A Typed Calculus for Querying Distributed XML Documents

Lucia Acciai[1], Michele Boreale[2], and Silvano Dal Zilio[1]

[1] Laboratoire d'Informatique Fondamentale de Marseille, France
[2] Dipartimento di Sistemi e Informatica, Università di Firenze, Italy

Abstract. We study the problems related to querying large, distributed XML documents. Our proposal takes the form of a new process calculus in which XML data are processes that can be queried by means of concurrent pattern-matching expressions. What we achieve is a functional, strongly-typed programming model based on three main ingredients: an asynchronous process calculus that draws features from π-calculus and concurrent-ML; a model where both documents and expressions are represented as processes, and where evaluation is represented as a parallel composition of the two; a static type system based on regular expression types.

1 Introduction

There is by now little doubt that XML will succeed as a lingua franca of data interchange on the Web. As a matter of fact, XML is a building block in the development of new models of concurrent applications, often referred to as Service-Oriented Architecture (SOA), where computational resources are made available on a network as a set of loosely-coupled, independent services.

The SOA model is characterized by the need to exchange and query XML documents. In this paper, we concentrate on the specific problems related to querying *large, distributed XML documents*. This is the case, for example, of applications interacting with distributed heterogeneous databases or that process data acquired dynamically, such as those originating from arrays of sensors (in this case, we can assume that the document is in effect infinite). For another example, consider the programs involved in the maintenance of the big Web indexes used by search engines [14]. A typical example is the computation of a *term vector*, that is a list of words found on some documents of the index together with their frequency. Distribution, concurrency and dynamic acquisition of data must be explicitly taken into account when designing an effective computational model for this kind of applications.

We most particularly pay attention to the processing model. Our proposal takes the form of a process calculus in which XML data are processes that can be queried by means of concurrent pattern-matching expressions. In this model, the evaluation of patterns is distributed among locations, in the sense that the evaluation of a pattern at a node triggers concurrent evaluation of sub-patterns at other nodes, and actions can be carried out upon success or failure of patterns. The calculus also provides primitives for storing and aggregating the results of intermediate computations and for orchestrating the evaluation of patterns. In this respect, we radically depart from previous works on XML-centered process calculi, see e.g. [2,9,17], where queries would be programmed as operations invoked on (servers hosting) Web Services and XML documents would be exchanged in messages. In contrast, we view queries as code being dispatched to the locations "hosting" a document. This shift of view is motivated by our target application domain. In particular, our model is partly inspired by the *MapReduce* paradigm described in [14] that is used to write programs to be executed on Google's large clusters of computers in a simple functional style. Continuing with the "term vector example" above, assume that the documents of interest are cached on different (maybe replicated) servers. A query that accomplishes the aforementioned task would dispatch sub-queries to every server and create a dedicated reference cell to aggregate the partial results from each server. Sub-queries sifts the local documents and transmit to the central reference cell a sequence of pairs (*word*, *frequency*) produced locally. The task of the aggregating function is to collect the frequencies for identical keywords as they arrive, so as to eventually produce the global term vector. To achieve reliability, sub-queries may have to report back periodically with status updates while the "master query" may decide to abort or reinstate queries in case of servers failure.

Another important feature of our model is the definition of a static type system based on *regular expression types* that is compatible with Document Type Definitions (DTD) and other XML schema languages. What we achieve is a functional, strongly-typed programming model for computing over distributed XML documents based on three main ingredients: a semantics defined by an asynchronous process calculus in the style of the π-calculus [23] and proposed semantics for concurrent-ML [16]; a model where documents and expressions are both represented as processes, and where evaluation is represented as a parallel composition of the two; a type system based on regular expression types (the soundness of the static semantics is proved via a subject reduction property, Theorem 4.1). Each of these choices is motivated by a feature of the problem: the study of service-oriented applications calls for including concurrency and explicit locations; the need to manipulate large, possibly dynamically generated, documents calls for a streamed model of processing; the documents handled by a service should often obey a predefined schema, hence the need to check that queries are well-typed, preferably before they are executed or "shipped".

The rest of the paper is organized as follows. Section 2 presents the core components of the calculus — documents, types and patterns — and Section 3 gives the formal semantics of the calculus. In Section 4 we define a first-order type system with subtyping based on regular expression types and prove the soundness of our type discipline. Before concluding with a review of related works, we study possible extensions of our model in Section 5. Omitted proofs may be found in a long version of this paper [3].

2 Documents, Types and Patterns

We consider a simple language of first-order functional expressions, denoted $e, e', \ldots,$ enriched with references and recursive pattern definitions that are used to extract values from documents. Patterns are built on top of a syntax for defining regular tree grammars [13], which is also at the basis of our type system.

Documents. An XML document may be seen as a simple textual representation for nested sequences of elements <a>...</a>. In this paper, we follow notations similar to [21] and choose a simplified version of documents by leaving aside attributes among other things. We assume an infinite set of *tag names*, ranged over by $\mathtt{a}, \mathtt{b}, \ldots$ (we will often choose the symbol $\boldsymbol{o}$ for the tag of the root element of a document). A document is an ordered sequence of elements $\mathtt{a}_1[v_1] \ldots \mathtt{a}_n[v_n]$, where $v_1, \ldots, v_n$ are documents. Documents may be empty, denoted `()`, and can be concatenated, denoted v, v'. The composition operation is associative with identity `()`.

In the following we consider distributed documents, meaning that each element $\mathtt{a}_\mathtt{j}[v_j]$ is placed in a given location, say $\imath_j$. Locations are visible only at the level of the operational semantics, in which the contents of a document is represented by the index $\imath_1 \ldots \imath_n$ (the list of locations) of its elements. For the sake of simplicity, locations and indexes are the only values handled in our calculus and we leave aside atomic data values such as strings or integers.

Document Types. Applications that exchange and process XML documents rely on type information, such as DTDs, to describe structural constraints on the occurrences of elements. In our model, types take the form of regular tree expressions, which are a set of recursive definitions of the form $A := \mathit{Reg}(\mathtt{a}_\mathtt{i}[A_i])_{i \in 1..n}$, where Reg is a regular expression and $A, A_1, \ldots, A_n$ are type variables. This is essentially a syntax for defining regular tree grammar. A regular expression $\mathit{Reg}(\alpha_i)_{i \in 1..n}$ can be an atom α_i with $i \in 1..n$; it can be the constant `All`, which matches everything, or `Empty`, which matches the empty sequence; it can be a choice $\mathit{Reg}_1 \mid \mathit{Reg}_2$, a sequential composition $\mathit{Reg}_1, \mathit{Reg}_2$, or an iteration $\mathit{Reg}*$. For instance, the declaration below defines the type L of family trees, which are sequences of male or female person such that each person has a `name` element, and two elements, `d` and `s`, for the list of his daughters and sons.

$$\begin{array}{llll} L & := (\mathtt{man}[P] \mid \mathtt{woman}[P])* & P & := \mathtt{name}[\mathtt{All}], \mathtt{d}[\mathit{WL}], \mathtt{s}[\mathit{ML}] \\ \mathit{WL} & := \mathtt{woman}[P]* & \mathit{ML} & := \mathtt{man}[P]*. \end{array}$$

There is a natural notion of subtyping $A <: B$ between regular expression types, meaning that every document in A is also in B. The type system is close to what is defined in functional languages for manipulating XML, see e.g. XDuce [19,20,21] or the review in [10], hence we stay consistent with actual frameworks used in sequential languages for processing XML data.

Selectors and Patterns. The core of our programming model is a system of distributed pattern matching expressions that concurrently sift through documents to extract information. Basically, patterns are types enhanced with parameters and capture variables. However, like functions, patterns are declared and have a name.

