper •). The prec-sibling axis uses a similar method and collects every node reachable by Rs and stops before the parent node (again, for node `<e>`, the preceding siblings are reached by R_2 and R_3). The anc axis simply iterates the parent axis recursively until there is no L zipper anymore (*i.e.*, until the root of the document has been reached). In the example, starting from node `<f>`, the zippers that denote the ancestors are the ones starting with an R, just before L_2, L_3, and L_4 which is the root of the document. Lastly, anc-or-self is simply a combination of anc and self.

For space reasons, the encoding of XPath into the navigational patterns is given in Appendix D. We just stress that, with that encoding, the ℂDuce version of the "*get_links*" function of the introduction becomes as compact as in XQuery:

```
let get_links (page: <_>_) (print: <a>_ -> <a>_) : [ <a>_ * ] =
          transform page/desc::a[not(anc::b)] with x -> [ (print x) ]
```

As a final remark, one may notice that patterns of forward axes use snoc (*i.e.*, they build the sequence of the results in order), while reverse axes use cons (thus reversing the results). The reason for this difference is to implement the semantics of XPath axis steps which return elements *in document order*.

5 XQuery 3.0

This section shows that our surface language can be used as a compilation target for XQuery 3.0 programs. We proceed in two steps. First, we extend the XQuery 1.0 Core fragment and $\mathsf{XQ_H}$ defined by Benedikt and Vu [3] to our own XQuery 3.0 Core, which we call $\mathsf{XQ_H^+}$. As with its 1.0 counterpart, $\mathsf{XQ_H^+}$

1. can express *all* navigational XQuery programs, and
2. explicitly separates navigational aspects from data value ones.

$$\begin{array}{rcll}
\mathit{query} & ::= & () \mid c \mid \texttt{<}l\texttt{>}\mathit{query}\texttt{</}l\texttt{>} & \mid\ \mathit{query}, \mathit{query} \mid x \mid x/\mathit{axis}::\mathit{test} \\
 & \mid & \mathsf{for}\ x\ \mathsf{in}\ \mathit{query}\ \mathsf{return}\ \mathit{query} & \mid\ \mathsf{some}\ x\ \mathsf{in}\ \mathit{query}\ \mathsf{statisfies}\ \mathit{query} \\
 & \mid & \mathit{query}(\mathit{query}, \ldots, \mathit{query}) & \mid\ \mathsf{fun}\ x_1 : t_1, \ldots, x_n : t_n\ \mathsf{as}\ t.\ \mathit{query} \\
 & \mid & \begin{array}{l}\mathsf{switch}\ \mathit{query} \\ \quad \mathsf{case}\ c\ \mathsf{return}\ \mathit{query} \\ \quad \mathsf{default\ return}\ \mathit{query}\end{array} & \mid\ \begin{array}{l}\mathsf{typeswitch}\ \mathit{query} \\ \quad \mathsf{case}\ t\ \mathsf{as}\ x\ \mathsf{return}\ \mathit{query} \\ \quad \mathsf{default\ return}\ \mathit{query}\end{array} \\
\mathit{test} & ::= & \texttt{node()} \mid \texttt{text()} \mid l & \qquad\qquad\qquad \text{(node test)}
\end{array}$$

where t ranges over types and l ranges over element names.

Fig. 7. Syntax of XQ_H^+

We later use the above separation in the translation to straightforwardly map navigational XPath expressions into extended ℂDuce pattern matching, and to encode data value operations (for which there can be no precise typing) by built-in ℂDuce functions.

5.1 XQuery 3.0 Core

Figure 7 shows the definition of XQ_H^+, an *extension* of XQ_H. To the best of our knowledge, XQ_H was the first work to propose a "Core" fragment of XQuery which abstracts away most of the idiosyncrasies of the actual specification while retaining essential features (*e.g.*, path navigation). XQ_H^+ differs from XQ_H by the last three productions (in the yellow/gray box): it extends XQ_H with type and value cases (described informally in the introduction) and with *type annotations* on functions (which are only optional in the standard). It is well known (*e.g.*, see [24]) that full XPath expressions can be encoded using the XQuery fragment in Figure 7 (see Appendix E for an illustration).

