A Core Calculus for XQuery 3.0

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. 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 [24] is a declarative language standardized by the W3C that relies heavily on XPath [22, 23] as a data extraction primitive. Interestingly, the next version of XQuery (version 3.0, currently being drafted [26]) 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 [27]), 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 [25] which gives sound (but sometime imprecise) typing rules for XQuery.

In contrast, ℂDuce [3] 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* [9] 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.

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.

XQuery code:

```
1  declare function get_links($page, $print)
2  {
3    for $i in $page/descendant::a[not(ancestor::b)]
4    return $print($i)
5  }
6
7  declare function pretty($link)
8  {
9    typeswitch($link)
10   case $l as element(a)
11           return switch ($l/@class)
12                  case "style1"
13                         return <a href={$l/@href}>
14                                   <b>{$l/text()}</b>
15                                </a>
16                  default return $l
17   default return $link
18 }
```

ℂDuce code:

```
19 let get_links : <_>_ → (<a>_ → <a>_) → [ <a>_ * ] =
20 fun page -> fun print ->
21  match page with
22   <a>_ & x -> [ (print x) ]
23 | <_\b> l -> transform l with i -> get_links i print
24 |  _ -> [ ]
25
26 let pretty : (<a>_ → <a>_) & (Any\<a>_ → Any\<a>_) =
27 fun link ->
28  match link with
29   <a class="style1" href=h> l -> <a href=h>[ <b>l ]
30 |  x -> x
```

Figure 1. Document transformation, in XQuery 3.0 and ℂDuce

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).

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).

Contributions. Our contribution is to 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 [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 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.

Another contribution is the extension of 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.

On the XQuery side, we extend $\mathsf{XQ_H}$, a core version of XQuery 3.0 proposed by Benedikt and Vu [2], 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. *A minima*, it is straightforward to use this basis to add overloaded functions to XQuery. More crucially, the recent advances on polymorphism for semantic subtyping [6, 28] 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. In the meanwhile, to palliate the absence of a full-fledged polymorphic system, we show that the typing operators introduced here can be used to provide a poor man's version of parametric polymorphism, a most needed feature when dealing with functional programs.

Plan. The remainder of the paper is structured as follows. Section 2 presents the core typed λ-calculus equipped with zipper annotated values, accumulators, constructors, recursive functions, and pattern matching. Section 3 then 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 then uses this language as a compilation target for XQuery. Section 6 gives some directions towards which XQuery and ℂDuce's type system can be extended, and lastly Section 7 compares our work to other related approaches and concludes. Proofs and some technical definitions are given in the supplementary appendix attached to this submission.

2. Syntax

We extend the ℂDuce language [3] with zippers *à la* Huet [18]. To ensure the well-foundness of the definition, we stratify it, introducing first pre-values (which are normal ℂDuce values) and then values, which are pre-values possibly indexed by a zipper; we proceed similarly for types and patterns.

Definition 2.1 (Pre-value, value, and zipper).

$$\begin{array}{lll} \textbf{Pre-values} & w & ::= \; c \mid (w,w) \mid \mu f^{(t\to t;\dots;t\to t)}(x).e \\ \textbf{Zippers} & \delta & ::= \; \bullet \mid \mathsf{L}\,(w)_\delta \cdot \delta \mid \mathsf{R}\,(w)_\delta \cdot \delta \\ \textbf{Values} & v & ::= \; w \mid (v,v) \mid (w)_\delta \end{array}$$

We denote by $\mathcal{V}$ the set of all values. We denote by Ω a special value that represents a runtime error and does not inhabit any type.

[1] We took some liberties with ℂDuce's syntax in order to better match XQuery's one. The actual ℂDuce definitions are more compact and can be found in Appendix A (included in the submission supplementary material).

Pre-values (ranged over by w) are the usual ℂDuce values without zipper annotations. Constants are ranged over by c and represent integers ($\mathtt{1}, \mathtt{2}, \dots$), characters (`'a'`, `'b'`, ...), atoms (`` `nil ``, `` `true ``, `` `false ``, `` `foo ``, ...), and so on. The value (w, w) represents pairs of pre-values. Lastly, the calculus 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 only need 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 choose 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 2.2. 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, that is, having $\mathsf{L}\,w \cdot \delta$ in the Definition 2.1 would require a more complex treatment to reconstruct the parent).

Definition 2.3 (Expression). An *expression* is a finite term produced by the following grammar, with entry point e:

e	$::=$	v	(value)
	$\mid$	x	(variable)
	$\mid$	$\dot{x}$	(accumulator)
	$\mid$	(e, e)	(pair)
	$\mid$	$(e)_\bullet$	(initial context)
	$\mid$	$\mathsf{match}\ e\ \mathsf{with}\ p \to e \mid p \to e$	(pattern matching)
	$\mid$	$\mathsf{op}(e, \dots, e)$	(operator)

We denote by $\mathcal{E}$ the set of all expressions.

Basic expressions may be values (as previously defined), variables (ranged over by x, y, ...), accumulators (which are a particular kind of variables, ranged over by $\dot{x}$, $\dot{y}$, ...), or pairs. The pattern matching expression is the usual one (with a first match policy) and will be thoroughly presented in the following section. Lastly, our calculus is parameterized by a set $\mathcal{O}$ of built-in operators ranged over by op. 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 define our type algebra.

2.1 Types

We first recall the usual ℂDuce type algebra, as defined in [9], 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 $s \le t \overset{\text{def}}{\iff} [\![s]\!] \subseteq [\![t]\!]$).

Definition 2.4 (Pre-type). A pre-type u is a possibly infinite term produced by the following grammar, with entry point u:

$$u \ ::= \ b \mid c \mid u \times u \mid u \to u \mid u \vee u \mid \neg u \mid \mathbb{0}$$

with two additional requirements:

1. (regularity) the number of distinct subterms of u is finite;
2. (contractivity) 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 and terminate, return 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. Contractivity [1] 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 $u_1 \le u_2$, if all values in u_1 are also in u_2) and it is decidable in EXPTIME (see [9]). Before defining full types (*i.e.*, the type of values potentially annotated with zippers), we introduce *zipper types*.

Definition 2.5 (Zipper type). A *zipper type* τ is a possibly infinite term produced by the following grammar, with entry point τ:

$$\tau \quad ::= \quad \bullet \mid \top \mid \mathsf{L}\,(u)_\tau \cdot \tau \mid \mathsf{R}\,(u)_\tau \cdot \tau \mid \tau \vee \tau \mid \neg\tau$$

that is regular as in Definition 2.4 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 $\neg(\neg\tau_1 \vee \neg\tau_2)$.

Remark 2.6. Zipper type negation is redundant since it can be encoded by the remaining zipper types. This could be done by using classic regular language techniques but would also cause a (further) exponential explosion of the subtyping algorithms, which is why we prefer to include negation directly in the syntax. In contrast, negation in (pre-)types cannot be encoded because of the presence of arrow (pre-)types.

We now define the type algebra of our core calculus which contains pre-types possibly indexed by a zipper type.

Definition 2.7 (Type). A *type* is a possibly infinite term produced by the following grammar, with entry point t:

$$t \ ::= \ u \mid t \times t \mid t \to t \mid t \vee t \mid \neg t \mid (u)_\tau$$

that is both regular and contractive as in Definition 2.4.

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 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.

It is straightforward to extend the subtyping relation of pre-types (*i.e.*, of [9]) to types: the addition of the $(u)_\tau$ corresponds to the addition of a new type constructor (such as $\to$ 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 judgments are defined in Figure 7 in the Appendix). From this we deduce that $(\mathbb{1})_\top$ (equivalently, $(\mathbb{1}_{\mathsf{NZ}})_\top$) is the type of all (pre-)values decorated by a zipper. The formal definition is more involved (see Appendix B) 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 it 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 can be 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 B.3 in the Appendix).

2.2 Operators and Accumulators

As previously explained, our calculus is parameterized by a set $\mathcal{O}$ of operators which have the following formal definition:

Definition 2.8 (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 allow operators to be non-deterministic (for instance, to simulate non-deterministic choice). Note that an operator may fail, thus returning the special value Ω during evaluation.

Definition 2.9 (Accumulator). An *accumulator* $\dot{x}$ is a variable equipped with a binary operator $\mathsf{Op}(\dot{x}) \in \mathcal{O}$ and initial value $\mathsf{Init}(\dot{x}) \in \mathcal{V}$.

2.3 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 (pre-patterns), enriched with accumulators, before extending them with zippers.

Definition 2.10 (Pre-pattern). A pre-pattern is a possibly infinite term produced by the following grammar, with entry point q:

$$\begin{array}{rcll} q & ::= & t & \text{(type constraint)} \\ & | & x & \text{(capture variable)} \\ & | & \dot{x} & \text{(accumulator)} \\ & | & (q, q) & \text{(pair)} \\ & | & q \,|\, q & \text{(or/alternative)} \\ & | & q \,\&\, q & \text{(and/conjunction)} \\ & | & (x := c) & \text{(default)} \end{array}$$

that is regular as in Definition 2.4 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.

$$\begin{array}{rl} E[\,] ::= & [\,] \mid (E[\,], e) \mid (e, E[\,]) \mid (E[\,])_\bullet \mid o(e, \ldots, E[\,], \ldots, e) \\ \mid & \mathsf{match}\ E[\,]\ \mathsf{with}\ p_1 \to e_1 \,|\, p_2 \to e_2 \end{array}$$

where $o \in \mathcal{O}$

$$\frac{(v_1, \ldots, v_{n_o}) \overset{o}{\rightsquigarrow} e}{o(v_1, \ldots, v_{n_o}) \rightsquigarrow e}\ \text{for } o \in \mathcal{O} \qquad \frac{e \rightsquigarrow e'}{E[e] \rightsquigarrow E[e']}$$

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

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

$$\frac{}{e \rightsquigarrow \Omega}\quad \text{(if no other rule applies and } e \text{ is not a value)}$$

Figure 2. Operational semantics (reduction contexts and rules)

Definition 2.11 (Zipper pattern). A *zipper pattern* is a possibly infinite term produced by the following grammar

$$\varphi \quad ::= \quad \tau \quad | \quad \mathsf{L}\, p \cdot \varphi \quad | \quad \mathsf{R}\, p \cdot \varphi \quad | \quad \varphi \,|\, \varphi$$

that is both regular and contractive as in Definition 2.5.

Definition 2.12 (Pattern). A pattern is a possibly infinite term produced by the following grammar, with entry point p:

$$\begin{array}{rcll} p & ::= & q \mid (p, p) \mid p \,|\, p \mid p \,\&\, p & \\ & | & (q)_\varphi & \text{(zipper constraint)} \end{array}$$

with the same requirements as in Definition 2.10. Besides, the subpatterns forming zipper constraints 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.

3. Semantics

In this section, 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 2, where Ω is a special value representing a run-time 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, returns two substitutions, respectively, from capture variables and accumulators to values. Before explaining in detail the rules for pattern matching, we introduce a minimal set of operators (for application, projections, zipper erasure, and sequence building). In what follows we give only their reduction relation and delay the presentation of their typing relation to Section 3.2.

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

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

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 $\mathsf{app}(e_1, e_2)$.

Projections: the operator $\pi_1(\_)$ (resp. $\pi_2(\_)$) implements the first (resp. second) projection of pairs. When applied to a pair annotated by a zipper, they update the zipper accordingly:

$$\begin{array}{lcll} (w_1, w_2)_\delta & \overset{\pi_1}{\rightsquigarrow} & (w_1)_{\mathsf{L}\,(w_1,w_2)_\delta \cdot \delta} & \\ (w_1, w_2)_\delta & \overset{\pi_2}{\rightsquigarrow} & (w_2)_{\mathsf{R}\,(w_1,w_2)_\delta \cdot \delta} & \\ (v_1, v_2) & \overset{\pi_i}{\rightsquigarrow} & v_i & \text{for } i \in \{1, 2\} \end{array}$$

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

Zipper erasure: given a zipper annotated value, it is sometimes necessary to remove the zipper (for instance, to embed this value into a new data structure). This is achieved by the remove $\mathsf{rm}(\_)$ and deep remove $\mathsf{drm}(\_)$ operators defined as follows:

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

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:

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

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 (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 *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.

The semantics of pattern matching is given by the judgment $\sigma \vdash v/p \rightsquigarrow \sigma', \gamma$, 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. The judgment is derived by the rules given in Figure 3. In this figure, rules [PAT-ACC] and [ZPAT-*] are novel, extending pattern matching with accumulators and zippers, while the others are derived from [3, 8]. 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 projection of the value succeeds (rule [PAT-PAIR]). Note that we use operator $\pi_i(\_)$ to update the zipper annotation of the value, if any. An alternative pattern $p_1 | p_2$ 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]).

The matching of a zipper pattern φ against a zipper δ 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 $\mathrm{Acc}(p)$.

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 with some examples.

Example 3.1. Let $v \equiv (1, (\texttt{`true}, (3, \texttt{`nil})))$, $\mathsf{Init}(\dot{x}) = \texttt{`nil}$, $\mathsf{Op}(\dot{x}) = \mathsf{cons}$, and $\sigma \equiv \{\dot{x} \mapsto \texttt{`nil}\}$. Then, we have

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

In the first case, the input v (the sequence `[1 `true 3]` encoded *à la* Lisp) is matched against a pattern that checks whether 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, thus yielding a mapping $\{x \mapsto (3, \texttt{`nil})\}$ (third set in the definition of $\oplus$ in Figure 3). 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 1 and a pair is formed from this binding and the one found in the recursive call, yielding the final binding $\{x \mapsto (1, (3, \texttt{`nil}))\}$.

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 is to type pattern matching and, more specifically, to infer the types of the accumulators occurring in patterns.