We assume a countable set of *names*, partitioned into *locations* $\imath, \jmath, \ell, \ldots$ and *variables* $x, y, \ldots$ We use the vector notation $\vec{x}$ for tuples of names. The declaration $p(\vec{x}) := \big(\mathit{Reg}(\mathtt{a}_\mathtt{i}[p_i(\vec{y}_i)])_{i \in 1..n}\big)$ `as` y defines a pattern called p, with parameters $\vec{x}$, that will collect matched documents in the reference y (where y is a variable in $\vec{x}$).

For instance, the patterns defined below can be used to extract the names of persons occurring in a document of type L.

$$\begin{aligned} names(x,y) \; &:= \; \big(\texttt{man}[p(x,y,x)] \,|\, \texttt{woman}[p(x,y,y)]\big)* \\ p(x,y,z) \; &:= \; \texttt{name}[all(z)], \texttt{d}[names(x,y)], \texttt{s}[names(x,y)] \\ all(z) \; &:= \; \texttt{All as } z. \end{aligned}$$

A call to $names(\imath, \ell)$ stores in (the reference located at) $\imath$ the name of men and in ℓ the name of women. A call to $names(\ell, \ell)$ will store the names of all persons in ℓ. Actually, the most general form of pattern declaration allows `let` definitions and setting continuations to be evaluated upon success or failure of the pattern, i.e. a pattern declaration is of the form, where S is a *selector* $Reg(\texttt{a}_\texttt{i}[p_i(\vec{y_i})])_{i \in 1..n}$:

$$p(\vec{x}) \; := \; \texttt{let} \; \big(z_1 = e'_1, \ldots, z_m = e'_m\big) \; \texttt{in} \; \big(S \; \texttt{as} \; y\big) \; \texttt{then} \; e_1 \; \texttt{else} \; e_2 \;\; ,$$

An important feature of our model is that patterns may extract multiple sets of values from documents in one pass, which contrasts with the monadic queries expressible with technologies such as XPath. In the next section, we give a formal definition of the calculus, which embeds an operator $\texttt{try}\, v\, p(\vec{u})$ for applying the pattern p to the value v. During reduction, the index v is matched against S after all the expressions $e'_1, \ldots, e'_m$ have been evaluated. If the matching succeeds, then v is added to the values stored in y and e_1 is evaluated. Otherwise, the compensation e_2 is evaluated. These optional continuations allow to add basic exception and transaction mechanisms to the calculus.

Clearly, types are particular kind of patterns: a pattern declaration without parameters, `let` definitions, capture variables and continuations is a type declaration. Moreover, every pattern p can be associated with the type A obtained by erasing these extra information: A is the type of all documents that are matched by p.

In the following, we assume that functions and patterns are typed explicitly. For instance, we assume that the pattern *names* is declared with the type $(\texttt{All}, \texttt{All}) \rightarrow L$. More generally, a reference that merges values of type B will have a type A such that $A, B <: A$.

Witness and Unambiguous Patterns. We define formally what it means for a pattern to match an index and define a notion of *unambiguous* patterns. Assume S is the selector $Reg(\texttt{a}_\texttt{i}[p_i(\vec{v_i})])_{i \in 1..n}$. The sequence $\texttt{a}_{\texttt{i}_1} \ldots \texttt{a}_{\texttt{i}_\texttt{m}}$ matches S if and only if it is a "word" in the language of $Reg(\texttt{a}_\texttt{i})_{i \in 1..n}$. This relation is denoted $\texttt{a}_{\texttt{i}_1} \ldots \texttt{a}_{\texttt{i}_\texttt{m}} \vdash_S p_{i_1}(\vec{v}_{i_1}) \ldots p_{i_m}(\vec{v}_{i_m})$ and we call $(p_{i_j}(\vec{v}_{i_j}))_{j \in 1..m}$ a *witness* for S of $\texttt{a}_{\texttt{i}_1} \ldots \texttt{a}_{\texttt{i}_\texttt{m}}$. We write $\texttt{a}_{\texttt{i}_1} \ldots \texttt{a}_{\texttt{i}_\texttt{m}} \not\vdash_S$ if the sequence has no witness for S.

It is standard in XML to restrict to expressions that denote sequences of elements unequivocally. We say that a pattern with selector S is *unambiguous* if each sequence of tags has at most one witness for S. Assume that $(p_{i_j}(\vec{v}_{i_j}))_{j \in 1..m}$ is "the witness" of S for $\texttt{b}_1 \ldots \texttt{b}_\texttt{m}$. When a document $\texttt{b}_1[v_1] \ldots \texttt{b}_\texttt{m}[v_m]$ is matched against a pattern with selector S, each sub-document v_j is matched against $p_{i_j}(\vec{v}_{i_j})$. If $\texttt{b}_1 \ldots \texttt{b}_\texttt{m}$ has no witness then the pattern-matching fails.

3 The Calculus

The presentation of the calculus can be naturally divided into two fragments: a language of functional expressions, or *programs*, that are used in the body of pattern and function declarations; and a language of processes, or *configurations*, that models distributed documents and the concurrent execution of programs. Typically, expressions are “program sources” that should be evaluated (they do not contain references to active locations), while a configuration represents the running state of a set of processes.

Programs. The calculus embeds a first-order functional language with references, pattern-matching and constructs for building documents. In the following, we assume that every function identifier f has associated arity $n \geqslant 0$ and a unique definition $f(\vec{x}) := e$ where the variables in $\vec{x}$ are distinct and include the free variables of e. We take similar hypotheses for patterns. The syntax of expressions $e, e', \ldots$ is given below:

$u, v ::=$	results
$\quad x$	name: variable or location
$\quad \imath_1 \ldots \imath_n$	index (with $n \geqslant 0$)
$e ::=$	expressions
$\quad u$	result
$\quad a[u]$	element creation
$\quad u, v$	result composition
$\quad f(u_1, \ldots, u_n)$	function call
$\quad \texttt{let}\, x = e_1 \,\texttt{in}\, e_2$	let
$\quad \texttt{newref}\, u$	new reference (with initial value u)
$\quad !u$	dereferencing
$\quad u \mathrel{+}= v$	update (adds v to the values stored in u)
$\quad \texttt{try}\, u\, p(u_1, \ldots, u_n)$	pattern matching call
$\quad \texttt{wait}\, u(x)\, \texttt{then}\, e_1\, \texttt{else}\, e_2$	wait matching

A result is either a name or an index, i.e. an expression that immediately returns itself. Expressions include results, operators for creating new elements $\mathtt{a}[u]$, for concatenating indexes u, v, and for creating, accessing and updating references. Expressions also include operators for applying a pattern to a document index, `try`, and for branching on the result of pattern-matching, `wait`.

Configurations. The syntax of processes $P, Q, \ldots$ is as follows:

$P, Q, R ::=$	processes
$\quad e$	expression
$\quad \texttt{let}\, x = P \,\texttt{in}\, Q$	let
$\quad \langle \imath \mapsto d \rangle$	location
$\quad P \Rsh Q$	parallel composition
$\quad (\nu\imath)P$	restriction
$d ::=$	resources
$\quad \texttt{ref}\, u$	reference with value u
$\quad \texttt{node}\, \mathtt{a}(u)$	node, element tagged $\mathtt{a}$ with index u
$\quad \texttt{try}\, \imath\, p(u_1, \ldots, u_n)$	try matching
$\quad \texttt{test}\, \imath\, u\, v_k$	test matching
$\quad \texttt{ok}\, \imath$	successful match
$\quad \texttt{fail}\, \imath$	failed match

The calculus features operators from the π-calculus: restriction $(\nu\imath)P$ specifies the scope of a name $\imath$ local to P; parallel composition $P \upharpoonright Q$ represents the concurrent evaluation of P and Q. Overall, a process is a multiset of `let` expressions, describing threads execution, and locations $\langle\, \imath \mapsto d\, \rangle$, that describes a *resource* d located in $\imath$.