Our translation of XQuery 3.0, defined in Figure 8, thus focuses on XQ_H^+ and has following characteristics. If one considers the "typed" version of the standard, that is, XQuery programs where function declarations have an explicit signature, then the translation to our surface language *(i)* provides a formal semantics and a typechecking algorithm for XQuery and *(ii)* enjoys the soundness property that the original XQuery programs do not yield runtime errors. In the present work, we assume that the type algebra of XQuery is the one of ℂDuce, rather than XMLSchema. Both share regular expression types for which subtyping is implemented as the inclusion of languages, but XMLSchema also features *nominal subtyping*. The extension of ℂDuce types with nominal subtyping is beyond the scope of this work and is left as future work.

In XQuery, all values are sequences: the constant "42" is considered as the *singleton sequence* that contains the element "42". As a consequence, there are only "flat" sequences in XQuery and the only way to create nested data structures is to use XML constructs. The difficulty for our translation is thus twofold: *(i)* it needs to embed/extract values explicitly into/from sequences and *(ii)* it also needs to disambiguate types: an XQuery function that takes an integer as argument can also be applied to a sequence containing only one integer.

$$\begin{array}{rcl}
\llbracket \texttt{()} \rrbracket_{\mathsf{X}\mathbb{C}} & = & \texttt{`nil} \\
\llbracket c \rrbracket_{\mathsf{X}\mathbb{C}} & = & [\,c\,] \\
\llbracket \texttt{<}l\texttt{>}q\texttt{</}l\texttt{>} \rrbracket_{\mathsf{X}\mathbb{C}} & = & [\,\texttt{<}l\texttt{>}\llbracket q \rrbracket_{\mathsf{X}\mathbb{C}}\,] \\
\llbracket q_1 , q_2 \rrbracket_{\mathsf{X}\mathbb{C}} & = & \llbracket q_1 \rrbracket_{\mathsf{X}\mathbb{C}} \;\texttt{@}\; \llbracket q_2 \rrbracket_{\mathsf{X}\mathbb{C}} \\
\llbracket \$x \rrbracket_{\mathsf{X}\mathbb{C}} & = & x
\end{array}
\qquad
\begin{array}{rcl}
\left\llbracket \begin{array}{l} \mathsf{switch}\ q_1 \\ \quad \mathsf{case}\ c\ \mathsf{return}\ q_2 \\ \quad \mathsf{default\ return}\ q_3 \end{array} \right\rrbracket_{\mathsf{X}\mathbb{C}} & = & \begin{array}{l} \mathsf{match}\ \llbracket q_1 \rrbracket_{\mathsf{X}\mathbb{C}}\ \mathsf{with} \\ \quad [\,c\,] \to \llbracket q_2 \rrbracket_{\mathsf{X}\mathbb{C}} \\ \quad \mid \ \_ \ \to \llbracket q_3 \rrbracket_{\mathsf{X}\mathbb{C}} \end{array} \\
\left\llbracket \begin{array}{l} \mathsf{typeswitch}\ q_1 \\ \quad \mathsf{case}\ t\ \mathsf{as}\ \$x\ \mathsf{return}\ q_2 \\ \quad \mathsf{default\ return}\ q_3 \end{array} \right\rrbracket_{\mathsf{X}\mathbb{C}} & = & \begin{array}{l} \mathsf{match}\ \llbracket q_1 \rrbracket_{\mathsf{X}\mathbb{C}}\ \mathsf{with} \\ \quad x \,\&\, \mathsf{seq}(t) \to \llbracket q_2 \rrbracket_{\mathsf{X}\mathbb{C}} \\ \quad \mid \ \_ \to \llbracket q_3 \rrbracket_{\mathsf{X}\mathbb{C}} \end{array}
\end{array}$$