Definition 3.2 (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:

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

$$\textsc{[Pat-Type]}\ \frac{}{\sigma \vdash v/t \rightsquigarrow \sigma, \varnothing}\ (\vdash v : t) \qquad \textsc{[Pat-Var]}\ \frac{}{\sigma \vdash v/x \rightsquigarrow \sigma, \{x \mapsto v\}} \qquad \textsc{[Pat-Acc]}\ \frac{}{\sigma \vdash v/\dot{x} \rightsquigarrow \sigma[{}^{\mathsf{Op}(\dot{x})(v,\sigma(\dot{x}))}/_{\dot{x}}], \varnothing}$$

$$\textsc{[Pat-Pair]}\ \frac{\sigma \vdash \pi_1(v)/p_1 \rightsquigarrow \sigma', \gamma_1 \qquad \sigma' \vdash \pi_2(v)/p_2 \rightsquigarrow \sigma'', \gamma_2}{\sigma \vdash v/(p_1, p_2) \rightsquigarrow \sigma'', \gamma_1 \oplus \gamma_2} \qquad \textsc{[Pat-Or1]}\ \frac{\sigma \vdash v/p_1 \rightsquigarrow \sigma', \gamma}{\sigma \vdash v/p_1 | p_2 \rightsquigarrow \sigma', \gamma} \qquad \textsc{[Pat-Or2]}\ \frac{\sigma \vdash v/p_1 \rightsquigarrow \Omega \qquad \sigma \vdash v/p_2 \rightsquigarrow \sigma', \gamma}{\sigma \vdash v/p_1 | p_2 \rightsquigarrow \sigma', \gamma}$$

$$\textsc{[Pat-And]}\ \frac{\sigma \vdash v/p_1 \rightsquigarrow \sigma', \gamma_1 \qquad \sigma' \vdash v/p_2 \rightsquigarrow \sigma'', \gamma_2}{\sigma \vdash v/p_1 \,\&\, p_2 \rightsquigarrow \sigma'', \gamma_1 \oplus \gamma_2} \qquad \textsc{[Pat-Def]}\ \frac{}{\sigma \vdash v/(x := c) \rightsquigarrow \sigma, \{x \mapsto c\}} \qquad \textsc{[Pat-Zip]}\ \frac{\sigma \vdash w/q \rightsquigarrow \sigma', \gamma_1 \qquad \sigma' \vdash \delta/\varphi \rightsquigarrow \sigma'', \gamma_2}{\sigma \vdash (w)_\delta/(q)_\varphi \rightsquigarrow \sigma'', \gamma_1 \oplus \gamma_2}$$

$$\textsc{[ZPat-Type]}\ \frac{}{\sigma \vdash \delta/\tau \rightsquigarrow \sigma, \varnothing}\ (\vdash \delta : \tau) \qquad \textsc{[ZPat-Left]}\ \frac{\sigma \vdash (w)_\delta/p \rightsquigarrow \sigma', \gamma_1 \qquad \sigma' \vdash \delta/\varphi \rightsquigarrow \sigma'', \gamma_2}{\sigma \vdash \mathsf{L}\,(w)_\delta \cdot \delta/\mathsf{L}\,p \cdot \varphi \rightsquigarrow \sigma'', \gamma_1 \oplus \gamma_2} \qquad \textsc{[ZPat-Right]}\ \frac{\sigma \vdash (w)_\delta/p \rightsquigarrow \sigma', \gamma_1 \qquad \sigma' \vdash \delta/\varphi \rightsquigarrow \sigma'', \gamma_2}{\sigma \vdash \mathsf{R}\,(w)_\delta \cdot \delta/\mathsf{R}\,p \cdot \varphi \rightsquigarrow \sigma'', \gamma_1 \oplus \gamma_2}$$

$$\textsc{[ZPat-Or1]}\ \frac{\sigma \vdash \delta/\varphi_1 \rightsquigarrow \sigma', \gamma}{\sigma \vdash \delta/\varphi_1 | \varphi_2 \rightsquigarrow \sigma', \gamma} \qquad \textsc{[ZPat-Or2]}\ \frac{\sigma \vdash \delta/\varphi_1 \rightsquigarrow \Omega \qquad \sigma \vdash \delta/\varphi_2 \rightsquigarrow \sigma', \gamma}{\sigma \vdash \delta/\varphi_1 | \varphi_2 \rightsquigarrow \sigma', \gamma} \qquad \textsc{[Pat-Error]}\ \frac{}{\sigma \vdash v/p \rightsquigarrow \Omega}\ \text{(otherwise)} \qquad \textsc{[ZPat-Error]}\ \frac{}{\sigma \vdash \delta/\varphi \rightsquigarrow \Omega}\ \text{(otherwise)}$$

$$\gamma_1 \oplus \gamma_2 \overset{\text{def}}{=} \{x \mapsto \gamma_1(x) \mid x \in \mathsf{dom}(\gamma_1)\backslash\mathsf{dom}(\gamma_2)\} \cup \{x \mapsto \gamma_2(x) \mid x \in \mathsf{dom}(\gamma_2)\backslash\mathsf{dom}(\gamma_1)\} \cup \{x \mapsto (\gamma_1(x), \gamma_2(x)) \mid x \in \mathsf{dom}(\gamma_1) \cap \mathsf{dom}(\gamma_2)\}$$

Figure 3. Pattern matching

Definition 3.3 (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 values that make a pattern succeed:

Theorem 3.4 (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. The set of all values v such that $\{\dot{x} \mapsto \mathit{Init}(\dot{x}) \mid \dot{x} \in \mathrm{Acc}(p)\} \vdash v/p \not\rightsquigarrow \Omega$ is a type. We call this set the* accepted type *of p and denote it by $\wr p \int$. It can be computed by solving the following guarded system of equations (the variables are the $\wr p' \int$ and $\wr \varphi \int$ for the subterms p' and φ of p).*

$$\begin{array}{lcl@{\qquad}lcl}
\wr t \int & = & t & \wr \tau \int & = & \tau \\
\wr x \int & = & \mathbb{1} & \wr \mathsf{L}\,p \cdot \varphi \int & = & \mathsf{L}\,\wr p \int \cdot \wr \varphi \int \\
\wr \dot{x} \int & = & \mathbb{I}(\mathit{Op}(\dot{x})) & \wr \mathsf{R}\,p \cdot \varphi \int & = & \mathsf{R}\,\wr p \int \cdot \wr \varphi \int \\
\wr (p_1, p_2) \int & = & \wr p_1 \int \times \wr p_2 \int & \wr \varphi_1 \vee \varphi_2 \int & = & \wr \varphi_1 \int \vee \wr \varphi_2 \int \\
\wr p_1 | p_2 \int & = & \wr p_1 \int \vee \wr p_2 \int & & & \\
\wr p_1 \,\&\, p_2 \int & = & \wr p_1 \int \wedge \wr p_2 \int & & & \\
\wr (x := c) \int & = & \mathbb{1} & & & \\
\wr (q)_\varphi \int & = & (\wr q \int)_{\wr \varphi \int} & & &
\end{array}$$

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 to types) an expression e has type t. This judgment is derived by the set of rules given in Figure 7 in Appendix. Here, we show only the most important rules, namely those for accumulators and zippers:

$$\textsc{[T-Acc]}\ \frac{}{\Gamma \vdash \dot{x} : \Gamma(\dot{x})} \qquad \textsc{[T-Zip-Val]}\ \frac{\vdash w : t \quad \vdash \delta : \tau \quad t \leq \mathbb{1}_{\mathsf{NZ}}}{\Gamma \vdash (w)_\delta : (t)_\tau} \qquad \textsc{[T-Zip-Expr]}\ \frac{\vdash e : t \quad t \leq \mathbb{1}_{\mathsf{NZ}}}{\Gamma \vdash (e)_\bullet : (t)_\bullet}$$

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

$$\textsc{[T-Op]}\ \frac{\forall i = 1..n_o,\ \Gamma \vdash e_i : t_i \quad t_1, \ldots, t_{n_o} \xrightarrow{o} t}{\Gamma \vdash o(e_1, \ldots, e_{n_o}) : t}\ \text{for } o \in \mathcal{O}$$

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

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

This rule requires that the type t of the matched expression be smaller than $\wr p_1 \int \vee \wr p_2 \int$, that is, that the matching be 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. Each right-hand side e_i is typed in an environment enriched with the types for capture variables (computed by t_i/p_i) and the types for accumulators (computed by $t_i \underset{\Sigma_i}{/\!\!/} p_i$). While the former is rather standard (its precise computation is described in [8] and already implemented in the ℂDuce compiler: see Figure 9 in Appendix for the details), the latter is specific to our calculus.

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 $\dot{x}$ of 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 whose the important ones are (see Figure 8 in Appendix for the complete definition):

$$\begin{array}{lcl@{\qquad}r}
t \underset{\Sigma}{/\!\!/} \dot{x} & = & \Sigma[{}^{s}/_{\dot{x}}] & \text{where } t, \Sigma(\dot{x}) \xrightarrow{\mathsf{Op}(\dot{x})} s \\
t \underset{\Sigma}{/\!\!/} p_1 | p_2 & = & t \underset{\Sigma}{/\!\!/} p_1 & \text{if } t \leq \wr p_1 \int \\
t \underset{\Sigma}{/\!\!/} p_1 | p_2 & = & t \underset{\Sigma}{/\!\!/} p_2 & \text{if } t \leq \neg \wr p_1 \int \\
t \underset{\Sigma}{/\!\!/} p_1 | p_2 & = & (t \wedge \wr p_1 \int) \underset{\Sigma}{/\!\!/} p_1 \bigsqcup (t \wedge \neg \wr p_1 \int) \underset{\Sigma}{/\!\!/} p_2 & \text{otherwise}
\end{array}$$

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 and the type computed thus far for $\dot{x}$, namely $\Sigma(\dot{x})$. The other equations recursively apply the matching on the subcomponents 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 $t_1 \vee t_2$. Lastly, we introduce a technical notation for the matching of product types. Indeed, the most general type for any pair is a *finite* union of products (or, said differently, while it is possible to push intersections below product constructors, it is not possible to do it for unions without introducing an approximation since in general: $(a \times b) \vee (c \times d) \lneq (a \vee c) \times (b \vee d)$). This can be generalized to zipped product types as stated by this lemma:

Lemma 3.5 (Product decomposition). *Let t be a type such that $t \leq (\mathbb{1} \times \mathbb{1})_\top \vee (\mathbb{1} \times \mathbb{1})$. There exists a finite set of pairs of types*

$$\Pi(t) = \bigcup_{i \leq n} \{((u_1^i)_{\mathsf{L}\,(u_0^i)_{\tau^i} \cdot \tau^i},\ (u_2^i)_{\mathsf{R}\,(u_0^i)_{\tau^i} \cdot \tau^i})\} \cup \bigcup_{j \leq m} \{(t_1^j,\ t_2^j)\}$$

such that

$$\bigvee_{((u')_{\mathsf{L}\,\_\cdot\tau},(u'')_{\mathsf{R}\,\_\cdot\tau}) \in \Pi(t)} (u' \times u'')_\tau \vee \bigvee_{(t',t'') \in \Pi(t)} t' \times t'' \simeq t$$

Furthermore, given a decomposition $\Pi(t)$, we define the first and second type projection as:

$$\Pi_i(t) = \bigcup_{((u_1)_{\tau_1},(u_2)_{\tau_2}) \in \Pi(t)} \{(u_i)_{\tau_i}\} \cup \bigcup_{(t_1,t_2) \in \Pi(t)} \{t_i\}$$

There exists many such product decompositions. For instance, one decomposition is obtained by taking the syntactic expression given for a product type t and pushing intersections below products until only unions remain at top-level. More complex decompositions (which yield more precise typing or more efficient pattern matching) are described in [5, 8, 21].

As the knowledgeable reader may have guessed, the equations given in Figure 8 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 of course 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:

$$s, t \overset{\mathsf{app}}{\to} \min\{t' \mid s \leq t \to t'\}$$

provided that $s \leq t \to \mathbb{1}$ (see [9]).

Projections: the first and second projections are typed by using their type-level counter part:

$$t \overset{\pi_i}{\to} \bigvee_{s \in \Pi_i(t)} s$$

provided that $t \leq (\mathbb{1} \times \mathbb{1})_\top \vee \mathbb{1} \times \mathbb{1}$.

Zipper erasure: the top-level erasure simply removes the top-level zipper type annotation, while the deep erasure is typed by recursively removing the zipper annotations from the input type:

$$\begin{array}{lll} (t)_\tau & \overset{\mathsf{rm}}{\to}\ t & \text{if } t \wedge (\mathbb{1})_\top \simeq \mathbb{0} \\ t & \overset{\mathsf{rm}}{\to}\ (t \wedge \neg(\mathbb{1})_\top) \vee s & \text{where } t \wedge (\mathbb{1})_\top \overset{\mathsf{rm}}{\to} s \end{array}$$

$$\begin{array}{lll} t & \overset{\mathsf{drm}}{\to}\ t & \text{if } t \leq \mathbb{1}_{\mathsf{NZ}} \\ t & \overset{\mathsf{drm}}{\to}\ t \wedge (\mathbb{1}_{\mathsf{basic}} \vee \mathbb{1}_{\mathsf{fun}}) & \\ & \quad \vee \bigvee_{(t_1,t_2) \in \Pi(t \wedge (\mathbb{1}_{\mathsf{prod}} \vee (\mathbb{1}_{\mathsf{prod}})_\top))} t_1' \times t_2' & \text{where } t_i \overset{\mathsf{drm}}{\to} t_i' \\ & \quad \vee\ s & \text{where } t \wedge (\mathbb{1}_{\mathsf{NZ}})_\top \overset{\mathsf{rm}}{\to} s \end{array}$$

There are two cases for the deep erasure. If an input type does not contain any zipper type annotation (in-depth), it is left unchanged. Otherwise, the type is split into three parts. The first part consists of basic types and arrow types and is "copied" unchanged in the output type. The second part corresponds to the zipped product type and product type components of the union. In this case, the output type is the union of the products formed from the erasure of each component. The third part corresponds to zipped types in which the zipper type annotations are removed using the top-level erasure rm. Again, we need to ensure that the drm function terminates, that is, given a type t, the number of t_i in the second part is finite (we show this property in Section 3.3).

Sequence building: it is typed in the following way:

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

Notice that the output types are approximations. Indeed, the operator "$\mathsf{cons}(\_)$" is *less* precise than returning a pair of two values, for instance, since 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 element is lost). As we will see now, this loss of precision is instrumental in typing accumulators and therefore pattern matching.

Example 3.6. Consider the matching of a pattern p against a value v of type t defined as follows:

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

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

- $\{\dot{x} \mapsto \text{`}\mathtt{nil}\} \vdash v/p \rightsquigarrow \{\dot{x} \mapsto (\text{`}\mathtt{a}, (\text{`}\mathtt{a}, (\text{`}\mathtt{b}, (\text{`}\mathtt{b}, \text{`}\mathtt{nil}))))\}, \varnothing$
- $\underset{\{\dot{x} \mapsto \text{`}\mathtt{nil}\}}{t /\!\!/ p} = \{\dot{x} \mapsto \mu Z.(((\text{`}\mathtt{a} \vee \text{`}\mathtt{b}) \times Z) \vee \text{`}\mathtt{nil})\}$

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

3.3 Properties of the Type System

We first show that the algorithms we defined (such as the equations of Figure 8) or the typing of operator "$\mathsf{drm}(\_)$" terminates, even for infinite (regular) input. To that end, we define the notion of plinth.

Definition 3.7 (Plinth). Let $\mathcal{O}$ be a set of operators. A *plinth* $\sqsupset_{\mathcal{O}} \subset \mathcal{T}$ over $\mathcal{O}$ is a set of types with the following properties:

Finiteness $\sqsupset_{\mathcal{O}}$ is finite;

Boolean closure $\beth_{\mathcal{O}}$ contains $\mathbb{1}$ and $\mathbb{0}$ and is closed under Boolean connectives $(\wedge, \vee, \neg)$;

Stability w.r.t. operators for all operators $o \in \mathcal{O}$ and types $t_1, \ldots, t_{n_o} \in \beth_{\mathcal{O}}$, if $t_1, \ldots, t_{n_o} \xrightarrow{o} t$ then $t \in \beth_{\mathcal{O}}$.