The calculus is based on an abstract notion of location that is, at the same time, the minimal unit of interaction and the minimal unit of storage. Failures are not part of this model (they can be viewed as an orthogonal feature) but could be added, e.g. in the style of [5]. Locations store resources. The main resources are `ref` u, to store the current state of a reference, and `node` $\mathtt{a}(u)$, to describe an element of the form $\mathtt{a}[u]$. The calculus explicitly takes into account the distribution of document nodes and, for example, the document $\mathtt{a}[\mathtt{b}[\,]\,\mathtt{c}[\,]]$ can be represented (at runtime) by the process: $(\nu\imath_1\imath_2)\big(\langle\, \imath \mapsto \mathtt{node}\ \mathtt{a}(\imath_1\, \imath_2)\, \rangle \upharpoonright \langle\, \imath_1 \mapsto \mathtt{node}\ \mathtt{b}(\,)\, \rangle \upharpoonright \langle\, \imath_2 \mapsto \mathtt{node}\ \mathtt{c}(\,)\, \rangle\big)$. The other resources arise in the evaluation of pattern-matching and correspond to different phases in its execution: scheduling a "pattern call" (`try`); waiting for the result of sub-patterns (`test`); stopping and reporting success (`ok`) or failure (`fail`).

Syntactic conventions: the operators `let`, `wait` and ν are name binders. Notions of α-equivalence and of free and bound names arise as expected: we denote $\mathit{fv}(P)$ the set of variables that occur free in P and $\mathit{fn}(P)$ the set of free names. We identify expressions and terms up-to α-equivalence. Substitutions are finite partial maps from variables to results: we write $P\{x{\leftarrow}u\}$ for the simultaneous, capture-avoiding substitution of all free occurrences of x in P with u. Assume σ is the substitution $\{x_1{\leftarrow}u_1\}\dots\{x_n{\leftarrow}u_n\}$ and $\vec{u} = (u_1, \dots, u_n)$. We write $f(\vec{u}) := e'$ if $f(\vec{x}) := e$ and $e' = \sigma(e)$ and we write $p(\vec{u}) := S'$ if the selector of $p(\vec{x})$ is S and $S' = \sigma(S)$. Finally, we make use of the following abbreviations: if $u = \imath_1 \dots \imath_n$ then $(\nu u)P$ is a shorthand for $(\nu\imath_1)\dots(\nu\imath_n)P$; the term $(\nu\ell)P \upharpoonright Q$ stands for $((\nu\ell)P) \upharpoonright Q$; the term $\mathtt{let}\ x = P\ \mathtt{in}\ Q \upharpoonright R$ stands for $(\mathtt{let}\ x = P\ \mathtt{in}\ Q) \upharpoonright R$; and $\mathtt{wait}\ \ell(x)\ \mathtt{then}\ e_1$ stands for $\mathtt{wait}\ \ell(x)\ \mathtt{then}\ e_1\ \mathtt{else}\ ()$ (and similarly for omitted `then` clause).

Reduction Semantics. The semantics of our calculus follows the chemical style found in the π-calculus [23]: it is based on structural congruence and a reduction relation. Reduction represents individual computation steps and is defined in terms of structural congruence and evaluation contexts.

Structural congruence $\equiv$ allows the rearrangement of terms so that reduction rules may be applied. It is the least congruence on processes to satisfy the following axioms:

(Struct Par Assoc)
$$\frac{}{(P \upharpoonright Q) \upharpoonright R \equiv P \upharpoonright (Q \upharpoonright R)}$$

(Struct Par Let)
$$\frac{x \notin \mathit{fn}(P)}{P \upharpoonright \mathtt{let}\ x = Q\ \mathtt{in}\ R \equiv \mathtt{let}\ x = (P \upharpoonright Q)\ \mathtt{in}\ R}$$

(Struct Par Com)
$$\frac{}{(P \upharpoonright Q) \upharpoonright R \equiv (Q \upharpoonright P) \upharpoonright R}$$

(Struct Res Let)
$$\frac{\ell \notin \mathit{fn}(Q)}{(\nu\ell)\mathtt{let}\ x = P\ \mathtt{in}\ Q \equiv \mathtt{let}\ x = (\nu\ell)P\ \mathtt{in}\ Q}$$

(Struct Res Res)
$$\frac{}{(\nu\imath)(\nu\ell)P \equiv (\nu\ell)(\nu\imath)P}$$

(Struct Res Par R)
$$\frac{\imath \notin \mathit{fn}(P)}{(\nu\imath)(P \upharpoonright Q) \equiv P \upharpoonright (\nu\imath)Q}$$

(Struct Res Par L)
$$\frac{\imath \notin \mathit{fn}(Q)}{(\nu\imath)(P \upharpoonright Q) \equiv ((\nu\imath)P) \upharpoonright Q}$$

(Struct Let Assoc)

$$\frac{x \notin fn(R)}{\texttt{let}\, y = (\texttt{let}\, x = P\, \texttt{in}\, Q)\ \texttt{in}\, R \equiv \texttt{let}\, x = P\ \texttt{in}\, (\texttt{let}\, y = Q\ \texttt{in}\, R)}$$

Since processes may return values, we take the convention that the result of a composition $P_1 \Rsh \ldots \Rsh P_n$ is the result of its rightmost term P_n. The values returned by the other processes are discarded. This entails that the order of parallel components is relevant. For this reason, unlike the situation in most process calculi, parallel composition is "left commutative" but not commutative: we have $(P \Rsh Q) \Rsh R$ equivalent to $(Q \Rsh P) \Rsh R$ but not necessarily $P \Rsh Q \equiv Q \Rsh P$. This choice is similar to what is found in calculi introduced for defining the semantics of concurrent-ML [16] and for concurrent extension of object calculi [18]. An advantage of this approach is that we directly include sequential composition of processes: the term $P; Q$ can be interpreted by `let` $x = P$ `in` Q, where $x \notin fv(Q)$. We also obtain a more direct style of programming since the operation of returning a result does not require using continuations and sending a message on a result channel, as in the π-calculus.

Reduction $\rightarrow$ is the least binary relation on closed terms to satisfy the rules in Table 1. The rules for expressions are similar to traditional semantics for first-order languages, with the difference that the resources in a configuration play the role of the store. Likewise, the rules for operators that return new values (the operators `newref`, `a[]` and `try`) yields reductions of the form $e \rightarrow (\nu\ell)(\langle\, \ell \mapsto d \,\rangle \Rsh \ell)$, which means that new values are always allocated in a fresh location. Actually a quick inspection of the rules shows that resources are created in fresh locations and are always used in a linear way: an expression cannot discard a resource or create two different resources at the same location.

Informal Semantics. We can divide the rules in Table 1 according to the locations involved in the reduction. A location $\langle\, \ell \mapsto \texttt{ref}\ w \,\rangle$ is a reference at ℓ with value w. Reference access, rule (Red Read), replaces a top-level occurrence of $!\ell$ with the value w. Reference update $\ell \mathrel{+}= v$, rule (Red Write), has a slightly unusual semantics since its effect is to append v to the value stored in ℓ. Actually, we could imagine that each reference is associated with an "aggregating function" (denoted `op` in Table 1) that specifies how the sequence of values stored in the reference has to be combined. For example, assume ℓ is an "integer reference" that increments its value by one on every assignment. Then, in the example of Section 2, a call to $names(\ell, \ell)$ counts the number of people in a document of type L. We only consider index composition in this work.