$$\begin{array}{rcl}
\llbracket \$x/\mathit{axis}::\mathit{test} \rrbracket_{\mathsf{X}\mathbb{C}} & = & \mathsf{transform}\ x\ \mathsf{with}\ \mathit{axis}\{y \mid \mathsf{t}(\mathit{test})\} \to y \\
\llbracket \mathsf{for}\ \$x\ \mathsf{in}\ q_1\ \mathsf{return}\ q_2 \rrbracket_{\mathsf{X}\mathbb{C}} & = & \mathsf{transform}\ \llbracket q_1 \rrbracket_{\mathsf{X}\mathbb{C}}\ \mathsf{with}\ x \to \llbracket q_2 \rrbracket_{\mathsf{X}\mathbb{C}} \\
\llbracket \mathsf{some}\ \$x\ \mathsf{in}\ q_1\ \mathsf{statisfies}\ q_2 \rrbracket_{\mathsf{X}\mathbb{C}} & = & \begin{array}[t]{l} \mathsf{match}\ (\ \mathsf{transform}\ \llbracket q_1 \rrbracket_{\mathsf{X}\mathbb{C}}\ \mathsf{with} \\ \qquad\quad x \to \mathsf{match}\ \llbracket q_2 \rrbracket_{\mathsf{X}\mathbb{C}}\ \mathsf{with} \\ \qquad\qquad\qquad [\,\texttt{`true}\,] \ \to [\,\texttt{`dummy}\,] \\ \qquad\qquad\quad \mid [\,\texttt{`false}\,] \to [\,]\ ) \\ \mathsf{with}\ \ \texttt{`nil} \to [\,\texttt{`false}\,] \ \mid \ \_ \ \to [\,\texttt{`true}\,] \end{array} \\
\llbracket \mathsf{fun}\ \$x_1 : t_1, \ldots, \$x_n : t_n\ \mathsf{as}\ t.\ q \rrbracket_{\mathsf{X}\mathbb{C}} & = & \begin{array}[t]{l} \mu\_^{\mathsf{seq}(t_1) \times \ldots \times \mathsf{seq}(t_n) \to \mathsf{seq}(t)}(x_0). \\ \qquad \mathsf{match}\ x_0\ \mathsf{with}\ (x_1, (\ldots, x_n)) \to \llbracket q \rrbracket_{\mathsf{X}\mathbb{C}} \end{array} \\
\llbracket q(q_1, \ldots, q_n) \rrbracket_{\mathsf{X}\mathbb{C}} & = & \llbracket q \rrbracket_{\mathsf{X}\mathbb{C}}\ \ (\llbracket q_1 \rrbracket_{\mathsf{X}\mathbb{C}}, (\ldots, \llbracket q_n \rrbracket_{\mathsf{X}\mathbb{C}}))
\end{array}$$

where $\mathsf{seq}(t) \equiv (t \wedge [\,\mathbb{1}*]) \vee ([\,t \setminus [\,\mathbb{1}*]\,])$
and $\mathsf{t}(\texttt{node()}) \equiv \mathbb{1}$, $\mathsf{t}(\texttt{text()}) \equiv \texttt{String}$, $\mathsf{t}(l) \equiv \texttt{<}l\texttt{>}\mathbb{1}$