Intuitively, the plinth is the approximation of the set of types that can be found by saturating an initial set with the operators of $\mathcal{O}$. This means that any algorithm visiting types produced by the application of such operators will visit only a finite number of types and therefore terminate.

Lemma 3.8. *Let $\mathcal{O}_{\mathbb{C}Duce}$ be the set*

$$\mathcal{O}_{\mathbb{C}Duce} \equiv \{\pi_1, \pi_2, \mathit{rm}, \mathit{cons}, \mathit{snoc}\}.$$

For all type t, there exists a plinth over $\mathcal{O}_{\mathbb{C}Duce}$ that contains t.

Corollary 3.9. *The typing of the operator* drm *terminates.*

Corollary 3.10. *For any type t and pattern p such that*

$$\forall \dot{x} \in \mathrm{Acc}(p), \mathit{Op}(\dot{x}) \in \mathcal{O}_{\mathbb{C}Duce}$$

the computation of $t \underset{\Sigma}{/\!/} p$ terminates, where

$$\Sigma \equiv \{\dot{x} \mapsto \mathit{Init}(\dot{x}) \mid \dot{x} \in \mathrm{Acc}(p)\}$$

Lemma 3.11. *All operators $\mathcal{O}_{\mathbb{C}Duce}$ have exact input.*

Note that the set of operators we consider do not include, *e.g.*, the function application operator. Indeed, in general, this operator is not stable w.r.t. a given set of types. Of course, it does not mean that our calculus does not feature function application, but only — and this is a rather reasonable restriction— that function application cannot be used as an operator for accumulators.

Before stating the soundness property for the whole language, we first need to define what it means for an operator to be sound and second to show that the set of operators we consider is sound.

Definition 3.12 (Sound operator). An operator $(o, n, \overset{o}{\rightsquigarrow}, \xrightarrow{o})$ 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 $t_1, \ldots, t_{n_o} \xrightarrow{o} s$ and $v_1, \ldots, v_{n_o} \overset{o}{\rightsquigarrow} e$ then $\vdash e : s$

This allows us to finally state the soundness of the whole language (through type preservation, as usual).

Theorem 3.13 (Type preservation). *If all operators in the language are* sound*, then typing is preserved by reduction:*

$$\textit{if } e \rightsquigarrow e' \textit{ and } \vdash e : t, \textit{ then } \vdash e' : t$$

In particular, $e' \not\rightsquigarrow \Omega$.

Note that since we made a runtime error (the special value Ω) explicit in the dynamic semantics, we do not need to show progress; showing type preservation is sufficient since Ω does not inhabit any type. All it remains to prove is that the operators we used are sound and whence deduce the soundness of the whole calculus.

Theorem 3.14. *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

$$e ::= \ \ldots \mid [\, e \cdots e \,]$$

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

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

(Regexp. over types)	R	$::=$	$t \mid R\,\vert\,R \mid R\,R \mid R{*} \mid \epsilon$
(Regexp. over patterns)	r	$::=$	$p \mid r\,\vert\,r \mid r\,r \mid r{*} \mid \epsilon$

with the usual syntactic sugar: $R? \equiv R \,\vert\, \epsilon$ and $R{+} \equiv R\,R{*}$. We then extend the grammar of types and patterns as follows:

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

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

$$\mu X.\texttt{`nil} \vee (\mathsf{bool} \times \texttt{`nil}) \vee (\mathsf{int} \times X).$$

Sequence concatenation is added to the language in the form of a binary infix operator $\_\,\texttt{@}\,\_$ defined by:

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

Note that this operator is sound but cannot be used to accumulate in patterns (since it does not satisfy the stability property of plinths). However, it has an exact typing.

Sequence iteration is added to the language to iterate transformations over sequences without resorting to recursive functions (which must be explicitly typed). 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:

$$\begin{array}{lcl} \texttt{`nil} & \overset{\mathsf{trs}}{\rightsquigarrow} & \texttt{`nil} \\ (v_1, v_2) & \overset{\mathsf{trs}}{\rightsquigarrow} & \left(\begin{array}{c} \mathsf{match}\ v_1\ \mathsf{with} \\ p_1 \rightarrow e_1 \\ \mid p_2 \rightarrow e_2 \end{array}\right) \texttt{@}\,\mathsf{trs}(v_2) \end{array}$$

Intuitively, the construct "$\mathsf{transform}\ e\ \mathsf{with}\ p_1 \rightarrow e_1 \mid p_2 \rightarrow e_2$" iterates 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 an empty sequence to delete some 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 $\texttt{<}t_1\texttt{>}t_{2\,\tau}$ 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 $\texttt{<}t_1\texttt{>}t_2$ 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 following construct

$$e ::= \ \ldots \mid \texttt{<}e\texttt{>}e$$

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

Example 4.1. Consider the code:

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

where: $\mathsf{Op}(\dot{x}) = \mathsf{snoc}$ and $\mathsf{Init}(\dot{x}) = $ `‘nil`
and: $\mathsf{Op}(\dot{y}) = \mathsf{cons}$ and $\mathsf{Init}(\dot{y}) = $ `‘nil`

Figure 4. Encoding of axis patterns

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

Due to our definition of pattern matching, x is bound to the second XML child of the input 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 (w.r.t. to its new root <b>[...]).

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

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

The semantics of $\mathit{axis}\{x \mid t\}$ is to capture in x all document fragments of the matched document along the *axis* that have type t. We show in Section 5.1 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 4. First, remark that each pattern has a default branch "...|_" 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 intersection checks that the matched value has type t. The child axis is encoded by iterating the self axis on every child element of the matched value. The recursive axis desc-or-self is encoded by a recursive pattern which matches the root of the current element (using a self pattern) and is recursively applied to each element of the sequence. Note the double recursion: vertically in the tree by using a recursive binder and horizontally at a given level using a star. The non-reflexive variant desc evaluates desc-or-self 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 5. 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 filters therefore the following siblings that are seeked (<d> and <e> in the figure). The parent axis is more involved. Consider 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 $\bullet$). The prec-sibling axis uses a similar method and collects every node reachable by an R 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 (that is 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 just the combination of anc and self.

As a remark, one may notice that patterns of forward axes use snoc (that is, build the sequence of the results in order), while upward axes use cons (thus reversing the results). The reason for this difference is to implement the semantics of XPath queries which returns elements *in document order*.

5. XQuery 3.0

In this section, we show that our surface language can be used as a compilation target for a sizable fragment of XQuery 3.0. The main difficulty resides in translating XPath queries into axis patterns, the translation of other XQuery constructs being straightforward.

5.1 XPath

In this work, we consider only the so-called *navigational* fragment of XPath that we can handle at the level of types and patterns. To support XPath queries involving data value comparisons and aggregate functions such as nth-child predicates, it suffices to break queries into structural parts (to be handled by patterns) and data value parts (to be handled by regular ℂDuce functions).

Definition 5.1 (XPath query). An *XPath query* is a finite term produced by the following grammar:

$$
\begin{array}{lcl}
\mathit{path} & ::= & \mathit{step} \mid \mathit{path}/\mathit{step}\\
\mathit{step} & ::= & \mathit{axis} :: \mathit{test}[\mathit{pred}]\\
\mathit{test} & ::= & l \mid *\\
\mathit{pred} & ::= & \mathit{path} \mid \mathtt{not}(\mathit{pred}) \mid \mathit{pred}\ \mathtt{and}\ \mathit{pred} \mid \mathit{pred}\ \mathtt{or}\ \mathit{pred}
\end{array}
$$

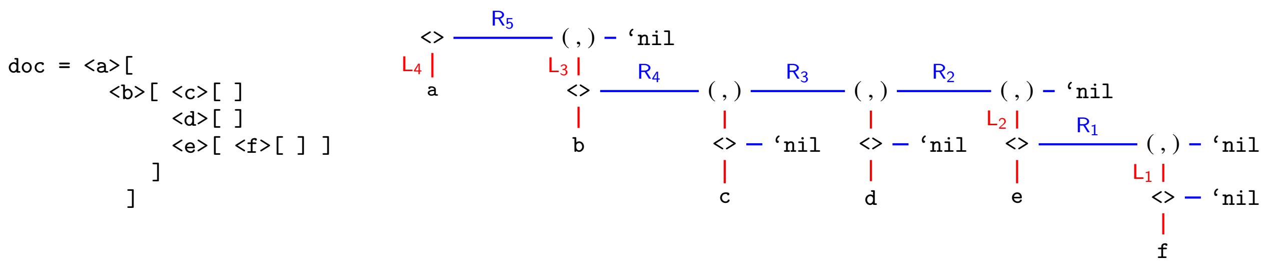
where *axis* produces the axes defined in the previous section, l ranges over XML element names (*i.e.*, atoms), and $*$ is a wildcard test that denotes any label.

The semantics of XPath (as defined in [22]) relies on sets of nodes. Informally, given an initial set of nodes N_1 and a sequence of steps $s_1/\ldots/s_n$, the result of the evaluation of an XPath query is the set N_{n+1} obtained by applying the composition $s_n \circ \ldots \circ s_1$ to N_1. An application of a step $a :: l[p]$ to a set of nodes is computed as follows. First, for each node n in N, we compute the set N_a of nodes reachable through the axis a from n. Next, we filter N_a to keep only the set N_l of nodes whose label is l. Then, we keep only the set N_p of nodes of N_l for which the predicate p evaluates to true. Lastly, we *remove any duplicate* from N_p and return the nodes in document order. The truth value of predicates with respect to a node n is inductively defined as:

$$
n \Vdash \mathit{path} : (\mathit{path}(\{n\}) \neq \varnothing) \qquad \frac{n \Vdash p : b}{n \Vdash \mathtt{not}(p) : \neg b}
$$

$$
\frac{n \Vdash p_1 : b_1 \quad n \Vdash p_2 : b_2}{n \Vdash p_1\ \mathtt{or}\ p_1 : b_1 \vee b_2} \qquad \frac{n \Vdash p_1 : b_1 \quad n \Vdash p_2 : b_2}{n \Vdash p_1\ \mathtt{and}\ p_1 : b_1 \wedge b_2}
$$

Recalling the example given in the introduction (translated to the more concise syntax given above), the path



```
doc = <a>[
        <b>[ <c>[ ]
             <d>[ ]
             <e>[ <f>[ ] ]
           ]
       ]
```

Figure 5. A binary tree representation of an XML document `doc = <a>[ <b>[ <c>[ ] <d>[ ] <e>[ <f>[ ] ] ] ]`

$$\mathsf{desc} :: \mathtt{a}[\mathtt{not}(\mathsf{anc} :: \mathtt{b})]$$

returns all descendants of the input that are labelled a and do *not* have an ancestor labelled b.

The biggest challenge in implementing the XPath semantics into patterns is to stick to the set-based semantics, without introducing internal node identifiers (that could be used to remove duplicates and sort the results in document order).[2] This precludes giving a compositional, step-by-step semantics of XPath using the surface language. Indeed, consider the document:

$$d \equiv \texttt{<a>}[\,\texttt{<a>}[\,\texttt{<a>}[\;]\,]\,]$$

Applying $\mathsf{desc\text{-}or\text{-}self} :: \mathtt{a}$ to d yields three intermediate results:

$$N = \{\texttt{<a>}[\texttt{<a>}[\texttt{<a>}[\,]]],\quad \texttt{<a>}[\texttt{<a>}[\,]],\quad \texttt{<a>}[\,]\}$$

but applying $\mathsf{desc\text{-}or\text{-}self} :: \mathtt{a}$ again to N yields N itself, since the semantics is set-based.

The solution we propose is based on two key observations:

i. Given a predicate $[pred]$, we can write a ℂDuce type t such that for all value v, $v \Vdash pred : \mathsf{true}$ if and only if $\vdash v : t$ (v has type t in ℂDuce);

ii. Any path $p \equiv s_1/\ldots/s_n$ can be put in the form of a predicate $[pred]$, such that the set of nodes selected by p is exactly the set of nodes for which $[pred]$ holds.

Using *(ii.)* we can put an XPath query in the form of a predicate and using *(i.)* we translate the predicate into a type t, and therefore express the whole XPath query as a pattern

$$\mathsf{desc\text{-}or\text{-}self}\{x \mid t\}.$$

When this pattern is matched against a value v, it selects all the subtrees of v for which $[pred]$ holds, that is, all the subtrees returned by $p(\{v\})$.

Giving the full translation of XPath predicates into types is out of scope and we omit it here since it is well-known result (see [4, 14] and Appendix F for more details). Moreover, regular types such as those we use are equivalent to tree automata, which are known to be strictly more expressive than XPath (*i.e.*, for every XPath query, one can give a tree automaton that recognizes exactly all the documents for which the XPath query is satisfiable). This translation into tree automata may nevertheless be problematic. We do not give a detailed account of such translations as they can be found in the literature (see Section 7). We nevertheless illustrate the encoding of XPath predicates into types with the example below:

Example 5.2. The XPath predicate

$$p \equiv \mathsf{parent} :: \mathtt{a}\ \ \mathtt{or}\ \ \mathsf{desc\text{-}or\text{-}self} :: \mathtt{b}/\mathsf{child} :: \mathtt{c}$$

is equivalent to the ℂDuce type t where:

$$\begin{array}{lcl} t & \equiv & t_1 \vee t_2 \\ t_1 & \equiv & (\mathbb{1})_{\mathsf{L}\_\cdot\mu X.(\mathsf{R}\,(\texttt{<a>}\mathbb{1})\cdot\mathsf{L}\_\cdot\top\ \mid\ \mathsf{R}\_\cdot X)} \\ t_2 & \equiv & \mu X.(\texttt{<b>}[\,\mathbb{1}{*}\ (\texttt{<c>}\mathbb{1})\ \mathbb{1}{*}\,] \vee \texttt{<}\mathbb{1}_{\mathsf{basic}}\texttt{>}[\,\mathbb{1}{*}\ X\ \mathbb{1}{*}\,]) \end{array}$$

Here, we see that the XPath operator `or` is translated into its set-theoretic counterpart (likewise for `and` and `not`). Upward path expressions are translated into zipper types, while downward path expressions are translated into recursive XML types. Furthermore, the chaining of steps is achieved by nesting the type obtained for the second step into the type obtained for the first step (see how $\texttt{<c>}\mathbb{1}$ is embedded in t_2).