A location $\langle\, \ell \mapsto \texttt{try}\ \imath\, p(\vec{v}) \,\rangle$ is created by the evaluation of a `try` operator. The expression $\texttt{try}\ u\, p(\vec{v})$ applies the pattern p to the index $u = \imath_1 \ldots \imath_n$, rule (Red Try). A `try` expression returns at once with the index ℓ of the fresh location where the matching occurs. It also creates a document node $\langle\, \imath \mapsto \texttt{node}\ \boldsymbol{o}(u) \,\rangle$ that points to the index u that is processed (we use the reserved name $\boldsymbol{o}$ for the root tag of this node). Assume that S is the selector of p, the `try` resource will trigger evaluation of sub-patterns selected from a witness of S. If there is no witness, the matching fails, rule (Red Try Error). If a witness exists, the `try` resource spawns new `try` resources and turns into a `test`, rule (Red Try Match), waiting for the results of these evaluations. Upon termination of all the sub-patterns, a `test` resource turns into `ok` or `fail`, rules (Red Test Ok) and (Red Test Fail). The `ok` and `fail` resources are immutable.

(Red Fun)

$$\frac{f \text{ declared as } f(\vec{x}) := e}{f(u_1,\ldots,u_n) \rightarrow e\{x_1{\leftarrow}u_1\}\ldots\{x_n{\leftarrow}u_n\}}$$

(Red Let)

$$\frac{}{\texttt{let}\ x = u\ \texttt{in}\ P \rightarrow P\{x{\leftarrow}u\}}$$

(Red Struct)

$$\frac{P \equiv Q, \quad Q \rightarrow Q', \quad Q' \equiv P'}{P \rightarrow P'}$$

(Red Context)$^{(\star)}$

$$\frac{P \rightarrow P'}{E[P] \rightarrow E[P']}$$

(Red Ref)

$$\frac{u = \imath_1 \ldots \imath_n}{\texttt{newref}\ u \rightarrow (\nu\ell)(\langle\, \ell \mapsto \texttt{ref}\ u \,\rangle \Rsh \ell)}$$

(Red Read)

$$\frac{}{\langle\, \ell \mapsto \texttt{ref}\ u \,\rangle \Rsh !\ell \rightarrow \langle\, \ell \mapsto \texttt{ref}\ u \,\rangle \Rsh u}$$

(Red Write)$^{(\star\star)}$

$$\frac{w = u, v}{\langle\, \ell \mapsto \texttt{ref}\ u \,\rangle \Rsh \ell \mathrel{+}= v \rightarrow \langle\, \ell \mapsto \texttt{ref}\ w \,\rangle \Rsh \texttt{()}}$$

(Red Node)

$$\frac{u = \imath_1 \ldots \imath_n}{\mathtt{a}[u] \rightarrow (\nu\imath)(\langle\, \imath \mapsto \texttt{node}\ \mathtt{a}(u) \,\rangle \Rsh \imath)}$$

(Red Comp)

$$\frac{u_1 = \imath_1 \ldots \imath_k \quad u_2 = \imath_{k+1} \ldots \imath_n}{u_1, u_2 \rightarrow \imath_1 \ldots \imath_n}$$

(Red Try)

$$\frac{u = \imath_1 \ldots \imath_n \qquad \imath, \ell \text{ fresh names}}{\texttt{try}\ u\ p(\vec{v}) \rightarrow (\nu\imath)(\nu\ell)(\langle\, \imath \mapsto \texttt{node}\ \boldsymbol{o}(u) \,\rangle \Rsh \langle\, \ell \mapsto \texttt{try}\ \imath\ p(\vec{v}) \,\rangle \Rsh \ell\,)}$$

(Red Try Match)

$$\frac{\begin{array}{c} P = \langle\, \imath \mapsto \texttt{node}\ \mathtt{a}(\imath_1 \ldots \imath_n) \,\rangle \Rsh \prod_{l \in 1..n} \langle\, \imath_l \mapsto \texttt{node}\ \mathtt{a}_l(w_l) \,\rangle \\ p(\vec{v}) := S\ \texttt{as}\ v_k \quad \mathtt{a}_1 \ldots \mathtt{a}_\mathtt{n} \vdash_S p_1(\vec{v}_1) \ldots p_n(\vec{v}_n) \qquad w = \jmath_1 \ldots \jmath_n \text{ fresh names} \end{array}}{P \Rsh \langle\, \ell \mapsto \texttt{try}\ \imath\ p(\vec{v}) \,\rangle \rightarrow P \Rsh (\nu w)\big(\prod_{l \in 1..n} \langle\, \jmath_l \mapsto \texttt{try}\ \imath_l\ p_l(\vec{v_l}) \,\rangle \Rsh \langle\, \ell \mapsto \texttt{test}\ \imath\ w\ v_k \,\rangle\big)}$$

(Red Try Error)

$$\frac{P = \langle\, \imath \mapsto \texttt{node}\ \mathtt{a}(\imath_1 \ldots \imath_n) \,\rangle \Rsh \prod_{k \in 1..n} \langle\, \imath_k \mapsto \texttt{node}\ \mathtt{a}_k(w_k) \,\rangle \quad p(\vec{v}) := S\ \texttt{as}\ v_k \quad \mathtt{a}_1 \ldots \mathtt{a}_\mathtt{n} \nvdash_S}{P \Rsh \langle\, \ell \mapsto \texttt{try}\ \imath\ p(\vec{v}) \,\rangle \rightarrow P \Rsh \langle\, \ell \mapsto \texttt{fail}\ \imath \,\rangle}$$

(Red Test Ok)

$$\frac{P = \langle\, \imath \mapsto \texttt{node}\ \mathtt{a}(\imath_1 \ldots \imath_n) \,\rangle \Rsh \prod_{k \in 1..n} \langle\, \jmath_k \mapsto \texttt{ok}\ \imath_k \,\rangle \quad w = \jmath_1 \ldots \jmath_n \quad x \text{ fresh name}}{P \Rsh \langle\, \ell \mapsto \texttt{test}\ \imath\ w\ v_k \,\rangle \rightarrow P \Rsh \texttt{let}\ x = (v_k \mathrel{+}= (\imath_1 \ldots \imath_n))\ \texttt{in}\ \langle\, \ell \mapsto \texttt{ok}\ \imath \,\rangle}$$

(Red Test Fail)

$$\frac{\begin{array}{c} P = \langle\, \imath \mapsto \texttt{node}\ \mathtt{a}(\imath_1 \ldots \imath_n) \,\rangle \Rsh \prod_{k \in 1..n} \langle\, \jmath_k \mapsto d_k \,\rangle \qquad w = \jmath_1 \ldots \jmath_n \\ \forall k \in 1..n:\ d_k \in \{\texttt{ok}\ \imath_k, \texttt{fail}\ \imath_k\} \qquad \exists j \in 1..n:\ d_j = \texttt{fail}\ \imath_j \end{array}}{P \Rsh \langle\, \ell \mapsto \texttt{test}\ \imath\ w\ v_k \,\rangle \rightarrow P \Rsh \langle\, \ell \mapsto \texttt{fail}\ \imath \,\rangle}$$

(Red Wait Ok)

$$\frac{P = \langle\, \imath \mapsto \texttt{node}\ \mathtt{a}(u) \,\rangle \Rsh \langle\, \ell \mapsto \texttt{ok}\ \imath \,\rangle}{P \Rsh \texttt{wait}\ \ell(x)\ \texttt{then}\ e_1\ \texttt{else}\ e_2 \rightarrow P \Rsh e_1\{x{\leftarrow}u\}}$$