Fig. 8. Translation of $\mathsf{XQ}_{\mathsf{H}}^{+}$ into ℂDuce

The translation is defined by a function $\llbracket\_\rrbracket_{\mathsf{X}\mathbb{C}}$ that converts an XQuery query into a ℂDuce expression. It is straightforward and ensures that the result of a translation $\llbracket q \rrbracket_{\mathsf{X}\mathbb{C}}$ always has a sequence type. We assume that both languages have the same set of variables and constants. An empty sequence is translated into the atom `'nil`, a constant is translated into a singleton sequence containing that constant, and similarly for XML fragments. The sequence operator is translated into concatenation. Variables do not require any special treatment. An XPath navigation step is translated into the corresponding navigational pattern, whereas "for in" loops are encoded similarly using the transform construct (in XQuery, an XPath query applied to a sequence of elements is the concatenation of the individual applications). The "switch" construct is directly translated into a "match with" construct. The "typeswitch" construct works in a similar way but special care must be taken with respect to the type t that is tested. Indeed, if t is a sequence type, then its translation returns the sequence type, but if t is something else (say int), then it must be embedded into a sequence type. Interestingly, this test can be encoded as the ℂDuce type $\mathsf{seq}(t)$ which keeps the part of t that is a sequence unchanged while embedding the part of t that is not a sequence (namely $t \setminus [\,\mathbb{1}*]$) into a sequence type (*i.e.*, $[\,t \setminus [\,\mathbb{1}*]\,]$). The "some $\$x$ in q_1 statisfies q_2" expression iterates over the sequence that is the result of the translation of q_1, binding variable x in turn to each element, and evaluates (the translation of) q_2 in this context. If the evaluation of q_2 yields the singleton sequence `true`, then we return a dummy non-empty sequence; otherwise, we return the empty sequence. If the whole transform yields an empty sequence,

$$\frac{\Gamma \vdash_{\mathrm{XQ}} q : s \quad s \leq t}{\Gamma \vdash_{\mathrm{XQ}} q : t} \qquad \frac{\Gamma \vdash_{\mathrm{XQ}} q_1 : [\,s*] \quad \Gamma, x : [\,s] \vdash_{\mathrm{XQ}} q_2 : t \quad t \leq [\,\mathbb{1}*]}{\Gamma \vdash_{\mathrm{XQ}} \mathsf{for}\ \$x\ \mathsf{in}\ q_1\ \mathsf{return}\ q_2 : t}$$

$$\frac{\begin{array}{c}\{\dot{y} \mapsto \texttt{`nil}\}; \square | s /\!\!/ \mathit{axis}\{y \mid \mathsf{t}(\mathit{test})\} = \{\dot{y} \mapsto t\} \\ \Gamma \vdash_{\mathrm{XQ}} x : [\,s*] \quad t \leq [\,\mathbb{1}*] \quad t' = \mathsf{min}\{t' \mid t \leq [\,t'*]\}\end{array}}{\Gamma \vdash_{\mathrm{XQ}} \$x/\mathit{axis}::\mathit{test} : [\,t'*]}\ \textsf{typ-path}$$

$$\frac{\Gamma \vdash_{\mathrm{XQ}} q : t \quad \begin{cases} t \not\leq \neg[\,c] \Rightarrow \Gamma \vdash_{\mathrm{XQ}} q_1 : s \\ t \not\leq [\,c] \quad \Rightarrow \Gamma \vdash_{\mathrm{XQ}} q_2 : s \end{cases}}{\Gamma \vdash_{\mathrm{XQ}} \begin{array}{l}\mathsf{switch}\ q \\ \mathsf{case}\ c\ \mathsf{return}\ q_1 \\ \mathsf{default\ return}\ q_2\end{array} : s} \qquad \frac{\begin{array}{c} t_1 = s \wedge \mathsf{seq}(t) \quad \Gamma, x : t_1 \vdash_{\mathrm{XQ}} q_1 : t_1' \\ \Gamma \vdash_{\mathrm{XQ}} q : s \quad t_2 = s \wedge \neg\mathsf{seq}(t) \quad \Gamma \vdash_{\mathrm{XQ}} q_2 : t_2' \end{array}}{\Gamma \vdash_{\mathrm{XQ}} \begin{array}{l}\mathsf{typeswitch}\ q \\ \mathsf{case}\ t\ \mathsf{as}\ \$x\ \mathsf{return}\ q_1 \\ \mathsf{default\ return}\ q_2\end{array} : \bigvee_{\{i \mid t_i \not\simeq \mathbb{0}\}} t_i'}$$

$$\frac{\Gamma \vdash_{\mathrm{XQ}} q_1 : [\,s*] \quad \Gamma, x : [\,s] \vdash_{\mathrm{XQ}} q_2 : [\,\mathsf{bool}]}{\Gamma \vdash_{\mathrm{XQ}} \mathsf{some}\ \$x\ \mathsf{in}\ q_1\ \mathsf{statisfies}\ q_2 : [\,\mathsf{bool}]}$$