The last part of our XPath to pattern translation is the rewriting of XPath queries of the form $s_1/\ldots/s_n$ into an XPath predicate. This transformation is also known (see Section 7 for references) and we formulate it as a lemma:

Lemma 5.3 (XPath to predicate translation [14]). *Let*

$$p \equiv a_1 :: l_1[p_1]/\ldots/a_n :: l_n[p_n]$$

be an XPath query. It can be translated into the following predicate:

$$p_n\ \mathsf{and}\ \mathsf{self} :: l_n/a_n^{-1} :: l_{n-1}[p_{n-1}]/\ldots/a_1^{-1} :: {*}[\mathtt{isroot}]$$

where a^{-1} is the inverse axis of a, defined as:

$$\begin{array}{lll} \mathsf{self}^{-1} = \mathsf{self} & \mathsf{child}^{-1} = \mathsf{parent} & \mathsf{parent}^{-1} = \mathsf{child} \\ \mathsf{desc}^{-1} = \mathsf{anc} & \mathsf{anc}^{-1} = \mathsf{desc} & \mathsf{foll\text{-}sibling}^{-1} = \mathsf{prec\text{-}sibling} \\ \mathsf{desc\text{-}or\text{-}self}^{-1} = \mathsf{anc\text{-}or\text{-}self} & \mathsf{anc\text{-}or\text{-}self}^{-1} = \mathsf{desc\text{-}or\text{-}self} & \\ \mathsf{prec\text{-}sibling}^{-1} = \mathsf{foll\text{-}sibling} & & \end{array}$$

and $\mathtt{isroot} \equiv \mathtt{not}(\mathsf{parent} :: *)$.

The key point of the translation is the fact that, for instance, the nodes selected by an XPath query $\mathsf{child} :: \mathtt{a}/\mathsf{desc} :: \mathtt{b}$ are all the nodes labelled b that have an *ancestor* a, the parent of which is the root of the document. In other words, the nodes selected by "$\mathsf{child} :: \mathtt{a}/\mathsf{desc} :: \mathtt{b}$" are all the nodes for which the predicate "$[\mathsf{self} :: \mathtt{b}/\mathsf{anc} :: \mathtt{a}/\mathsf{child} :: *[\mathtt{isroot}]]$" holds.

To conclude the encoding of XPath, we remark that since our pattern traverses an XML tree using a depth-first recursion when going downward and accumulates in reverse when going upward, the values returned by our translation of XPath queries are *(i)* in document order and *(ii)* without duplicates. This allows us to add to our surface language one last extension to the syntax of expressions: "$e/path$" which is translated into:

$$\mathsf{match}\ e\ \mathsf{with}\ \mathsf{desc}\{x \mid t_{path}\} \rightarrow x \ \mid\ \_ \rightarrow [\,]$$

where t_{path} is the type translation of the XPath query *path*. With this extension, the ℂDuce version of the "*get_links*" function given in the introduction becomes as compact as in XQuery:

```
let get_links : <_>_ → (<a>_ → <a>_) → [ <a>_ * ] =
fun page -> fun print ->
 transform page/descendant::a[not(ancestor::b)] with
        x -> [ (print x) ]
```

5.2 XQuery

This section shows how our surface language can be used to encode a relevant fragment of XQuery 3.0. The fragment we consider is an *extension* of $\mathsf{XQ_H}$ defined by Benedikt and Vu in [2].

Definition 5.4 ($\mathsf{XQ_H^+}$). An $\mathsf{XQ_H^+}$ *query* is a finite term produced by the following grammar:

[2] We could introduce a "set" type constructor exploiting internal node identifiers and new patterns for sets (besides sequences). However, this is merely the same as adding a separate layer for XPath on top of ℂDuce. Rather, we chose to encode XPath into ℂDuce patterns and thus benefit from the already existing static type system and efficient execution model of ℂDuce.

$$\begin{array}{lll}
query ::= & \texttt{()} & \text{(empty sequence)}\\
\quad | & c & \text{(constant)}\\
\quad | & \texttt{<}l\texttt{>}query\texttt{</}l\texttt{>} & \text{(XML fragment)}\\
\quad | & query, query & \text{(sequence operator)}\\
\quad | & x & \text{(variable)}\\
\quad | & x/path & \text{(XPath)}\\
\quad | & \mathsf{for}\ x\ \mathsf{in}\ query\ \mathsf{return}\ query & \text{(sequence iteration)}\\
\quad | & query(query, \ldots, query) & \text{(application)}\\
\quad | & \mathsf{fun}\ x_1 : t_1, \ldots, x_n : t_n\ \mathsf{as}\ t.\ query & \textbf{(abstraction)}\\
\quad | & \mathsf{switch}\ query & \textbf{(value switch)}\\
 & \quad \mathsf{case}\ c\ \mathsf{return}\ query & \\
 & \quad \mathsf{default}\ \mathsf{return}\ query & \\
\quad | & \mathsf{typeswitch}\ query & \textbf{(type switch)}\\
 & \quad \mathsf{case}\ t\ \mathsf{as}\ x\ \mathsf{return}\ query & \\
 & \quad \mathsf{default}\ \mathsf{return}\ query &
\end{array}$$

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

To the best of our knowledge, [2] is the first work to propose a "Core" fragment of XQuery 3.0 which abstracts away most of the idiosyncrasies of the actual specification while retaining essential features (path navigation for instance). Definition 5.4 differs from $\mathsf{XQ_H}$ by the last three productions (with bold names) since $\mathsf{XQ}_{\mathsf{H}}^{+}$ extends $\mathsf{XQ_H}$ with type and switch cases (described informally in the introduction) and with *type annotations* on functions (which are only optional in the standard). Our claim is the following. 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 provides *(i)* a formal semantics and a typechecking algorithm for XQuery and *(ii)* the soundness property that the original XQuery programs do not yield a runtime error. 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.

Note that in XQuery, all values are sequences. Therefore, the constant "42" is considered as the *singleton sequence* that contains the element "42". A consequence is that there are only "flat" sequences in XQuery; the only way to create nested data structures is to use XML constructs. The only difficulty with respect to our translation is that we need to embed/extract values explicitly into/from sequences. It also needs to disambiguate types: an XQuery function that takes an integer as argument can be applied to a sequence containing only one integer. The translation is defined in Figure 6, by a function $[\![\_]\!]_{\mathsf{X}\mathbb{C}}$ that converts an XQuery query into a ℂDuce expression.

The translation is straightforward and ensures that the result of a translation $[\![q]\!]_{\mathsf{X}\mathbb{C}}$ 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``, constants are translated into singleton sequences containing that constant and similarly for XML fragments. The sequence operator is translated into a concatenation. Variables do not require any special treatment. XPath queries and "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 fashion 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. Abstractions are

$$\begin{array}{ll}
[\![\texttt{()}]\!]_{\mathsf{X}\mathbb{C}} & = \texttt{`nil}\\
[\![c]\!]_{\mathsf{X}\mathbb{C}} & = [\,c\,]\\
[\![\texttt{<}l\texttt{>}q\texttt{</}l\texttt{>}]\!]_{\mathsf{X}\mathbb{C}} & = [\,\texttt{<}l\texttt{>}[\![q]\!]_{\mathsf{X}\mathbb{C}}\,]\\
[\![q_1,\ q_2]\!]_{\mathsf{X}\mathbb{C}} & = [\![q_1]\!]_{\mathsf{X}\mathbb{C}}\ \texttt{@}\ [\![q_2]\!]_{\mathsf{X}\mathbb{C}}\\
[\![x]\!]_{\mathsf{X}\mathbb{C}} & = x\\
[\![x/path]\!]_{\mathsf{X}\mathbb{C}} & = \begin{array}{l}\mathsf{transform}\ x\ \mathsf{with}\\ \quad i \rightarrow i/path\\ \mid \_ \rightarrow \texttt{`nil}\end{array}\\
[\![\mathsf{for}\ x\ \mathsf{in}\ q_1\ \mathsf{return}\ q_2]\!]_{\mathsf{X}\mathbb{C}} & = \begin{array}{l}\mathsf{transform}\ [\![q_1]\!]_{\mathsf{X}\mathbb{C}}\ \mathsf{with}\\ \quad x \rightarrow [\![q_2]\!]_{\mathsf{X}\mathbb{C}}\\ \mid \_ \rightarrow \texttt{`nil}\end{array}\\
\left[\!\!\left[\begin{array}{l}\mathsf{switch}\ q_1\\ \quad \mathsf{case}\ c\ \mathsf{return}\ q_2\\ \quad \mathsf{default\ return}\ q_3\end{array}\right]\!\!\right]_{\mathsf{X}\mathbb{C}} & = \begin{array}{l}\mathsf{match}\ [\![q_1]\!]_{\mathsf{X}\mathbb{C}}\ \mathsf{with}\\ \quad [\,c\,] \rightarrow [\![q_2]\!]_{\mathsf{X}\mathbb{C}}\\ \mid \ \_ \ \rightarrow [\![q_3]\!]_{\mathsf{X}\mathbb{C}}\end{array}\\
\left[\!\!\left[\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]\!\!\right]_{\mathsf{X}\mathbb{C}} & = \begin{array}{l}\mathsf{match}\ [\![q_1]\!]_{\mathsf{X}\mathbb{C}}\ \mathsf{with}\\ \quad x\ \&\ \mathsf{seq}(t) \rightarrow [\![q_2]\!]_{\mathsf{X}\mathbb{C}}\\ \mid \_ \rightarrow [\![q_3]\!]_{\mathsf{X}\mathbb{C}}\end{array}\\
\multicolumn{2}{l}{[\![\mathsf{fun}\ x_1 : t_1, \ldots, x_n : t_n\ \mathsf{as}\ t.\ q]\!]_{\mathsf{X}\mathbb{C}} =}\\
 & \mu\_^{\mathsf{seq}(t_1)\rightarrow\ldots\rightarrow\mathsf{seq}(t)}(x_1).\\
 & \quad \ddots\\
 & \quad\quad \mu\_^{\mathsf{seq}(t_n)\rightarrow\mathsf{seq}(t)}(x_n).[\![q]\!]_{\mathsf{X}\mathbb{C}}\\
q(q_1, \ldots, q_n) & = [\![q]\!]_{\mathsf{X}\mathbb{C}}\ \ [\![q_1]\!]_{\mathsf{X}\mathbb{C}} \cdots [\![q_n]\!]_{\mathsf{X}\mathbb{C}}
\end{array}$$

where $\mathsf{seq}(t) \equiv (t \wedge [\,\mathbb{1}*]) \vee ([\,t \setminus [\,\mathbb{1}*]\,])$

Figure 6. Translation of $\mathsf{XQ}_{\mathsf{H}}^{+}$ into ℂDuce.

translated into a currified ℂDuce function, and the same treatment of "sequencing" the type is applied to the type of the arguments and type of the result. Lastly, application is translated as-is.

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

6. Extensions

Bridging the gap between ℂDuce and XQuery is beneficial to both languages. While ℂDuce gains the declarativity of XPath queries, XQuery could be improved in the following way. First, supporting overloaded function in XQuery would now be possible. Note that in the absence of parametric polymorphism, ad-hoc polymorphism is a way to avoid code duplication while remaining in a typed setting. More interestingly, the fact that ℂDuce types can now express constraints on *path* (in particular, upward ones) shows us that such constraints could also be added to XQuery. While we do not expect the average XQuery programmer to write complex recursive types on zippers, a syntax of types using XPath could easily be added. This would allow one to define a function:

```
declare function f($x as parent::a)
                       as self::d[child::b]
```

which accepts as argument any XML element whose parent is labelled `a` and returns elements whose label is `d` and that have a child labelled `b`.

Both our calculus and XQuery require functions to be annotated with types. This is particularly troublesome when we use XPath axis expressions such as parent inside the function body as follows:

```
let f : ??? = fun x -> parent(x)
```

The simplest way to type `f` is to use the imprecise type $(\texttt{<}\mathbb{1}\texttt{>}\mathbb{1})_{\top} \rightarrow [\,(\texttt{<}\mathbb{1}\texttt{>}\mathbb{1})_{\top}?\,]$ (*i.e.*, accept any XML element and return a list of at most one XML element) which is indeed the approach taken by XQuery. However, such a type is barely interesting since the type information about the result is useless. The syntax of XPath in types introduced just above is not of any help for this problem.

For non-recursive first-order functions, adopting C++ template style polymorphism allows more flexibility by typing not the function definition but each application individually. We already posses all the machinery needed to implement this style of polymorphism since we can use mappings of our patterns as C++ templates. Instead of specifying for functions a set of arrows from types to types, we specify a set of *templates*, that is, arrows from *patterns* to *patterns*. The idea is to use pattern capture variables as poor man's polymorphic type variables.

For instance, we can specify for the function `f` above the "template" $((\_)_{\mathsf{L}\_\cdot\mu X.(\mathsf{R}\,x\cdot(\mathsf{L}\_\cdot\top|\bullet)\ |\ \mathsf{R}\_\cdot X)}) \to [\,x?\,]$. With the syntactic sugar of Figure 4, it corresponds to the following definition:

`let f : parent{`$x \mid \_$`}` $\to$ `[`x`?] = fun x -> parent(x)`

The definitions of template functions are lightly typechecked: essentially, if the template $p_1 \to p_2$ is specified, the system checks that the template function has type $\wr p_1 \int \to \wr p_2 \int$ (for `f` this roughly corresponds to check that it has the imprecise type defined at the beginning of the section). The applications of template functions instead are precisely typed by retypechecking the function from scratch: if a function with template $p_1 \to p_2$ is applied to an argument of type t, then p_1 is matched against t (via $\_/\_$ and $\_/\!\!/\_$) and the corresponding substitutions are applied to p_2 yielding the result type; finally, the whole function definition is typechecked against the deduced types. Therefore, when `f` is applied to an element whose parent has type t, the system correctly deduces the type $[\,t?\,]$ for the application. Formally, we add to expressions top-level declarations of template functions of the form $\lambda F^{(p\to p;\ldots;p\to p)}(x).e$ as well as their applications Fe. We check well-formedness of declarations (lightweight typechecking) and type each application by applying "template instantiation", which matches the input of the template with the type of the argument and instantiates the output of the template by the resulting substitutions. Formal definitions are given in Appendix D and soundness is ensured by construction.

7. Related Work and Conclusion

Our work tackles several aspects of XML programming, the more salient being: *(i)* encoding of XPath or XPath-like expressions in 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 differences with the present 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 addition of accumulators to pattern matching is reminiscent of Macro Tree Transducers (MTTs, [7]), 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 ([10, 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 ([13, 14] where zippers to manage XPath backward 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 ([25]) gives a polynomial time typechecking algorithm for XQuery (in the absence of nested "`let`" or "`for`" constructs), it remains too much imprecise (in particular, backward axes are left untyped). Recent work by Genevès *et al.* [15] introduces XQTC, a typechecker for XQuery based again on μ-calculus satisfiability but using the backward type inference approach. While it certainly is a step in the right direction, it only considers XQuery 1.0 queries and, in particular, does not feature arrow types yet. Interestingly, while Genevès *et al.* use zippers to represent XML trees in [14], in [15] they do not.