(Red Wait Fail)

$$\frac{P = \langle\, \imath \mapsto \texttt{node}\ \mathtt{a}(u) \,\rangle \Rsh \langle\, \ell \mapsto \texttt{fail}\ \imath \,\rangle}{P \Rsh \texttt{wait}\ \ell(x)\ \texttt{then}\ e_1\ \texttt{else}\ e_2 \rightarrow P \Rsh e_2\{x{\leftarrow}u\}}$$

$^{(\star)}$ where $\quad E ::= Q \Rsh E \mid E \Rsh P \mid [.] \mid (\nu\ell)E \mid \texttt{let}\ x = E\ \texttt{in}\ P$

$^{(\star\star)}$ in the general case we have $w = \mathtt{op}(u, v)$, where op is some "aggregating" function

Table 1. Reductions

The remaining rules are related to the evaluation of a `wait` expression. The status of a pattern evaluation can be checked with the expression `wait` $\ell(x)$ `then` e_1 `else` e_2, see rules (Red Wait Ok) and (Red Wait Fail). If the resource at ℓ is `ok` $\imath$ then the `wait` expression evaluates to $e_1\{x{\leftarrow}v\}$, where v is the index of the node located at $\imath$. If the resource is `fail` $\imath$ then the expression evaluates to $e_2\{x{\leftarrow}v\}$. In all the other cases the expression is stalled.

Remark. In rule (Red Try Match), we compute the witness for all the children of an element in one go. This is not always realistic since the size of the children's index can be very large (actually, in real applications, big documents are generally shallow and have a large number of children). It is possible to refine the operational semantics so that each sub-pattern is fired independently, not necessarily following the order of the document. For instance, we should be able to start the evaluation on an element without necessarily matching all its preceding siblings beforehand. Also, we can imagine that indexes are implemented using streams or linked lists. We have chosen this presentation for sake of simplicity (it is one of the simplifications used in this paper so that we can concentrate on the innovative features of the calculus and its type system).

4 Static Semantics

The types of document indexes are the same than the types for documents defined in Section 2. Apart from regular expressions types A, the type t of a process can also be the resource type $\star$ (a constant type for terms that return no values); a reference type `ref` A; a node type `node` $\mathtt{a}(u)$ for the type of a location holding an element $\mathtt{a}[u]$; or a try type `loc` $\mathtt{a}(A)$, that is the type of a location hosting the evaluation of a pattern of type A on the contents of an element tagged $\mathtt{a}$.

$t ::=$	type
$\star$	no value
A	regular expression type
`ref` A	reference
`node` $\mathtt{a}(u)$	node location
`loc` $\mathtt{a}(A)$	try location

We can easily adapt the definition of witness to types (a type is some sort of selector). Assume A is declared as $A := \mathit{Reg}(\mathtt{a_i}[A_i])_{i\in 1..n}$. We say that there is a witness for A of $\mathtt{a_{i_1}}\ldots\mathtt{a_{i_m}}$, denoted $\mathtt{a_{i_1}}\ldots\mathtt{a_{i_m}} \vdash_A A_{i_1}\ldots A_{i_m}$, if and only if the sequence of tags $\mathtt{a_{i_1}}\ldots\mathtt{a_{i_m}}$ is in the language of the regular expression $\mathit{Reg}(\mathtt{a_i})_{i\in 1..n}$. We can define the language of a type A as the set of documents that are matched by the pattern $\mathit{Reg}(\mathtt{a_i}[A_i])_{i\in 1..n}$. Based on this definition, we obtain a natural notion of subtyping $A <: B$, meaning that the language of A is included in the language of B. We write $A \doteq B$ if the languages of A and B are equal. We write $\overline{A}$ for some chosen regular expression type whose language is the complement of A. (The type $\overline{A}$ is unnecessary when $A \doteq \mathtt{All}$, which means that we do not need to introduce a type with an empty language.) In the case of type witness, we have $\mathtt{a_{i_1}}\ldots\mathtt{a_{i_m}} \not\vdash_A$ if and only if there is a witness for $\overline{A}$ of $\mathtt{a_{i_1}}\ldots\mathtt{a_{i_m}}$.

The type system is given in Table 2. A type environment E is a finite mapping $x_1 : t_1, \ldots, x_n : t_n$ between names and types. The type system is based on a single

(Type x)
$$\frac{}{E, x : t, E' \vdash x : t}$$

(Type Sub)
$$\frac{A <: B \quad E \vdash P : A}{E \vdash P : B}$$

(Type Fun)
$$\frac{f : (t_1, \ldots, t_n) \to t_0 \quad E \vdash u_i : t_i \quad i \in 1..n}{E \vdash f(\vec{u}) : t_0}$$

(Type Let)
$$\frac{E \vdash P : t \quad E, x{:}t \vdash Q : t'}{E \vdash \mathtt{let}\; x = P \;\mathtt{in}\; Q : t'}$$

(Type Doc)
$$\frac{E \vdash \imath_k : \mathtt{node}\; \mathsf{a}_\mathsf{k}(u_k) \quad E \vdash u_k : B_k \quad k \in 1..n}{E \vdash \imath_1 \ldots \imath_n : \mathsf{a}_1[B_1], \ldots, \mathsf{a}_\mathsf{n}[B_n]}$$

(Type Node)
$$\frac{E \vdash u : A}{E \vdash \mathsf{a}[u] : \mathsf{a}[A]}$$

(Type Comp)
$$\frac{E \vdash u_i : A_i \quad i \in \{1, 2\}}{E \vdash u_1, u_2 : A_1, A_2}$$

(Type Ref)
$$\frac{E \vdash u : A}{E \vdash \mathtt{newref}\; u : \mathtt{ref}\; A}$$

(Type Read)
$$\frac{E \vdash u : \mathtt{ref}\; A}{E \vdash !u : A}$$

(Type Write)
$$\frac{E \vdash u : \mathtt{ref}\; A \quad E \vdash v : B \quad A, B <: A}{E \vdash u \mathrel{+}= v : \mathtt{Empty}}$$

(Type Res)
$$\frac{E, \ell_1 : t_1, \ldots, \ell_n : t_n \vdash P : t \quad \{\ell_1, \ldots, \ell_n\} \cap \mathit{fn}(E) = \emptyset}{E \vdash (\nu\ell_1) \ldots (\nu\ell_n) P : t}$$

(Type Par)
$$\frac{E \vdash P : t' \quad E \vdash Q : t}{E \vdash P \upharpoonright Q : t}$$

(Type Try Doc)
$$\frac{p : (t_1, \ldots, t_n) \to A \quad E \vdash v_i : t_i \quad i \in 1..n \quad E \vdash u : B}{E \vdash \mathtt{try}\; u\; p(v_1, \ldots, v_n) : \mathtt{loc}\; \boldsymbol{o}(A)}$$

(Type Wait)
$$\frac{E \vdash u : \mathtt{loc}\; \mathsf{a}(A) \quad E, x : A \vdash e_1 : t \quad E, x : \overline{A} \vdash e_2 : t}{E \vdash \mathtt{wait}\; u(x)\; \mathtt{then}\; e_1\; \mathtt{else}\; e_2 : t}$$

(Type Loc Ref)
$$\frac{E \vdash \ell : \mathtt{ref}\; A \quad E \vdash u : A}{E \vdash \langle\, \ell \mapsto \mathtt{ref}\; u \,\rangle : \star}$$