$$\frac{\Gamma, x_1 : \mathsf{seq}(t_1), \cdots, x_n : \mathsf{seq}(t_n) \vdash_{\mathrm{XQ}} q : \mathsf{seq}(t)}{\Gamma \vdash_{\mathrm{XQ}} \mathsf{fun}\ \$x_1 : t_1, \ldots, \$x_n : t_n\ \mathsf{as}\ t.\ q \ : \ \mathsf{seq}(t_1) \times \cdots \times \mathsf{seq}(t_n) \to \mathsf{seq}(t)}$$

$$\frac{\Gamma \vdash_{\mathrm{XQ}} q : t_1 \times \cdots \times t_n \to t \quad \Gamma \vdash_{\mathrm{XQ}} q_i : t_i \quad (i = 1..n)}{\Gamma \vdash_{\mathrm{XQ}} q(q, \ldots, q) : t}$$

Fig. 9. Typing rules for $\mathsf{XQ}_{\mathsf{H}}^{+}$

it means that none of the iterated elements matched satisfied the predicate q_2 and therefore the whole expression evaluates to the singleton `false`, otherwise it evaluates to the singleton `true`. Abstractions are translated into ℂDuce functions, and the same treatment of "sequencing" the type is applied to the types of the arguments and type of the result. Lastly, application is translated by building nested pairs with the arguments before applying the function.

Not only does this translation ensure soundness of the original XQuery 3.0 programs, it also turns ℂDuce into a sandbox where one can experiment various typing features that can be readily back-ported to XQuery afterwards.

5.2 Toward and beyond XQuery 3.0

We now discuss the salient features and address some shortcomings of $\mathsf{XQ}_{\mathsf{H}}^{+}$. First and foremost, we can define a precise and sound type system directly on $\mathsf{XQ}_{\mathsf{H}}^{+}$ as shown in Figure 9 (standard typing rules are omitted and for the complete definition, see Appendix E). While most constructs are typed straightforwardly (the typing rules are deduced from the translation of $\mathsf{XQ}_{\mathsf{H}}^{+}$ into ℂDuce) it is interesting to see that the rules match those defined in XQuery Static Semantics specification [24] (with the already mentioned difference that we use ℂDuce types instead of XMLSchema). Two aspects however diverge from the standard. Our use of ℂDuce's semantic subtyping (rather than XMLSchema's nominal subtyping), and the rule typ-path where we use the formal developments of Section 3 to provide a precise typing rule for XPath navigation. Deriving the typing rules from our translation allows us to state the following theorem:

Theorem 13. *If* $\Gamma \vdash_{\mathrm{XQ}} \mathit{query} : t$, *then* $\Gamma \vdash [\![\mathit{query}]\!]_{\mathsf{X}\mathbb{C}} : t$.

A corollary of this theorem is the soundness of the XQ^+_H type system (since the translation of a well-typed XQ^+_H program yields a well-typed ℂDuce program with the same type).

While the XQ^+_H fragment we present here is already very expressive, it does not account for all features of XQuery. For instance, it does not feature data value comparison or sorting (*i.e.*, the order by construct of XQuery) nor does it account for built-in functions such as `position()`, node identifiers, and so on. However, it is known that features such as data value comparison make typechecking undecidable (see for instance [1]). We argue that the main point of this fragment is to cleanly separate structural path navigation from other data value tests for which we can add built-in operators and functions, with an hardcoded, *ad-hoc* typing rule.

Lastly, one may argue that, in practice, XQuery database engines do *not* rely on XQuery Core for evaluation but rather focus on evaluating efficiently large (multi-step, multi-predicate) XPath expressions in one go and, therefore, that normalizing XQuery programs into XQ^+_H programs and then translating the latter into ℂDuce programs may seem overly naive. We show in Appendix D that XPath expressions that are purely navigational can be rewritten in a single pattern of the form: $axis\{x \mid t\}$ which can then be evaluated very efficiently (that is, without performing the unneeded extra traversals of the document that a single step approach would incur).