Future work includes extensions to other XQuery constructs as well as XMLSchema, the extension of ℂDuce compiler to implement XPath expressions, the addition of aggregate functions by associating accumulators to specific operators, and the definition and study of polymorphic systems for both XQuery and ℂDuce+XPath.

References

[1] R. M. Amadio and L. Cardelli. Subtyping recursive types. *ACM Trans. Program. Lang. Syst.*, 15(4):575–631, 1993.

[2] M. Benedikt and H. Vu. Higher-order functions and structured datatypes. In *WebDB*, 2012.

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

[4] D. Calvanese, G. D. Giacomo, M. Lenzerini, and M. Y. Vardi. Node selection query languages for trees. In *AAAI*, 2010.

[5] G. Castagna and K. Nguyễn. Typed iterators for XML. In *ICFP*, 2008.

[6] G. Castagna and Z. Xu. Set-theoretic foundation of parametric polymorphism and subtyping. In *ICFP*, 2011.

[7] J. Engelfriet and H. Vogler. Macro tree transducers. *J. Comput. Syst. Sci.*, 31(1):71–146, 1985.

[8] A. Frisch. *Théorie, conception et réalisation d'un langage adapté à XML.* PhD thesis, Université Paris 7 Denis Diderot, 2004.

[9] 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.

[10] A. Frisch and H. Hosoya. Towards practical typechecking for macro tree transducers. In *DBPL*, 2007.

[11] V. Gapeyev, F. Garillot, and B. C. Pierce. Statically typed document transformation: An Xtatic experience. In *PLAN-X*, 2006.

[12] V. Gapeyev and B. C. Pierce. Paths into patterns. Technical Report MS-CIS-04-25, University of Pennsylvania, Oct. 2004.

[13] P. Genevès and N. Layaïda. Eliminating dead-code from XQuery programs. In *ICSE*, 2010.

[14] P. Genevès, N. Layaïda, and A. Schmitt. Efficient static analysis of XML paths and types. In *PLDI*, 2007.

[15] P. Genevès, N. Layaïda, and C. Vanoirbeek. XQTC: A static type-checker for XQuery using backward type inference. Research Report RR-8149, INRIA, Nov. 2012.

[16] H. Hosoya and B. C. Pierce. Regular expression pattern matching for XML. *J. Funct. Program.*, 13(6):961–1004, 2003.

[17] H. Hosoya and B. C. Pierce. XDuce: A statically typed XML processing language. *ACM Trans. Internet Technol.*, 3(2):117–148, 2003.

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

[19] R. Lämmel. Scrap your boilerplate with XPath-like combinators. In *POPL*, 2007.

[20] S. Maneth, A. Berlea, T. Perst, and H. Seidl. XML type checking with macro tree transducers. In *PODS*, 2005.

[21] K. Nguyễn. *Combinator language for XML: design, typing, and implementation*. PhD thesis, Université Paris-Sud 11, 2008.

[22] W3C: XPath 1.0. `http://www.w3.org/TR/xpath`, 1999.

[23] W3C: XPath 2.0. `http://www.w3.org/TR/xpath20`, 2010.

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

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

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

[27] W3C: XML Schema. `http://www.w3.org/XML/Schema`, 2009.

[28] Z. Xu. *Parametric Polymorphism for XML Processing Languages*. PhD thesis, Université Paris 7 Denis Diderot, May 2013.

Appendix

A. Actual ℂDuce Code

The actual definitions in ℂDuce for the functions given in Figure 1 are as follows:

```
let get_links (page: AnyXml)(print: <a>_ -> <a>_) : [ <a>_ * ] =
  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>_)
    <a class="style1" href=h> l -> <a href=h>[ <b>l ]
  | x -> x
```

B. Subtyping

In order to formally extend the subtyping relation defined in [9] to the types of Section 2.1, it suffices to modify DEFINITION 4.3 of [9] as done by Definition B.2 (the reader can refer to [9] for the complete definitions of the notations used there). However, before extending the subtyping relation, we need some auxiliary definitions to give an interpretation of zipper types that is compatible with their infinite nature.

Let $\mathcal{T}^Z$ denote the set of zipper types. First of all, notice that the contractivity condition on zipper types implies that the binary relation $\triangleright \subseteq \mathcal{T}^Z \times \mathcal{T}^Z$ defined by $\tau_1 \vee \tau_2 \triangleright \tau_i$ and $\neg\tau \triangleright \tau$ is Noetherian (that is, strongly normalizing). This gives an induction principle on $\mathcal{T}^Z$ that we use below.

Let D be a set, $[\![\_]\!] : \mathcal{T} \to \mathcal{P}(D)$ be a set-theoretic interpretation (as defined in Definition 4.1 in [9], where $\mathcal{T}$ denotes the set of all types and $\mathcal{P}(D)$ is the powerset of D). Let $\mathcal{Z}$ denote $\{\mathsf{L}d \mid d \in D\} \cup \{\mathsf{R}d \mid d \in D\}$. We use $\mathcal{Z}^*$ to denote the free monoid on $\mathcal{Z}$.

We define a binary predicate $(s \in_{[\![]\!]} \tau)$ parametric in the set-theoretic interpretation $[\![]\!]$ where $s \in \mathcal{Z}^*$ and $\tau \in \mathcal{T}^Z$. The truth value of $(s \in_{[\![]\!]} \tau)$ is defined by induction on the pair (s, τ) ordered lexicographically, using the inductive structure for elements of $\mathcal{Z}^*$ (these are *finite* sequences s of decorated elements of D) and the induction principle we mentioned above for zipper types. Here is the definition:

$$
\begin{array}{rcll}
s \in_{[\![]\!]} \top & = & \texttt{true} & \\
\epsilon \in_{[\![]\!]} \bullet & = & \texttt{true} & \\
\mathsf{R}\, d \cdot s \in_{[\![]\!]} \mathsf{R}\,(u)_\tau \cdot \tau & = & (s \in_{[\![]\!]} \tau)\ \texttt{and}\ (d \in [\![u]\!]) & \\
\mathsf{L}\, d \cdot s \in_{[\![]\!]} \mathsf{L}\,(u)_\tau \cdot \tau & = & (s \in_{[\![]\!]} \tau)\ \texttt{and}\ (d \in [\![u]\!]) & \\
s \in_{[\![]\!]} \tau_1 \vee \tau_2 & = & (s \in_{[\![]\!]} \tau_1)\ \texttt{or}\ (s \in_{[\![]\!]} \tau_2) & \\
s \in_{[\![]\!]} \neg\tau & = & \texttt{not}(s \in_{[\![]\!]} \tau) & \\
s \in_{[\![]\!]} \tau & = & \texttt{false} & \text{otherwise}
\end{array}
$$

This predicate is then used to define the following interpretation of zipper types:

Definition B.1 (Zipper type interpretation). Let D be a set, $[\![\_]\!] : \mathcal{T} \to \mathcal{P}(D)$ be a set-theoretic interpretation, and $\mathcal{Z}$ denote $\{\mathsf{L}d \mid d \in D\} \cup \{\mathsf{R}d \mid d \in D\}$. The interpretation $[\![\tau]\!]$ of a zipper type τ with respect to $[\![\_]\!]$ is defined as:

$$[\![\tau]\!] = \{s \in \mathcal{Z}^* \mid s \in_{[\![]\!]} \tau\}$$

Subtyping for zipper types is defined as $\tau <: \tau' \overset{\text{def}}{=} [\![\tau]\!] \subseteq [\![\tau']\!]$.

Finally, the interpretation of zipper types is used to extend DEFINITION 4.3 of [9] to our new types, as follows.

Definition B.2 (Extensional interpretation). Let $[\![\_]\!] : \mathcal{T} \to \mathcal{P}(D)$ be a set-theoretic interpretation in some set D. Let $\mathcal{Z} \overset{\text{def}}{=} \{\mathsf{L}d \mid d \in D\} \cup \{\mathsf{R}d \mid d \in D\}$, and $\mathcal{Z}^*$ denote the free monoid on $\mathcal{Z}$.

We define the associated *extensional interpretation* as the unique set-theoretic interpretation

$$\mathbb{E}(\_) : \mathcal{T} \to \mathcal{P}(\mathbb{E}D)$$

(where $\mathbb{E}D = \mathcal{C} + D^2 + \mathcal{P}(D \times D_\Omega) + (\mathcal{P}(D) \times \mathcal{Z}^*)$) such that:

$$
\begin{array}{lcll}
\mathbb{E}(b) & = & \mathbb{B}[\![b]\!] & \subseteq \mathcal{C} \\
\mathbb{E}(t_1 \times t_2) & = & [\![t_1]\!] \times [\![t_2]\!] & \subseteq D^2 \\
\mathbb{E}(t_1 \to t_2) & = & [\![t_1]\!] \to [\![t_2]\!] & \subseteq \mathcal{P}(D \times D_\Omega) \\
\mathbb{E}((u)_\tau) & = & [\![u]\!] \times [\![\tau]\!] & \subseteq \mathcal{P}(D) \times \mathcal{Z}^*
\end{array}
$$

All the other definitions of [9] remain unchanged, in particular, those of a well-founded model and of its induced subtyping relation.

In order to decide the subtyping relation induced by a model, a possibility is to extend the definitions of Section 6 in [9] to account for the new zipper type constructor. A simpler way is to use a well-founded model, encode both zipper types and zipper values in the values and types of [9] (our pre-type and pre-values) via the encoding function $\mathsf{Enc}(\_)$ below, and prove that $[\![\tau_1]\!] \subseteq [\![\tau_2]\!]$ if and only if $\mathsf{Enc}(\tau_1) \leq \mathsf{Enc}(\tau_2)$.

Definition B.3 (Encoding of zippers). Zippers and zipper types are encoded (inductively) into pairs and (coinductively) into product types, respectively, as follows:

$$\begin{array}{rcl}
\mathsf{Enc}(\mathsf{L}\,(w)_\delta \cdot \delta) & \equiv & ((\text{‘L}, w), \mathsf{Enc}(\delta)) \\
\mathsf{Enc}(\mathsf{R}\,(w)_\delta \cdot \delta) & \equiv & ((\text{‘R}, w), \mathsf{Enc}(\delta)) \\
\mathsf{Enc}(\bullet) & \equiv & \text{‘}\mathtt{nil} \\
\\
\mathsf{Enc}(\mathsf{L}\,(u)_\tau \cdot \tau) & \equiv & (\text{‘L} \times u) \times \mathsf{Enc}(\tau) \\
\mathsf{Enc}(\mathsf{R}\,(u)_\tau \cdot \tau) & \equiv & (\text{‘R} \times u) \times \mathsf{Enc}(\tau) \\
\mathsf{Enc}(\top) & \equiv & \mu X.((\text{‘L} \vee \text{‘R}) \times \mathbb{1}) \times X \vee \text{‘}\mathtt{nil} \\
\mathsf{Enc}(\bullet) & \equiv & \text{‘}\mathtt{nil} \\
\mathsf{Enc}(\neg\tau) & \equiv & \mathsf{Enc}(\top) \setminus \mathsf{Enc}(\tau) \\
\mathsf{Enc}(\tau_1 \vee \tau_2) & \equiv & \mathsf{Enc}(\tau_1) \vee \mathsf{Enc}(\tau_2)
\end{array}$$

The termination of the subtyping algorithm in [9] implies the termination of the subtyping algorithm on (the encoding of) the extended type algebra. Since the encoding is linear on the size of terms, both algorithms have the same complexity.

As an aside, note that the whole calculus presented in this paper can be faithfully encoded in $\mathbb{C}$Duce without affecting the complexity of the algorithms: it is straightforward to extend the encodings of Definition B.3 to $\mathcal{E}$ and $\mathcal{T}$.

All it remains to prove is the soundness and completeness of the encoding, namely:

Theorem B.4. *Let $\leq$ be a subtyping relation for the calculus in [9] induced by a well-founded model. Then, there exist a set D and a set theoretic interpretation $[\![\_]\!] : \mathcal{T} \to \mathcal{P}(D)$ such that for all $\tau_1, \tau_2 \in \mathcal{T}^Z$, the following holds:*

$$[\![\tau_1]\!] \subseteq [\![\tau_2]\!] \iff \mathit{Enc}(\tau_1) \leq \mathit{Enc}(\tau_2)$$

Proof. Let us use λ_{FCB} to denote the λ-calculus defined in [9]. Let $\leq$ be any subtyping relation for λ_{FCB} induced by a well-founded model. Theorem 5.5 in [9] states that $\mathsf{Enc}(\tau_1) \leq \mathsf{Enc}(\tau_2)$ if and only if $[\![\mathsf{Enc}(\tau_1)]\!]_\mathcal{V} \subseteq [\![\mathsf{Enc}(\tau_2)]\!]_\mathcal{V}$, where $[\![\_]\!]_\mathcal{V}$ is the *value interpretation* for the types of λ_{FCB} defined as $[\![t]\!]_\mathcal{V} \overset{\text{def}}{=} \{v \mid \vdash v : t\}$ (where v and t respectively range over the values and types of λ_{FCB}).

The simplest way to prove this theorem, then, is to produce a set D and interpretation $[\![\_]\!] : \mathcal{T} \to \mathcal{P}(D)$ such that there is a one-to-one correspondence between $[\![\tau]\!]$ and $[\![\mathsf{Enc}(\tau)]\!]_\mathcal{V}$.

To that end, take as D the set of all values of λ_{FCB}, that is, $[\![\mathbb{1}]\!]_\mathcal{V}$, and as interpretation any interpretation that on the pre-values of our calculus (which are the values of λ_{FCB}) behaves as $[\![\_]\!]_\mathcal{V}$, that is, $[\![w]\!] = [\![w]\!]_\mathcal{V}$ for every pre-value w. Next, we define an embedding function $f : \mathcal{Z}^* \hookrightarrow [\![\mathsf{Enc}(\top)]\!]_\mathcal{V}$ from the resulting $\mathcal{Z}^*$ to set of values of type $\mathsf{Enc}(\top)$, by induction on the length of the elements of $\mathcal{Z}^*$ as follows:

$$\begin{array}{rcl}
f(\varepsilon) & = & \text{‘}\mathtt{nil} \\
f(\mathsf{L}\,w \cdot s) & = & ((\text{‘L}, w), f(s)) \\
f(\mathsf{R}\,w \cdot s) & = & ((\text{‘R}, w), f(s))
\end{array}$$

We can prove that f is injective by induction on $\mathcal{Z}^*$, and surjective by induction on $[\![\mathsf{Enc}(\top)]\!]_\mathcal{V}$ (recall that, contrary to types, values are inductively defined).