(Type Loc Node)
$$\frac{E \vdash \ell : \mathtt{node}\; \mathsf{a}(\imath_1 \ldots \imath_n)}{E \vdash \langle\, \ell \mapsto \mathtt{node}\; \mathsf{a}(\imath_1 \ldots \imath_n) \,\rangle : \star}$$

(Type Loc Ok)
$$\frac{E \vdash \ell : \mathtt{loc}\; \mathsf{a}(A) \quad E \vdash \imath : \mathtt{node}\; \mathsf{a}(u) \quad u = \imath_1 \ldots \imath_n \quad E \vdash u : A}{E \vdash \langle\, \ell \mapsto \mathtt{ok}\; \imath \,\rangle : \star}$$

(Type Loc Fail)
$$\frac{E \vdash \ell : \mathtt{loc}\; \mathsf{a}(A) \quad E \vdash \imath : \mathtt{node}\; \mathsf{a}(u) \quad u = \imath_1 \ldots \imath_n \quad E \vdash u : \overline{A}}{E \vdash \langle\, \ell \mapsto \mathtt{fail}\; \imath \,\rangle : \star}$$

(Type Try Loc)
$$\frac{E \vdash \ell : \mathtt{loc}\; \mathsf{a}(A) \quad E \vdash \imath : \mathtt{node}\; \mathsf{a}(\imath_1 \ldots \imath_n) \quad p : (t_1, \ldots, t_n) \to A \quad E \vdash v_i : t_i \quad i \in 1..n}{E \vdash \langle\, \ell \mapsto \mathtt{try}\; \imath\; p(\vec{v}) \,\rangle : \star}$$

(Type Test Loc)
$$\frac{E \vdash \ell : \mathtt{loc}\; \mathsf{a}(A) \quad E \vdash \imath : \mathtt{node}\; \mathsf{a}(u) \quad E \vdash \jmath_k : \mathtt{loc}\; \mathsf{a}_\mathsf{k}(A_k) \quad w = (\jmath_1 \ldots \jmath_n) \quad \mathsf{a}_1 \ldots \mathsf{a}_\mathsf{n} \vdash_A A_1 \ldots A_n \quad E \vdash v_k : t_k \quad t_k = \mathtt{ref}\; B \quad B, A <: B}{E \vdash \langle\, \ell \mapsto \mathtt{test}\; \imath\; w\; v_k \,\rangle : \star}$$

Table 2. Typing Rules

type judgment, $E \vdash P : t$, meaning that the process P has type t under the hypothesis E. We assume that there is a given, fixed set of type declarations of the form $A := \mathit{Reg}(\mathsf{a_i}[A_i])_{i \in 1..n}$. We assume that functions and patterns are well-typed, which is denoted $f : \vec{t} \to t_0$ and $p : \vec{t} \to A$. The types $t_1, \ldots, t_n$ in $\vec{t}$ are the types of the parameters, while t_0 is the type of the body of f and A is the type of the selector of p. The type of a selector $S = \mathit{Reg}(\mathtt{a_i}[p_i(\vec{x_i})])_{i\in 1..n}$ is obtained from S by substituting to every pattern p_i in the selector its corresponding type A_i. Hence the type of S is equivalent to some type variable A such that $A := \mathit{Reg}(\mathtt{a_i}[A_i])_{i\in 1..n}$. Note that if a pattern $p(\vec{x}) := S$ `as` x_k has type $\vec{t} \to A$, then the type t_k is *compatible* with A, which means that $t_k = \mathtt{ref}\ B$ and $B, A <: B$.

The typing rules for the functional part of the calculus are standard. In what follows, we consider that references can only hold document values: a reference is of type `ref` A and not `ref` t. Moreover, since a reference collects the sequence of values that are assigned to it, we check for every assignment of a value of type B into a reference of type `ref` A that the relation $A, B <: A$ holds, see rule (Type Write). This check allows us to enforce statically the type of references.

The remaining typing rules are for resources and pattern-matching operators. The type of an expression `try` $u\ p(\vec{v})$ is `loc` $\boldsymbol{o}(A)$ if the pattern p matches documents of type A, see rule (Type Try Doc). Indeed the effect of this expression is to return a fresh location hosting the evaluation of p on an element of the form $\boldsymbol{o}[u]$. Correspondingly, a `wait` expression is well typed only if it is blocking on a location of type `loc` $a(A)$, that is the location of a resource that can eventually turn into `ok` or `fail`. The important aspect of this rule is that, while the continuations e_1 and e_2 of the `wait` expression must have the same type, they are typed under different typing environment: the expression e_1 is typed with the hypothesis $x : A$ while e_2 is typed with the hypothesis $x : \overline{A}$. This leads to more precise types for filtering expressions (see below).

The typing rules for locations are straightforward. Since a resource returns no value it has type $\star$. By rule (Type Try Loc), a location ℓ containing a `try` resource, evaluating a pattern p of type A, is well typed if ℓ is of type `loc` $\mathtt{a}(A)$ and the root tag of the evaluated document is `a`. Note that no assumption is made on $(\imath_1, \ldots, \imath_n)$, which might well not be of type A. Finally, the rule for node location, (Type Loc Node), states that a location containing `node` $\mathtt{a}(u)$ has only one possible type, namely `node` $\mathtt{a}(u)$ itself. Hence this rule avoids the presence of two `node` resources with the same location but containing different elements. Actually, we could extend our type system in a simple way to ensure that a well-typed configuration cannot have two resources at the same location: we say that the configuration is *well-formed* (see e.g. [18] for an example of how to add simple linearity constraints to the type system to ensure well-formedness).

An important feature of our calculus is that every pattern is strongly typed: its type is the regular expression obtained by erasing capture variables. Likewise we can type locations, expressions and processes using a combination of regular expression types and `ref` types. As it is often the case with typed languages, the first important property we need to prove is that well-typedness of processes is preserved by reduction.

Theorem 4.1 (subject reduction). *Suppose that P is well formed and contains only unambiguous patterns and t contains only unambiguous types. If $E \vdash P : t$ and $P \to Q$ then $E \vdash Q : t$.*

The proof of this property is by induction on the derivation of the relation $P \to Q$. The proof is quite involved since it is not possible to reason on a whole document at once: its content is scattered across distinct resource locations. This complexity reflects actual restrictions imposed when working with distributed documents, e.g. that they can never be checked locally. The proof of Theorem 4.1 is given in [3].

Note that we do not state a *progress theorem* in connection with Theorem 4.1. Indeed, there exists no notion of errors in our calculus (like e.g. the notion of "message not understood" in object-oriented languages) as it is perfectly acceptable for a pattern matching to fail or to get blocked on a `wait` statement. Nonetheless the subject reduction theorem is still useful. For instance, we can use it for optimizations purposes, like detecting that a specific matching will always fail.

5 Examples and Possible Extensions

We study examples that show how to interpret interesting programming idioms in our model, like spawning an expression in a new thread or handling user-defined exceptions.