6 Related Work and Conclusion

Our work tackles several aspects of XML programming, the salient being: *(i)* encoding of XPath or XPath-like expressions (including reverse axes) into regular types and patterns, *(ii)* recursive tree transformation using accumulators and their typing, and *(iii)* type systems and typechecking algorithms for XQuery.

Regarding XPath and pattern matching, the work closest to ours is the implementation of paths as patterns in XTatic. XTatic [11] is an object-oriented language featuring XDuce regular expression types and patterns [16,17]. In [12], Gapeyev and Pierce alter XDuce's pattern matching semantics and encode a fragment of XPath as patterns. The main difference with our work is that they use a hard-coded all-match semantics (a variable can be bound to several subterms) to encode the accumulations of recursive axes, which are restricted by their data model to the "child" and "descendant" axes. Another attempt to use path navigation in a functional language can be found in [19] where XPath-like combinators are added to Haskell. Again, only child or descendant-like navigation is supported and typing is done in the setting of Haskell which cannot readily be applied to XML typing (results are returned as *homogeneous* sequences).

Our use of accumulators is reminiscent of Macro Tree Transducers (MTTs, [8]), that is, tree transducers (tree automata producing an output) that can also accumulate part of the input and copy it in the output. It is well known that given an input regular tree language, the type of the accumulators and results may not be regular. Exact typing may be done in the form of backward type inference, where the output type is given and a largest input type is inferred [20].

It would be interesting to use the backward approach to type our accumulators without the approximation introduced for "`cons`" for instance.

For what concerns XQuery and XPath, several complementary works are of interest. First, the work of Genevès *et al.* which encodes XPath and XQuery in the μ-calculus ([14,15] where zippers to manage XPath reverse axes were first introduced) supports our claim. Adding path expressions at the level of *types* is not more expensive: subtyping (or equivalently satisfiability of particular formulæ of the μ-calculus which are equivalent to regular tree languages) remains EXPTIME, even with upward paths (or in our case, zipper types). In contrast, typing path expressions and more generally XQuery programs is still a challenging topic. While the W3C's formal semantics of XQuery [24] gives a polynomial time typechecking algorithm for XQuery (in the absence of nested "`let`" or "`for`" constructs), it remains far too imprecise (in particular, reverse axes are left untyped). Recently, Genevès *et al.* [13] also studied a problem of typing reverse axes by using regular expressions of μ-calculus formulæ as types, which they call focused-tree types. Since, as our zipped types, focused-tree types can describe both the type of the current node and its context, their type system also gives a precise type for reverse axis expressions. However, while focused-tree types are more concise than zipper types, it is difficult to type construction of a new XML document, and thus their type system requires an explicit type annotation for each XML element. Furthermore, their type system does not feature arrow types. That said, it will be quite interesting to combine their approach with ours.

We are currently implementing axis patterns and XPath expressions on top of the ℂDuce compiler. Future work includes extensions to other XQuery constructs as well as XMLSchema, the addition of aggregate functions by associating accumulators to specific operators, the inclusion of navigational expressions in types so as to exploit the full expressivity of our zipped types (*e.g.*, to type functions that work on the ancestors of their arguments), and the application of the polymorphic type system of [5,6] to both XQuery and navigational ℂDuce so that for instance the function *pretty* defined in the introduction can be given the following, far more precise intersection of two arrow types:

```
(<a class="style1" href=β ..>γ -> <a href=β>[<b>γ])
   & (α\<a class="style1" href=_ ..>_ -> α\<a class="style1" href=_ ..>_ )
```

This type (where α, β, and γ denote universally quantified type variables) precisely describes, by the arrow type on the first line, the transformation of the sought links, and states, by the arrow on the second line, that in all the other cases (*i.e.*, for every type α different from the sought link) it returns the same type as the input. This must be compared with the corresponding type in Figure 1, where the types of the attribute `href`, of the content of the `a` element, and above all of any other value not matched by the first branch are not preserved.