Since $s \in [\![\tau]\!] \iff s \in_{[\![]\!]} \tau$, then to prove that f is a one-to-one mapping from $[\![\tau]\!]$ to $[\![\mathsf{Enc}(\tau)]\!]_\mathcal{V}$, it suffices to prove that for all $s \in \mathcal{Z}^*$ and $\tau \in \mathcal{T}^Z$, $s \in_{[\![]\!]} \tau \iff f(s) \in [\![\mathsf{Enc}(\tau)]\!]_\mathcal{V}$. This can be easily proved by induction on the pair (s, τ) ordered lexicographically, by performing a case analysis on the definition of $\in_{[\![]\!]}$. $\square$

C. Typing

Zipper typing rules $\boxed{\vdash \delta : \tau}$

$$\text{[ZT-Root]}\ \frac{}{\vdash \bullet : \bullet} \qquad \text{[ZT-Sub]}\ \frac{\vdash \delta : \tau \qquad \tau <: \tau'}{\vdash \delta : \tau'} \qquad \text{[ZT-Left]}\ \frac{\vdash w : t \qquad \vdash \delta : \tau}{\vdash \mathsf{L}\,(w)_\delta \cdot \delta : \mathsf{L}\,(t)_\tau \cdot \tau} \qquad \text{[ZT-Right]}\ \frac{\vdash w : t \qquad \vdash \delta : \tau}{\vdash \mathsf{R}\,(w)_\delta \cdot \delta : \mathsf{R}\,(t)_\tau \cdot \tau}$$

Typing rules $\boxed{\Gamma \vdash e : \tau}$

$$\text{[T-Cst]}\ \frac{}{\Gamma \vdash c : b_c} \qquad \text{[T-Var]}\ \frac{}{\Gamma \vdash x : \Gamma(x)} \qquad \text{[T-Acc]}\ \frac{}{\Gamma \vdash \dot{x} : \Gamma(\dot{x})} \qquad \text{[T-Zip-Val]}\ \frac{\vdash w : t \qquad \vdash \delta : \tau \qquad t \leq \mathbb{1}_{\mathsf{NZ}}}{\Gamma \vdash (w)_\delta : (t)_\tau} \qquad \text{[T-Zip-Expr]}\ \frac{\vdash e : t \qquad t \leq \mathbb{1}_{\mathsf{NZ}}}{\Gamma \vdash (e)_\bullet : (t)_\bullet}$$

$$\text{[T-Pair]}\ \frac{\Gamma \vdash e_1 : t_1 \qquad \Gamma \vdash e_2 : t_2}{\Gamma \vdash (e_1, e_2) : t_1 \times t_2} \qquad \text{[T-Sub]}\ \frac{\Gamma \vdash e : s \qquad s \leq t}{\Gamma \vdash e : t} \qquad \text{[T-Op]}\ \frac{\forall i = 1..n_o,\ \Gamma \vdash e_i : t_i \qquad t_1, \ldots, t_{n_o} \xrightarrow{o} t}{\Gamma \vdash o(e_1, \ldots, e_{n_o}) : t}\ \text{for } o \in \mathcal{O}$$

$$\text{[T-Fun]}\ \frac{t = \bigwedge_{i=1..n} (t_i \to s_i) \wedge \bigwedge_{j} \neg(t'_j \to s'_j) \qquad t \not\simeq \mathbb{0} \qquad \forall i = 1..n,\ \Gamma \cup \{f \mapsto t, x \mapsto t_i\} \vdash e : s_i}{\Gamma \vdash \mu f^{(t_1 \to s_1; \ldots; t_n \to s_n)}(x).e : t}$$

$$\text{[T-Match]}\ \frac{\begin{array}{l} t \leq \llparenthesis p_1 \rrparenthesis \vee \llparenthesis p_2 \rrparenthesis \\ t_2 \equiv t \wedge \neg \llparenthesis p_1 \rrparenthesis \end{array} \qquad \begin{array}{l} t_1 \equiv t \wedge \llparenthesis p_1 \rrparenthesis \\ \Sigma_i \equiv \{\dot{x} \mapsto \mathsf{Init}(\dot{x}) \mid \dot{x} \in \mathrm{Acc}(p_i)\} \end{array} \qquad \begin{array}{l} \Gamma \vdash e : t \\ \Gamma \cup t_i / p_i \cup t_i \mathbin{/\!\!/_{\Sigma_i}} p_i \vdash e_i : t'_i \end{array}}{\Gamma \vdash \mathsf{match}\ e\ \mathsf{with}\ p_1 \to e_1 \mid p_2 \to e_2 : \bigvee_{\{i \mid t_i \not\simeq \mathbb{0}\}} t'_i}$$

Figure 7. Typing rules

$$
\begin{array}{lcll}
t \underset{\Sigma}{/\!\!/} t' & = & \Sigma & \\
t \underset{\Sigma}{/\!\!/} x & = & \Sigma & \\
t \underset{\Sigma}{/\!\!/} \dot{x} & = & \Sigma[{}^{s}\!/_{\dot{x}}] & \text{where } t, \Sigma(\dot{x}) \overset{\mathsf{Op}(\dot{x})}{\rightarrow} s \\
t \underset{\Sigma}{/\!\!/} (p_1, p_2) & = & \displaystyle\bigsqcup_{(t_1,t_2)\in\Pi(t)} t_2 \underset{(t_1 \underset{\Sigma}{/\!\!/} p_1)}{/\!\!/} p_2 & \\
t \underset{\Sigma}{/\!\!/} p_1 | p_2 & = & t \underset{\Sigma}{/\!\!/} p_1 & \text{if } t \leq \wr p_1 \int \\
t \underset{\Sigma}{/\!\!/} p_1 | p_2 & = & t \underset{\Sigma}{/\!\!/} p_2 & \text{if } t \leq \neg\wr p_1 \int \\
t \underset{\Sigma}{/\!\!/} p_1 | p_2 & = & (t \wedge \wr p_1 \int) \underset{\Sigma}{/\!\!/} p_1 \bigsqcup (t \wedge \neg\wr p_1 \int) \underset{\Sigma}{/\!\!/} p_2 & \text{otherwise} \\
t \underset{\Sigma}{/\!\!/} p_1 \,\&\, p_2 & = & t \underset{(t \underset{\Sigma}{/\!\!/} p_1)}{/\!\!/} p_2 & \\
t \underset{\Sigma}{/\!\!/} (x := c) & = & \Sigma & \\
(u)_\tau \underset{\Sigma}{/\!\!/} (q)_\varphi & = & \tau \underset{(u \underset{\Sigma}{/\!\!/} q)}{/\!\!/} \varphi & \\
\tau \underset{\Sigma}{/\!\!/} \top & = & \Sigma & \\
\mathsf{L}\,(u)_\tau \cdot \tau \underset{\Sigma}{/\!\!/} \mathsf{L}\, p \cdot \varphi & = & \tau \underset{((u)_\tau \underset{\Sigma}{/\!\!/} p)}{/\!\!/} \varphi & \\
\mathsf{R}\,(u)_\tau \cdot \tau \underset{\Sigma}{/\!\!/} \mathsf{R}\, p \cdot \varphi & = & \tau \underset{((u)_\tau \underset{\Sigma}{/\!\!/} p)}{/\!\!/} \varphi & \\
\tau \underset{\Sigma}{/\!\!/} \varphi_1 | \varphi_2 & = & \tau \underset{\Sigma}{/\!\!/} \varphi_1 & \text{if } \tau \leq \wr \varphi_1 \int \\
\tau \underset{\Sigma}{/\!\!/} \varphi_1 | \varphi_2 & = & \tau \underset{\Sigma}{/\!\!/} \varphi_2 & \text{if } \tau \leq \neg\wr \varphi_1 \int \\
\tau \underset{\Sigma}{/\!\!/} \varphi_1 | \varphi_2 & = & (\tau \wedge \wr \varphi_1 \int) \underset{\Sigma}{/\!\!/} \varphi_1 \bigsqcup (\tau \wedge \neg\wr \varphi_1 \int) \underset{\Sigma}{/\!\!/} \varphi_1 & \text{otherwise} \\
(\tau_1 \vee \tau_2) \underset{\Sigma}{/\!\!/} \varphi & = & \tau_1 \underset{\Sigma}{/\!\!/} \varphi \bigsqcup \tau_2 \underset{\Sigma}{/\!\!/} \varphi & \text{if } \varphi \not\simeq \varphi_1 | \varphi_2 \text{ and } \varphi \not\simeq \top
\end{array}
$$

$$
(\Sigma_1 \bigsqcup \Sigma_2)(\dot{x}) \;=\; \begin{cases} \Sigma_1(\dot{x}) & \text{if } \dot{x} \in \mathsf{dom}(\Sigma_1) \setminus \mathsf{dom}(\Sigma_2) \\ \Sigma_2(\dot{x}) & \text{if } \dot{x} \in \mathsf{dom}(\Sigma_2) \setminus \mathsf{dom}(\Sigma_1) \\ \Sigma_1(\dot{x}) \vee \Sigma_2(\dot{x}) & \text{if } \dot{x} \in \mathsf{dom}(\Sigma_1) \cap \mathsf{dom}(\Sigma_2) \end{cases}
$$

Figure 8. Computing the type environment for accumulators

$$
\begin{array}{lcll}
(t/x)(x) & = & t & \\
(t/(p_1,p_2))(x) & = & (\Pi_1(t)/p_1)(x) & \text{if } x \in \mathrm{Var}(p_1) \backslash \mathrm{Var}(p_2) \\
(t/(p_1,p_2))(x) & = & (\Pi_2(t)/p_2)(x) & \text{if } x \in \mathrm{Var}(p_2) \backslash \mathrm{Var}(p_1) \\
(t/(p_1,p_2))(x) & = & (\Pi_1(t)/p_1)(x) \times (\Pi_2(t)/p_2)(x) & \text{if } x \in \mathrm{Var}(p_1) \cap \mathrm{Var}(p_2) \\
(t/p_1 | p_2)(x) & = & ((t \wedge \wr p_1 \int)/p_1)(x) \vee ((t \wedge \neg\wr p_1 \int)/p_2)(x) & \\
(t/p_1 \,\&\, p_2)(x) & = & (t/p_i)(x) & \text{if } x \in \mathrm{Var}(p_i) \\
(t/(x := c))(x) & = & t_c & \text{if } t \not\simeq \mathbb{0} \\
(t/(x := c))(x) & = & \mathbb{0} & \text{if } t \simeq \mathbb{0} \\
((u)_\tau/(q)_\varphi)(x) & = & (u/q)(x) & \text{if } x \in \mathrm{Var}(p) \\
((u)_\tau/(q)_\varphi)(x) & = & (\tau/\varphi)(x) & \text{if } x \in \mathrm{Var}(\varphi) \\
 & & & \\
(\mathsf{L}\,(u)_\tau \cdot \tau/\mathsf{L}\, p \cdot \varphi)(x) & = & ((u)_\tau/p)(x) & \text{if } x \in \mathrm{Var}(p) \backslash \mathrm{Var}(\varphi) \\
(\mathsf{L}\,(u)_\tau \cdot \tau/\mathsf{L}\, p \cdot \varphi)(x) & = & (\tau/\varphi)(x) & \text{if } x \in \mathrm{Var}(\varphi) \backslash \mathrm{Var}(p) \\
(\mathsf{L}\,(u)_\tau \cdot \tau/\mathsf{L}\, p \cdot \varphi)(x) & = & ((u)_\tau/p)(x) \times (\tau/\varphi)(x) & \text{if } x \in \mathrm{Var}(p) \cap \mathrm{Var}(\varphi) \\
(\mathsf{R}\,(u)_\tau \cdot \tau/\mathsf{R}\, p \cdot \varphi)(x) & = & ((u)_\tau/p)(x) & \text{if } x \in \mathrm{Var}(p) \backslash \mathrm{Var}(\varphi) \\
(\mathsf{R}\,(u)_\tau \cdot \tau/\mathsf{R}\, p \cdot \varphi)(x) & = & (\tau/\varphi)(x) & \text{if } x \in \mathrm{Var}(\varphi) \backslash \mathrm{Var}(p) \\
(\mathsf{R}\,(u)_\tau \cdot \tau/\mathsf{R}\, p \cdot \varphi)(x) & = & ((u)_\tau/p)(x) \times (\tau/\varphi)(x) & \text{if } x \in \mathrm{Var}(p) \cap \mathrm{Var}(\varphi) \\
(\tau/\varphi_1 | \varphi_2)(x) & = & ((\tau \wedge \wr \varphi_1 \int)/\varphi_1)(x) \vee ((\tau \wedge \neg\wr \varphi_1 \int)/\varphi_2)(x) & \\
((\tau_1 \vee \tau_2)/\varphi)(x) & = & (\tau_1/\varphi)(x) \vee (\tau_2/\varphi)(x) & \text{if } \varphi \not\simeq \varphi_1 | \varphi_2
\end{array}
$$

Figure 9. Computing the type environment for capture variables

D. Template Polymorphism

First, we define *template function declarations* Δ as a list of *template functions* of the form $\lambda F^{(p\to p;\ldots;p\to p)}(x).e$:

$$\Delta \quad ::= \quad \varnothing \mid \Delta, \lambda F^{(p\to p;\ldots;p\to p)}(x).e$$

Then, we define a lightweight system that checks well-formedness of template function declarations, composed of two rules:

$$\frac{}{\vdash \cdot\ \mathsf{wf}} \qquad \frac{\begin{array}{ll} (i=1..n) & \\ & \mathrm{Var}(p'_i)\cup\mathrm{Acc}(p'_i)\subseteq\mathrm{Var}(p_i)\cup\mathrm{Acc}(p_i) \\ \vdash \Delta\ \mathsf{wf} & (x:\llparenthesis p_i \rrparenthesis)\vdash_{\varnothing} e:\llparenthesis p'_i\rrparenthesis \end{array}}{\vdash \Delta, \lambda F^{(p_1\to p'_1;\ldots;p_n\to p'_n)}(x).e\ \mathsf{wf}}$$

The first rule states that the empty list is a well-formed declaration. The second rule checks the well-formedness of a template function by performing the following checks: it checks (i) that the variables occurring in the domain of each template contain those occurring in the corresponding codomain (alternatively, we could have required equality or have substituted variables occurring only in the codomain by $\mathbb{1}$) and (ii) that the definition of the function is compatible with the accepted types of the templates.

Although a template function is checked again at each application, we may want to be slightly more stringent in checking the type of the template function definition. For instance, we may want to check that a function with template $x\ \&\ \mathsf{int} \to x$ always returns at least a value of type int. This can be done, at the expenses of simplicity, by modifying the second rule as follows:

$$\frac{\begin{array}{ll} (i=1..n) & \\ & \mathrm{Var}(p'_i)\cup\mathrm{Acc}(p'_i)\subseteq\mathrm{Var}(p_i)\cup\mathrm{Acc}(p_i) \\ \vdash \Delta\ \mathsf{wf} & (x:\llparenthesis p_i\rrparenthesis)\vdash_{\varnothing} e: p'_i[\,\llparenthesis p_i\rrparenthesis/p_i;\ \llparenthesis p_i\rrparenthesis \underset{\Sigma_i}{/\!\!/} p_i\,] \end{array}}{\vdash \Delta, \lambda F^{(p_1\to p'_1;\ldots;p_n\to p'_n)}(x).e\ \mathsf{wf}}$$

where $\Sigma_i \equiv \{\dot{x}\mapsto \mathsf{Init}(\dot{x}) | \dot{x}\in\mathrm{Acc}(p_i)\}$. The idea is that the variables in the output patterns are replaced by the best types that are possible to deduce for them in the input patterns.