Types and Pattern-Matching. We can encode a "traditional" `match` operator, as found in XDuce for example, that matches the pattern p against u and conditionally proceeds with e_1 or e_2. Assume y is a fresh variable ($y \notin fv(e_1) \cup fv(e_2)$), we define:

$$\texttt{match}\; u\; \texttt{with}\; p(\vec{v})\; \texttt{then}\; e_1\; \texttt{else}\; e_2 \;=_{\text{def}}\; \begin{cases} \texttt{let}\; x = \big(\texttt{try}\; u\; p(\vec{v})\big) \\ \;\texttt{in}\; \big(\texttt{wait}\; x(y)\; \texttt{then}\; e_1\; \texttt{else}\; e_2\big) \end{cases} .$$

This example allows us to emphasize the role of the variable y when typing a `wait` statement. Let $e =_{\text{def}} \big(\texttt{match}\; z\; \texttt{with Empty then a}[z]\; \texttt{else}\; z\big)$ be the expression that returns z if it is not empty else returns $\texttt{a}[z]$. Assume z is a variable of type `All`, then the most precise type for e is also `All`. In contrast, if we consider the expression $\texttt{let}\, x = \big(\texttt{try}\, z\, \texttt{Empty}\big)\, \texttt{in}\, \big(\texttt{wait}\, x(y)\; \texttt{then a}[y]\; \texttt{else}\; y\big)$, which is equivalent to e, we obtain the more precise type $\overline{\texttt{Empty}}$, that is, we prove that the returned value cannot be empty. Indeed y plays the role of an alias for the value of z that is used with type `Empty` in the continuation $\texttt{a}[y]$ and with type $\overline{\texttt{Empty}}$ in y (and we have $\texttt{a}[\texttt{Empty}] <: \overline{\texttt{Empty}}$).

Concurrency. We show how to model simple threads, that is, we want to encode an operator `spawn` such that the effect of $\texttt{spawn}\; e_1;\; e_2$ is to evaluate e_1 in parallel with e_2, yielding the value of e_2 as a result. The simplest solution is to interpret $\texttt{spawn}\, e_1;\; e_2$ by the configuration $e_1 \Rsh e_2$. A disadvantage of this solution is that it is not possible to test in e_2 whether the evaluation of e_1 has ended.

Another simple approach to encode `spawn` is to rely on the pattern-matching mechanism. Let p be the pattern $p(\,) := (\texttt{Empty then}\; e_1)$. We can interpret the statement $\texttt{spawn}\, e_1; e_2$ with the expression $\texttt{let}\; x = (\texttt{try ()}\; p(\,))\; \texttt{in}\; e_2$. Indeed we have:

$$\texttt{let}\, x = (\texttt{try ()}\; p(\,))\, \texttt{in}\, e_2 \;\rightarrow^{*}\; (\nu\imath\ell)\big(\langle\, \imath \mapsto \texttt{node}\; \boldsymbol{o}(\,)\,\rangle \Rsh \\ \big(\texttt{let}\; z = e_1\; \texttt{in}\; \langle\, \ell \mapsto \texttt{ok}\; \imath\,\rangle\big) \Rsh e_2\{x{\leftarrow}\ell\}\big) \;.$$

In the resulting process, e_1 and e_2 are evaluated concurrently and the resource $\langle\, \ell \mapsto \texttt{ok}\; \imath\,\rangle$ cannot interact with e_2 until the evaluation of e_1 ends (see rule (Struct Par Let) for example). Hence an occurrence of the expression $(\texttt{wait}\; x(y)\; \texttt{then}\; e)$ in e_2

acts as an operator blocking the execution of e until e_1 returns a value. We can in fact improve our encoding so that the result of e_1 is bound to z in e as follows:

$$\texttt{spawn}\, e_1;\ e_2 \ =_{\text{def}}\ (\nu\imath\ell) \begin{pmatrix} \texttt{let}\ z = e_1\ \texttt{in}\ \big(\langle\, \imath \mapsto \texttt{node}\, \boldsymbol{o}(z)\,\rangle \Rsh \langle\, \ell \mapsto \texttt{ok}\, \imath\,\rangle\big) \\ \Rsh\, e_2\{x {\leftarrow} \ell\} \end{pmatrix} \ .$$

It emerges from this example that a `try` location can be viewed as a *future*, that is a reference to the "future result" of an asynchronous computation. More generally, we can liken a process $(\langle\, \imath \mapsto \texttt{node}\ \texttt{a}(u)\,\rangle \Rsh \langle\, \ell \mapsto \texttt{ok}\, \imath\,\rangle)$ to an (asynchronous) output action $\ell!\langle \texttt{ok}, u\rangle$ as found in process calculi such as the π-calculus. Similarly, we can compare an expression $\texttt{wait}\ \ell(x)\ \texttt{then}\ e_1\ \texttt{else}\ e_2$ with a combination of input action and matching, $\ell?(x).\{\texttt{ok} \Rightarrow e_1 \mid \texttt{fail} \Rightarrow e_2\}$, with the following synchronization rules:

$$\begin{array}{lcl} \ell!\langle \texttt{ok}, u\rangle \ \ \| \ \ell?(x).\{\texttt{ok} \Rightarrow e_1 \mid \texttt{fail} \Rightarrow e_2\} & \rightarrow & \ell!\langle \texttt{ok}, u\rangle \ \| \ e_1\{x {\leftarrow} u\} \\ \ell!\langle \texttt{fail}, u\rangle \ \| \ \ell?(x).\{\texttt{ok} \Rightarrow e_2 \mid \texttt{fail} \Rightarrow e_2\} & \rightarrow & \ell!\langle \texttt{ok}, u\rangle \ \| \ e_2\{x {\leftarrow} u\} \end{array}$$

The main distinction with "traditional process calculi" is that we are in a situation where inputs are replicated. For this reason, we can have multiple `wait` operators synchronizing on the same location ℓ without the need for global consensus (or a lock) on the resource at ℓ. Nonetheless, since the calculus can express atomic reads and writes on a shared memory, it could be useful to rely on a standard mutual exclusion algorithm for accessing references. We could also interpret high-level primitives for mutexes directly in our calculus (see e.g. [18] for an example). Note also that there is no need for replication in our calculus since resources are persistent and recursive behaviors can be encoded using recursive function declarations.

Exceptions. We show how to model a simple exception mechanism in our calculus. Suppose we need to check that a document u of type L (the type of family trees) contains only women. This can be achieved using the pattern declarations $p(\,) := \texttt{woman}[q(\,)]*$ and $q(\,) := \texttt{name}[\texttt{All}], \texttt{d}[p(\,)], \texttt{s}[\texttt{Empty}]$ and a matching expression $\texttt{try}\ u\ p(\,)$. A drawback of this approach is that we need to wait for the completion of all sub-patterns to terminate before completing the computation, even if the matching trivially fails because we find an element tagged `man` early in the matching. A natural optimization is to use an explicit handling of failures, e.g. to add primitives to kill and "ping" (the location of) a `try` resource in the style of [5]. Another solution is to encode a basic mechanism for handling exceptions using the following derived operators, where $\imath_e$ is a default name associated to the location $\langle\, \imath_e \mapsto \texttt{node}\, \boldsymbol{o}(\,)\,\rangle$:

$$\begin{array}{rcll} \texttt{exception} & = & (\nu\ell)\ell & \text{creates a fresh (location) exception} \\ \texttt{throw}\ \ell & = & \langle\, \ell \mapsto \texttt{ok}\, \imath_e\,\rangle \Rsh \texttt{()} & \text{raises an exception at } \ell \\ \texttt{catch}\ \ell\ e & = & \texttt{wait}\ \ell(x)\ \texttt{then}\ e & \text{catches exception } \ell \text{ and runs } e\ \ (x \notin \mathit{fv}(e)) \end{array}$$

A simple example is to raise the exception at the end of a computation, like in the expression $\texttt{let}\ x\ =\ \texttt{exception}\ \texttt{in}\ \big((\ldots; \texttt{throw}\ x) \Rsh \texttt{catch}\ x\ e\big)$. If and when the `throw` expression is evaluated, we obtain a configuration of the form $(\nu l)\big(\ldots \Rsh \langle\, \ell \mapsto \texttt{ok}\, \imath_e\,\rangle \Rsh \texttt{wait}\ \ell(x)\ \texttt{then}\ e\big)$, which starts the execution of e. For instance, it is possible to raise the exception in the compensation part of a pattern declaration and to redefine the pattern p above in: $p(x) := \texttt{woman}[q(\,)]*\ \texttt{else}\ \texttt{throw}\ x$.