Acknowledgments. We want to thank the reviewers of ESOP who gave detailed suggestions to improve our presentation. This work was partially supported by the ANR TYPEX project n. ANR-11-BS02-007.

References

1. Alon, N., Milo, T., Neven, F., Suciu, D., Vianu, V.: XML with data values: Typechecking revisited. In: PODS, pp. 138–149. ACM (2001)
2. Amadio, R.M., Cardelli, L.: Subtyping recursive types. ACM Trans. Program. Lang. Syst. 15(4), 575–631 (1993)
3. Benedikt, M., Vu, H.: Higher-order functions and structured datatypes. In: WebDB, pp. 43–48 (2012)
4. Benzaken, V., Castagna, G., Frisch, A.: CDuce: An XML-centric general-purpose language. In: ICFP, pp. 51–63 (2003)
5. Castagna, G., Nguyễn, K., Xu, Z., Abate, P.: Polymorphic functions with set-theoretic types. Part 2: Local type inference and type reconstruction. In: POPL, pp. 289–302 (2015)
6. Castagna, G., Nguyễn, K., Xu, Z., Im, H., Lenglet, S., Padovani, L.: Polymorphic functions with set-theoretic types. Part 1: Syntax, semantics, and evaluation. In: POPL, pp. 5–17 (2014)
7. Castagna, G., Xu, Z.: Set-theoretic foundation of parametric polymorphism and subtyping. In: ICFP, pp. 94–106 (2011)
8. Engelfriet, J., Vogler, H.: Macro tree transducers. J. Comput. Syst. Sci. 31(1), 71–146 (1985)
9. Frisch, A.: Théorie, conception et réalisation d'un langage adapté à XML. PhD thesis, Université Paris 7 Denis Diderot (2004)
10. Frisch, A., Castagna, G., Benzaken, V.: Semantic subtyping: Dealing set-theoretically with function, union, intersection, and negation types. J. ACM 55(4), 1–64 (2008)
11. Gapeyev, V., Garillot, F., Pierce, B.C.: Statically typed document transformation: An Xtatic experience. In: PLAN-X (2006)
12. Gapeyev, V., Pierce, B.C.: Paths into patterns. Technical Report MS-CIS-04-25, University of Pennsylvania (October 2004)
13. Genevès, P., Gesbert, N., Layaïda, N.: Xquery and static typing: Tackling the problem of backward axes (July 2014), `http://hal.inria.fr/hal-00872426`
14. Genevès, P., Layaïda, N.: Eliminating dead-code from XQuery programs. In: ICSE (2010)
15. Genevès, P., Layaïda, N., Schmitt, A.: Efficient static analysis of XML paths and types. In: PLDI (2007)
16. Hosoya, H., Pierce, B.C.: Regular expression pattern matching for XML. J. Funct. Program. 13(6), 961–1004 (2003)
17. Hosoya, H., Pierce, B.C.: XDuce: A statically typed XML processing language. ACM Trans. Internet Technol. 3(2), 117–148 (2003)
18. Huet, G.: The Zipper. J. Funct. Program. 7(5), 549–554 (1997)
19. Lämmel, R.: Scrap your boilerplate with XPath-like combinators. In: POPL (2007)
20. Maneth, S., Berlea, A., Perst, T., Seidl, H.: XML type checking with macro tree transducers. In: PODS (2005)
21. W3C: XPath 1.0 (1999), `http://www.w3.org/TR/xpath`,
22. W3C: XPath 2.0 (2010), `http://www.w3.org/TR/xpath20`,
23. W3C: XML Query (2010), `http://www.w3.org/TR/xquery`,
24. XQuery 1.0 and XPath 2.0 Formal Semantics, 2nd edn (2010), `http://www.w3.org/TR/xquery-semantics/`
25. W3C: XQuery 3.0 (2014), `http://www.w3.org/TR/xquery-3.0`
26. W3C: XML Schema (2009), `http://www.w3.org/XML/Schema`