Next, we extend the syntax of expressions to include applications of template functions:

$$e \quad ::= \quad \cdots \mid F\ e$$

and the operational semantics accordingly, first by extending the evaluation contexts

$$E[\,] \quad ::= \quad \cdots \mid F\ E[\,]$$

and then adding the corresponding reduction rule:

$$F\ v \rightsquigarrow e[{}^{v}\!/_{x}] \qquad \text{if } \lambda F^{(\ldots)}(x).e\in\Delta$$

All remains to do is to typecheck the new expression. First of all, notice that typing judgments are now parametric by Δ, a template function declarations: $\Gamma\vdash_{\Delta} e : t$. We already used these judgments in the second rule for well-formed declarations where we specified the empty declarations in order to forbid the definition of mutually recursive template functions. Before giving the typing rule for template function applications, we need a last definition.

To ease the notations, we denote function interfaces by a set theoretic notation (*e.g.*, $\{p_i \to p'_i\}_{i\in I}$) instead of the customary $(p_1\to p'_1;\ldots;p_n\to p'_n)$.

Definition D.1 (Template instantiation). Let $\{p_i\to p'_i\}_{i\in I}$ be a template function interface and t a type such that $t\le\bigvee_{i\in I}\llparenthesis p_i\rrparenthesis$. The *template instantiation* of $\{p_i\to p'_i\}_{i\in I}$ by t that we denote by $\{p_i\to p'_i\}_{i\in I}[t]$ is function interface defined as follows:

$$\{t\wedge\llparenthesis p_i\rrparenthesis\to t_i \mid i\in I, t\wedge\llparenthesis p_i\rrparenthesis\not\simeq\mathbb{0}\}$$

$$\begin{array}{lcl} \text{where}\quad t_i & \equiv & \llparenthesis\, p'_i[\,(t\wedge\llparenthesis p_i\rrparenthesis)/p_i;\ (t\wedge\llparenthesis p_i\rrparenthesis)\underset{\Sigma_i}{/\!\!/} p_i\,]\,\rrparenthesis \\ \Sigma_i & \equiv & \{\dot{x}\mapsto\mathsf{Init}(\dot{x}) \mid \dot{x}\in\mathrm{Acc}(p_i)\} \end{array}$$

This definition is the core of the typechecking. It deduces the type that must be used to instantiate a template function with interface $\{p_i\to p'_i\}_{i\in I}$ when it is applied to an expression of type t. To do that, it proceeds in the following four steps:

1. It check that the type t of the argument is compatible with the input types of the template function interface, that is, $t\le\bigvee_{i\in I}\llparenthesis p_i\rrparenthesis$.
2. It selects in the interface $\{p_i\to p'_i\}_{i\in I}$ only those templates whose input is compatible with the type of the arguments, that is, only those $p_i\to p'_i$ such that $t\wedge\llparenthesis p_i\rrparenthesis\not\simeq\mathbb{0}$.
3. For each of the remaining templates, it takes the part of the input that can concern the template, that is, $t\wedge\llparenthesis p_i\rrparenthesis$, and matches it against p_i (*i.e.*, $(t\wedge\llparenthesis p_i\rrparenthesis)/p_i$ and $(t\wedge\llparenthesis p_i\rrparenthesis)\underset{\Sigma_i}{/\!\!/} p_i$) so as to deduce to which type each variable in p_i must be bound.
4. It applies the substitutions found at the previous step to the output types of the corresponding template (this corresponds to the term) $p'_i[\,(t\wedge\llparenthesis p_i\rrparenthesis)/p_i;\ (t\wedge\llparenthesis p_i\rrparenthesis)\underset{\Sigma_i}{/\!\!/} p_i\,]$ to obtain the interface $\{p_i\to p'_i\}_{i\in I}[t]$ to instantiate the template function.

Once this is done, it is just matter of creating a fresh copy of the template function by using a fresh variable f (recall that template functions are not recursive) and specifying $\{p_i\to p'_i\}_{i\in I}[t]$ as interface: all it remains to do is to check whether the instance obtained is well typed and then apply the standard rule to type applications of the template function. All this is summarized by the following typing rule:

$$\frac{f \text{ fresh} \qquad \lambda F^{\{p_i \to p'_i\}_{i\in I}}(x).e' \in \Delta \qquad t \leq \bigvee_i \wr p_i \int \qquad \Gamma \vdash_\Delta e : t \qquad \Gamma \vdash_\Delta \mu f^{\{p_i \to p'_i\}_{i\in I}[t]}(x).e' : t'}{\Gamma \vdash_\Delta F\ e : \mathsf{min}\{s \mid t' \leq t \to s\}}$$

Soundness is then a direct consequence of the soundness of our system.

E. Soundness Proofs

Lemma E.1. *Let $\mathcal{O}_{\mathbb{C}Duce}$ be the set*

$$\mathcal{O}_{\mathbb{C}Duce} \equiv \{\pi_1, \pi_2, \mathit{rm}, \mathit{cons}, \mathit{snoc}\}.$$

For all type t, there exists a plinth over $\mathcal{O}_{\mathbb{C}Duce}$ that contains t.

Proof. We assume here (see [8] for the proof) that given a finite set S of types, there exists a finite set $S' \supseteq S$ that is closed under $\vee$ and $\neg$ (and therefore also closed under $\wedge$ which can be expressed in terms of union and negation).

We consider the initial set $S_0 = \{t, \mathbb{0}, \mathbb{1}\}$. Its saturation S'_0 is also finite. We now compute

$$S_1 = S'_0 \cup \bigcup_{t \in \{t \in S'_0 \mid t \leq \mathbb{1}_{\mathsf{prod}} \vee (\mathbb{1}_{\mathsf{prod}})_\top\}} \pi_1(t)$$

By construction, S_1 is finite, and so is its saturation S'_1. We similarly construct S'_2 (saturation of the closure by π_2) and S'_3 (saturation of the closure by rm). Next, consider the set $T \subseteq S'_3$ of types of the form $[\,(t_1 \vee \ldots \vee t_n)*]$ that are valid for the second argument of cons and snoc. We compute

$$S_4 = S'_3 \cup \bigcup_{[\,(t_1 \vee \ldots \vee t_n)*] \in T, U \subseteq S'_3} [\,(\bigvee_{s\in U} s \vee t_1 \vee \ldots \vee t_n)*]$$

This saturates S'_3 with respect to cons and snoc (which have the same typing function). Lastly, we compute the saturated set S'_4, which is the plinth we seek. The crucial point here is that since $[\,(t_1 \vee \ldots \vee t_n)*] \in S'_3$ which is in particular already saturated by π_1 and π_2, $t_1 \vee \ldots \vee t_n$ and `nil are already in S'_3, and so is $\bigvee_{s\in U} s$ (since $U \subseteq S'_3$). Since S'_3 is saturated w.r.t $\vee$, $\bigvee_{s\in U} s \vee t_1 \vee \ldots \vee t_n \in S'_3$ and therefore S'_4 remains saturated by π_i; and there is no need to iterate the process again. ☐

Corollary E.2. *The typing of the operator drm terminates.*

Proof. Lemma E.1 ensures that during the computation of drm, the operators π_1, π_2, and rm used in the definition of drm produce only a finite set of types. In other words, given a type t, drm needs to inspect only a finite set of types and this ensures the termination of drm. ☐

Corollary E.3. *For any type t and pattern p such that*

$$\forall \dot{x} \in \mathrm{Acc}(p), \mathit{Op}(\dot{x}) \in \mathcal{O}_{\mathbb{C}Duce}$$

the computation of $t \underset{\Sigma}{/\!\!/} p$ terminates, where

$$\Sigma \equiv \{\dot{x} \mapsto \mathit{Init}(\dot{x}) \mid \dot{x} \in \mathrm{Acc}(p)\}$$

Proof. We consider the type $t' = t \times t_0 \times \ldots \times t_n$ where $t_i = \mathsf{Init}(\dot{x_i})$. Thanks to Lemma E.1, there is a plinth containing t' and therefore the rules in Figure 8 only generate a finite set of equations, the solution of which is a set of (mutually recursive) types. ☐

Lemma E.4. *All operators $\mathcal{O}_{\mathbb{C}Duce}$ have exact input.*

Proof. We consider each operator separately. The projection operator π_i is defined for every pair and zipped pair, so its accepted input is the type $\mathbb{1} \times \mathbb{1} \vee (\mathbb{1} \times \mathbb{1})_\top$. The top-level erasure rm is defined for every value and thus its accepted input is the type $\mathbb{1}$. The operator cons does not fail for any pair of values, so its accepted input is the type $\mathbb{1} \times \mathbb{1}$. Finally, snoc never fails for the first argument and fails only if its second argument is not a sequence. Therefore, the accepted input of snoc is the type $\mathbb{1} \times [\,\mathbb{1}*]$. ☐

Lemma E.5 (Strengthening). *Let Γ_1 and Γ_2 be two typing environments such that for any $x \in \mathit{dom}(\Gamma_1)$, we have $\Gamma_2(x) \leq \Gamma_1(x)$. If $\Gamma_1 \vdash e : t$, then $\Gamma_2 \vdash e : t$.*

Proof. By induction on the derivation of $\Gamma_1 \vdash e : t$. We simply introduce an instance of the subsumption rule below each instance of the [T-VAR] rule. ☐

Lemma E.6 (Admissibility of the intersection rule). *If $\Gamma \vdash e : t_1$ and $\Gamma \vdash e : t_2$, then $\Gamma \vdash e : t_1 \wedge t_2$.*

Proof. By induction on the structure of the two typing derivations. ☐

Lemma E.7. *Let Γ be a typing environment and e an expression that is well typed under Γ. Then the set*

$$S = \{t \in \mathcal{T} \mid \Gamma \vdash e : t \ \vee\ \Gamma \vdash e : \neg t\}$$

contains $\mathbb{0}$ and closed under $\vee$ and $\neg$ (and thus $\wedge$).

Proof. By definition, S is clearly closed under $\neg$. We have $\Gamma \vdash e : \mathbb{1} \simeq \neg\mathbb{0}$ and thus $\mathbb{0} \in S$. To show that S is closed under $\vee$, consider two types t_1 and t_2 in S. If $\Gamma \nvdash e : t_1 \vee t_2$, then due to subsumption, we get $\Gamma \nvdash e : t_1$ and $\Gamma \nvdash e : t_2$. Because t_1 and t_2 are in S, we must have $\Gamma \vdash e : \neg t_1$ and $\Gamma \vdash e : \neg t_2$. By Lemma E.6, we have $\Gamma \vdash e : \neg t_1 \wedge \neg t_2$ and $\neg t_1 \wedge \neg t_2 \simeq \neg(t_1 \vee t_2)$. Therefore, either $\Gamma \vdash e : t_1 \vee t_2$ or $\Gamma \vdash e : \neg(t_1 \vee t_2)$ holds, which completes the proof. ☐

Lemma E.8 (Substitution). *Let $e, e_1, \ldots, e_n$ be expressions, $x_1, \ldots, x_n$ distinct variables, $t, t_1, \ldots, t_n$ types, and Γ a typing environment. Then:*

$$\begin{cases} \Gamma, (x_1 : t_1), \ldots, (x_n : t_n) \vdash e : t \\ \forall i = 1..n,\ \Gamma \vdash e_i : t_i \end{cases} \implies \Gamma \vdash e[{}^{e_1}/_{x_1}; \ldots; {}^{e_n}/_{x_n}] : t$$

For simplicity, in this lemma, we do not distinguish variables from accumulators, writing both using the metavariable x.

Proof. By induction on the derivation of $\Gamma, (x_1 : t_1), \ldots, (x_n : t_n) \vdash e : t$. We simply replace every instance of the rule [T-VAR] or [T-ACC] for variable x_i with a copy of the derivation of $\Gamma \vdash e_i : t_i$. ☐

Definition E.9. We write $[\![t]\!]_\mathcal{V}$ for $\{v \mid\ \vdash v : t\}$.

Lemma E.10. *If $t \leq s$, then $[\![t]\!]_\mathcal{V} \subseteq [\![s]\!]_\mathcal{V}$. In particular, if $t \simeq s$, then $[\![t]\!]_\mathcal{V} = [\![s]\!]_\mathcal{V}$.*

Proof. Consequence of the subsumption rule. ☐

Lemma E.11. $[\![\mathbb{0}]\!]_\mathcal{V} = \varnothing$.

Proof. We prove that $\vdash v : t$ implies $t \not\simeq \mathbb{0}$ by induction on the typing derivation. ☐

Lemma E.12. $[\![t_1 \wedge t_2]\!]_\mathcal{V} = [\![t_1]\!]_\mathcal{V} \cap [\![t_2]\!]_\mathcal{V}$.

Proof. By Lemma E.10, $[\![t_1 \wedge t_2]\!]_\mathcal{V} \subseteq [\![t_i]\!]_\mathcal{V}$ for $i = 1, 2$ and thus $[\![t_1 \wedge t_2]\!]_\mathcal{V} \subseteq [\![t_1]\!]_\mathcal{V} \cap [\![t_2]\!]_\mathcal{V}$. Lemma E.6 gives the opposite inclusion. ☐

Lemma E.13 (Inversion).

$$\begin{array}{ll} [\![t_1 \times t_2]\!]_\mathcal{V} & = \{(v_1, v_2) \mid\ \vdash v_1 : t_1,\ \vdash v_2 : t_2\} \\ [\![b]\!]_\mathcal{V} & = \{c \mid b_c \leq b\} \\ [\![t \to s]\!]_\mathcal{V} & = \{\mu f^{(t_1 \to s_1; \ldots; t_n \to s_n)}(x).e \in \mathcal{V} \mid \bigwedge_{i=1..n} t_i \to s_i \leq t \to s\} \\ [\![(t)_\tau]\!]_\mathcal{V} & = \{(w)_\delta \mid\ \vdash w : t,\ \vdash \delta : \tau\} \end{array}$$

Proof. For all four equalities, proving the $\supseteq$ inclusion is straightforward. We prove the $\subseteq$ inclusion by analyzing the typing derivation $\vdash v : t$, where t is instantiated to $t_1 \times t_2$, b, $t \to s$, or $(t)_\tau$ in each equality case. ☐

Lemma E.14. $[\![\neg t]\!]_\mathcal{V} = \mathcal{V} \setminus [\![t]\!]_\mathcal{V}$.

Proof. Note that $t \wedge \neg t \simeq \mathbb{0}$ and thus $[\![t]\!]_\mathcal{V} \cap [\![\neg t]\!]_\mathcal{V} = [\![t \wedge \neg t]\!]_\mathcal{V} = [\![\mathbb{0}]\!]_\mathcal{V} = \varnothing$. Hence, it remains to prove that $[\![t]\!]_\mathcal{V} \cup [\![\neg t]\!]_\mathcal{V} = \mathcal{V}$, that is: $\forall v, \forall t,\ \vdash v : t\ \vee\ \vdash v : \neg t$. We prove this statement by induction over the pair (v, t). ☐

Lemma E.15. $[\![t_1 \vee t_2]\!]_\mathcal{V} = [\![t_1]\!]_\mathcal{V} \cup [\![t_2]\!]_\mathcal{V}$.

Proof. By Lemmas E.10, E.12, and E.14. ☐

Definition E.16 (Sound operator). An operator $(o, n, \overset{o}{\rightsquigarrow}, \overset{o}{\to})$ is *sound* if and only if for all $v_1, \ldots, v_{n_o} \in \mathcal{V}$ such that $\vdash v_1 : t_1, \ldots, \vdash v_{n_o} : t_{n_o}$, the following holds:

$$\text{if } t_1, \ldots, t_{n_o} \overset{o}{\to} s \text{ and } v_1, \ldots, v_{n_o} \overset{o}{\rightsquigarrow} e, \text{ then } \vdash e : s$$

Lemma E.17. *Let p and φ respectively be a well-formed pattern and a zipper pattern. Let v and δ respectively be a closed value and a closed zipper. Then:*

(1) $\forall \sigma_0,\ \mathrm{Acc}(p) \subset \mathit{dom}(\sigma_0)\ \wedge\ \sigma_0 \vdash v/p \not\rightsquigarrow \Omega \implies\ \vdash v : \wr p \int$
(2) $\forall \sigma_0,\ \mathrm{Acc}(\varphi) \subset \mathit{dom}(\sigma_0)\ \wedge\ \sigma_0 \vdash \delta/\varphi \not\rightsquigarrow \Omega \implies\ \vdash \delta : \wr \varphi \int$

Proof. By simultaneous induction on derivations. Note that values are inductively defined and patterns are regular and contractive. ☐

Lemma E.18. *Let p be a well-formed pattern, t a type such that $t \leq \wr p \int$, x a variable such that $x \in \mathrm{Var}(p)$, and v a value. Then:*

$$\exists v', (\vdash v' : t)\ \wedge (\{\dot{x} \mapsto \mathit{Init}(\dot{x}) \mid \dot{x} \in \mathrm{Acc}(p)\} \vdash v'/p \rightsquigarrow \sigma, \gamma) \wedge (\gamma(x) = v) \Longrightarrow\ \vdash v : (t/p)(x)$$

Proof. By induction on a derivation of $\{\dot{x} \mapsto \mathsf{Init}(\dot{x}) \mid \dot{x} \in \mathrm{Acc}(p)\} \vdash v'/p \rightsquigarrow \sigma, \gamma$. ☐

Lemma E.19. *Let p be a well-formed pattern, t a type such that $t \leq \wr p \int$, $\dot{x}$ an accumulator such that $\dot{x} \in \mathrm{Acc}(p)$, and v a value. Moreover, suppose $\sigma_0 \equiv \Sigma_0 \equiv \{\dot{y} \mapsto \mathit{Init}(\dot{y}) \mid \dot{y} \in \mathrm{Acc}(p)\}$. Then:*

$$\exists v', (\vdash v' : t)\ \wedge (\sigma_0 \vdash v'/p \rightsquigarrow \sigma, \gamma) \wedge (\sigma(\dot{x}) = v) \Longrightarrow\ \vdash v : (t \underset{\Sigma_0}{/\!/} p)(\dot{x})$$

Proof. By induction on a derivation of $\sigma_0 \vdash v'/p \rightsquigarrow \sigma, \gamma$. □

Theorem E.20 (Type preservation). *If all operators in the language are* sound*, then typing is preserved by reduction:*

$$\text{if } e \rightsquigarrow e' \text{ and } \vdash e : t, \text{ then } \vdash e' : t$$

In particular, $e' \neq \Omega$.

Proof. By induction on the derivation of $\vdash e : t$. We proceed by a case analysis on the last rule used in the derivation of $\vdash e : t$.

- [T-CST]**:** the expression e is a constant (value). It cannot be reduced, which contradicts the assumption, and thus the result follows.
- [T-VAR]**:** the expression e is a variable. It cannot be well typed under the empty context, which contradicts the assumption and thus the result follows.
- [T-ACC]**:** similar to the [T-VAR] case.
- [T-ZIP-VAL]**:** similar to the [T-CST] case.
- [T-ZIP-EXPR]**:** let $\vdash (e_0)_\bullet : (s)_\bullet$ where $e \equiv (e_0)_\bullet$, $t \equiv (s)_\bullet$, $\vdash e_0 : s$, and $s \leq \mathbb{1}_{\mathsf{NZ}}$. Then, there exists an expression e'_0 such that $e_0 \rightsquigarrow e'_0$. We get $\vdash e'_0 : s$ by the induction hypothesis and then $\vdash (e'_0)_\bullet : (s)_\bullet$ by the rule [T-ZIP-EXPR].
- [T-PAIR]**:** let $\vdash (e_1, e_2) : t_1 \times t_2$ where $e \equiv (e_1, e_2)$, $t \equiv t_1 \times t_2$, $\vdash e_1 : t_1$, and $\vdash e_2 : t_2$. Then, there exists either an expression e'_1 such that $e_1 \rightsquigarrow e'_1$ or e'_2 such that $e_2 \rightsquigarrow e'_2$. If $e_1 \rightsquigarrow e'_1$, we get $\vdash e'_1 : t_1$ by the induction hypothesis and then $\vdash (e'_1, e_2) : t_1 \times t_2$ by the rule [T-PAIR]. The second case is similar to the first.
- [T-SUB]**:** there exists a type s such that $\vdash e : s$ and $s \leq t$. Since $e \rightsquigarrow e'$ by assumption, we have $\vdash e' : s$ by the induction hypothesis. Then, we get $\vdash e' : t$ by subsumption.
- [T-OP]**:** let $e \equiv o(e_1, \ldots, e_{n_o})$ where $\forall i = 1..n_o$, $\vdash e_i : t_i$ and $t_1, \ldots, t_{n_o} \overset{o}{\rightarrow} t$. There are two cases to consider:
  (1) Suppose $o(e_1, \ldots, e_i, \ldots, e_{n_o}) \rightsquigarrow o(e_1, \ldots, e'_i, \ldots, e_{n_o})$ where $e_i \rightsquigarrow e'_i$. We get $\vdash e'_i : t_i$ by the induction hypothesis and then $\vdash o(e_1, \ldots, e'_i, \ldots, e_{n_o}) : t$ by the rule [T-OP].
  (2) Suppose all e_i's are values and $(e_1, \ldots, e_{n_o}) \overset{o}{\rightsquigarrow} e'$. Since the operator $(o, n, \overset{o}{\rightsquigarrow}, \overset{o}{\rightarrow})$ is sound, we get $\vdash e' : t$ by Definition E.16.
- [T-FUN]**:** similar to the [T-CST] case.
- [T-MATCH]**:** let $e \equiv \mathsf{match}\ e_0\ \mathsf{with}\ p_1 \longrightarrow e_1 \mid p_2 \longrightarrow e_2$. We have two cases to consider. First, suppose $e_0 \rightsquigarrow e'_0$. Then, we complete the proof by using the induction hypothesis and the rule [T-MATCH]. Now, suppose e_0 is a value. Then, either the matching e_0/p_1 succeeds or it fails and the matching e_0/p_2 succeeds. Here, we show only the proof of the first case; the second case is similar. From $\Gamma \vdash e : t$, we have the following assumptions:

$$\begin{array}{c} (1)\ \ \vdash e_0 : s \qquad (2)\ \ s \leq \wr p_1 \int \vee \wr p_2 \int \qquad (3)\ \ s_1 \equiv s \wedge \wr p_1 \int \qquad (4)\ \ s_2 \equiv s \wedge \neg \wr p_1 \int \\ (5)\ \ \Sigma_i \equiv \{\dot{x} \mapsto \mathsf{Init}(\dot{x}) \mid \dot{x} \in \mathrm{Acc}(p_i)\} \qquad (6)\ \ s_i/p_i \cup s_i \underset{\Sigma_i}{/\!\!/} p_i \vdash e_i : t_i \qquad (7)\ \ t \equiv \bigvee_{\{i \mid s_i \not\simeq \mathbb{0}\}} t_i \end{array}$$

  - Let $\{\dot{x} \mapsto \mathsf{Init}(\dot{x}) \mid \dot{x} \in \mathrm{Acc}(p_1)\} \vdash e_0/p_1 \rightsquigarrow \sigma, \gamma$.
  - We get $\vdash e_0 : \wr p_1 \int$ by Lemma E.17.
  - Then, we get $\vdash e_0 : (s \wedge \wr p_1 \int) \equiv s_1$ by Lemma E.6.
  - By rewriting (6), we have $\{(x : (s_1/p_1)(x)) \mid x \in \mathrm{Var}(p_1)\} \cup \{(\dot{x} : (s_1 \underset{\Sigma_1}{/\!\!/} p_1)(\dot{x})) \mid \dot{x} \in \mathrm{Acc}(p_1)\} \vdash e_1 : t_1$.
  - Moreover, we have $\forall x \in \mathrm{Var}(p_1)$, $\vdash \gamma(x) : (s_1/p_1)(x)$ by Lemma E.18 and $\forall \dot{x} \in \mathrm{Acc}(p_1)$, $\vdash \sigma(\dot{x}) : (s_1 \underset{\Sigma_1}{/\!\!/} p_1)(\dot{x})$ by Lemma E.19.
  - Lemma E.8 then gives us $\vdash e_1[\sigma; \gamma] : t_1$.
  - Finally, by the rule [T-SUB], we get $\vdash e_1[\sigma; \gamma] : t$.

□

Corollary E.21. *The function application operator* app *is sound.*

Proof. Follows from Lemma E.8. □

Theorem E.22. *The operators* app, π_1, π_2, drm, rm, cons, *and* snoc *are sound.*

Proof. Corollary E.21 proves the soundness of the function application operator app. We prove the soundness of the operators π_1, π_2, rm, cons, and snoc by exploiting the fact that the types in the domains of their corresponding typing functions (especially for cons and snoc) are more precise than their exact input (defined in Lemma E.4). We prove the soundness of drm in a similar way, using the soundness of rm. □

Corollary E.23. *Let* $\mathcal{O}$ *be a set of operators* $\{\mathsf{app}, \pi_1, \pi_2, \mathsf{drm}, \mathsf{rm}, \mathsf{cons}, \mathsf{snoc}\}$. *Our core calculus equipped with* $\mathcal{O}$ *is sound in the sense that for any expression* e, *if* $\vdash e : t$, *then* $e \not\rightsquigarrow^* \Omega$.

Proof. The result follows from Theorems E.20 and E.22. □

F. Encoding of XPath Predicates into Regular Types

Definition F.1 (Encoding of XPath predicates into regular types). Given a predicate ϕ (produced by the rule *pred* of Defintion 5.1), we define the mutually recursive functions T_{pred}, T_{path}, T_{step}, T_{axis} and T_{test}:

$$
\begin{array}{lcl}
T_{\mathsf{pred}}(p) & = & T_{\mathsf{path}}(p) \\
T_{\mathsf{pred}}(\phi_1 \ \texttt{or} \ \phi_2) & = & T_{\mathsf{pred}}(\phi_1) \vee T_{\mathsf{pred}}(\phi_2) \\
T_{\mathsf{pred}}(\phi_1 \ \texttt{and} \ \phi_2) & = & T_{\mathsf{pred}}(\phi_1) \wedge T_{\mathsf{pred}}(\phi_2) \\
T_{\mathsf{pred}}(\texttt{not(}\phi\texttt{)}) & = & \neg T_{\mathsf{pred}}(\phi) \\
\\
T_{\mathsf{path}}(s) & = & T_{\mathsf{step}}(s, \mathbb{1}) \\
T_{\mathsf{path}}(s/p) & = & T_{\mathsf{step}}(s, T_{\mathsf{path}}(p)) \\
\\
T_{\mathsf{step}}(a \ \texttt{::} \ t[\phi], \tau) & = & T_{\mathsf{axis}}(a, \tau \wedge T_{\mathsf{test}}(t)) \wedge T_{\mathsf{pred}}(\phi) \\
\\
T_{\mathsf{test}}(l) & = & \mathtt{<}l\mathtt{>}\_ \\
T_{\mathsf{test}}(\texttt{*}) & = & \mathtt{<\_>}\_ \\
\\
T_{\mathsf{axis}}(\mathsf{self}, \tau) & = & \tau \\
T_{\mathsf{axis}}(\mathsf{child}, \tau) & = & \mathtt{<\_>}[\,\_ * \ \tau \ \_*\,] \\
T_{\mathsf{axis}}(\mathsf{desc\text{-}or\text{-}self}, \tau) & = & \mu X.\tau \vee \mathtt{<\_>}[\,\_ * \ X \ \_*\,] \\
T_{\mathsf{axis}}(\mathsf{desc}, \tau) & = & T_{\mathsf{axis}}(\mathsf{child}, T_{\mathsf{axis}}(\mathsf{desc\text{-}or\text{-}self}, \tau)) \\
T_{\mathsf{axis}}(\mathsf{foll\text{-}sibling}, \tau) & = & (\_)_{\mathsf{L}\,(\_,[\,\_* \, \tau \, \_*\,])\cdot\top} \\
T_{\mathsf{axis}}(\mathsf{parent}, \tau) & = & (\_)_{\mathsf{L}\,\_\cdot\mu X.((\mathsf{R}\,(\tau,\_)\cdot(\mathsf{L}\,\_\cdot\top\,|\,\bullet))\,|\,\mathsf{R}\,\_\cdot X)} \\
T_{\mathsf{axis}}(\mathsf{prec\text{-}sibling}, \tau) & = & (\_)_{\mathsf{L}\,\_\cdot\mu X.((\mathsf{R}\,(\tau,\_)\cdot\mathsf{R}\,\_\cdot\top)\,|\,\mathsf{R}\,\_\cdot X)} \\
T_{\mathsf{axis}}(\mathsf{anc}, \tau) & = & (\_)_{\mathsf{L}\,\_\cdot\mu X.((\mathsf{R}\,(\tau,\_)\cdot(\mathsf{L}\,\_\cdot\top\,|\,\bullet))\,|\,\mathsf{R}\,\_\cdot X\,|\,\mathsf{L}\,\_\cdot X)}
\end{array}
$$