With our encoding, it is not possible to abort the execution of a whole "program block" using exceptions. Using a more involved encoding, e.g. based on CPS transforms, we could interpret this more general exception model.

6 Future and related work

We study a formal model for computing over large, perhaps dynamic, distributed XML documents. We define a typed process calculus and show that it supports a first-order type system with subtyping based on regular expression types, a system compatible with DTD and other schema languages for XML. Our work may be compared with recent proposals for integrating XML data into π-calculus. It can also be compared with proposals for filtering and querying XML streams (or so-called *XML pipelining* frameworks) for which there exists almost no formal foundations.

Related Work. There are a few works mixing XML with process calculi: Iota [6] is a concurrent XML scripting language with channel-based communications that relies on types to guarantee the well-formedness (not the validity) of documents; XPi [2] is a typed π-calculus extended with XML values in which documents are exchanged during communications; PiDuce [9] features asynchronous communications and code mobility and includes pattern matching expressions with built-in type checks. In all these proposals, documents are first class values exchanged in messages, which make these approaches inappropriate in the case of very large or dynamically generated data. At the opposite, we consider documents as special kind of processes that can be randomly accessed through the use of distributed indexes.

Works on querying XML streams can be roughly divided in two approaches. The first is to provide efficient single-pass evaluator, working with one query at a time (generally XPath queries) on multiple documents. The second approach, in relation to peer-to-peer and event-notification systems, is to filter XML streams by a large number of queries. We look more closely at some examples of such systems. SPEX, XSQ and XSM [8,12,22] are single-pass evaluators of XPath queries in which queries are compiled into networks of independent, deterministic pushdown transducers with buffers. The query language in XSM is severely restricted and only streams with non-recursive structure definitions can be processed (this is akin to non-recursive types in our framework). XFilter, YFilter and XTrie [4,15,11] follow the second approach. XFilter is a filtering system based on finite state machines (FSM). It uses one FSM per path query and an indexing mechanism to allow all FSMs to be executed simultaneously during the processing of a document. YFilter extends XFilter using a lazy NFA-based representation in which state transitions for simultaneous queries are precomputed (hence exploiting commonalities among path queries). Likewise, XTrie is based on decomposing tree patterns into collection of substrings and indexing them using a trie with the purpose to share the processing of "common sub-queries".

Our work follows the first approach with some differences (patterns extend XPath queries and `try`-statements apply one pattern to one document at a time). Most notably, we take a strongly typed approach and, instead of using XPath or XQuery, we extend the functional approach taken in e.g. XDuce and define distributed *regular expression* *pattern*. As a byproduct, we also provide a possible semantics for a concurrent extensions of languages based on XDuce. Nonetheless, since our operational semantics does not dictate how regular patterns should be implemented, we can take inspiration from these systems to implement efficient and scalable filtering primitives in our calculus. Conversely, we could use our calculus to give a formal semantics to these systems.

Future Work The goal of this paper is not to define a new programming language. We rather try to provide formal tools for the study of concurrent computation models based on service composition and streamed XML data. However our calculus could be a basis for developing concurrent extensions of strongly typed languages for XML, such as XDuce. It could also be used to provide the semantics of systems in which XML documents contain active code that can be executed on distributed sites (i.e. processes and document text are mixed), like in the Active XML system for example [1]. To this end, it will be necessary to add an "`eval/quote`" mechanisms, as in LISP or multi-stage programming languages [24], and to fundamentally revise our static type checking approach.

Our work raises questions concerning observational equivalences that we intend to study in future work. Another avenue to investigate is the encoding of other concurrency related primitives, like channel-based synchronization and distributed transactions, or the possibility to dynamically update documents.

References

1. Abiteboul S., Benjelloun O., Milo T., Manolescu I., Weber R.: Active XML: Peer-to-Peer Data and Web Services Integration. In *Proc. of VLDB*, 2002.
2. Acciai L., Boreale M.: XPi: a typed process calculus for XML messaging. In *Proc. of FMOODS*, LNCS vol. 3535, Springer, 2005.
3. Acciai L., Boreale M., Dal Zilio, S.: A Typed Calculus for Querying Distributed XML Documents. LIF Research Report xx, 2006.
4. Altinel M., Franklin M.J.: Efficient filtering of XML documents for selective dissemination information. In *Proc. of the 26th VLDB Conference*, 2000.
5. Amadio R.: An Asynchronous Model of Locality, Failure And Process Mobility. In *Proc. of COORDINATION*, LNCS vol. 1282, Springer, 1997.
6. Bierman G., Sewell P.: Iota: A concurrent XML scripting language with applications to Home Area Networking. TR 577, Computer Lab., Cambridge, 2003.
7. Brüggemann-Klein A., Wood D.: One-unambiguous regular languages. Information and Computation, 142(2), 1998.
8. Bry F., Furche T., Olteanu D.: An efficient single-pass query evaluator for XML data structure. TR PMS-FB-2004-1, Computer Science Institute, Munich, 2004.
9. Brown A., Laneve C., Meredith G.: PiDuce: a process calculus with native XML datatypes. In *Proc. of Workshop on Web Services and Formal Methods*, 2005.
10. Castagna G.: Pattern and types for querying XML documents. In *Proc. of DBPL*, XSYM 2005 joint keynote talk, 2005.
11. Chan C.Y., Felber P., Garofalakis M., Rastogi R.: Efficient filtering of XML documents with XPath expressions. The VLDB Journal 11, 2002.
12. Chawathe S.S., Peng F.: XPath Queries on Streaming Data. In *Proc. of SIGMOD*, 2003.
13. Comon H., Dauchet M., Jacquemard F., Tison S., Lugiez D., Tommasi M.: *Tree Automata on their application*. 1999. `http://www.grappa.univ-lille3.fr/tata/`

14. Dean J., Ghemawat, S.: MapReduce: Simplified Data Processing on Large Cluster. In *Proc. of OSDI*, 2004.
15. Diao Y., Fisher P., Franklin M.J.: Yfilter: efficient and scalable filtering of XML documents. In *Proc. of 18th ICDE*, IEEE, 2002.
16. Ferreira W., Hennessy M., Jeffrey A.S.: A theory of weak bisimulation for core CML. J. Functional Programming 8(5), 1998.
17. Gardner P., Maffeis S.: Modelling dynamic web data. Theor. Comput. Sci. 342(1) (2005).
18. Gordon A.D., Hankin P.D.: A concurrent object calculus: reduction and typing. In *Proc. of HLCL*. Electr. Notes Theor. Comput. Sci. 16(3), 1998.
19. Hosoya H., Vouillon J., Pierce B.J.: Regular expression types for XML. ACM Transactions on Programming Languages and Systems, 27(1), 2004.
20. Hosoya H., Pierce B.J.: Regular expression pattern matching for XML. In *Proc. of POPL*, 2001.
21. Hosoya H., Pierce B.J.: XDuce: A Statically Typed XML Processing Language. In *Proc. of ACM Transaction on Internet Technology*, 2003.
22. Ludäscher B., Mukhopadhyay P., Papakonstantinou Y.: A Tranducer-Based XML Query Processor. In *Proc. of VLDB*, 2002.
23. Milner R.: Communicating and Mobile Systems: The π-Calculus. CUP , 1999.
24. Taha W., Sheard T.: MetaML and multi-stage programming with explicit annotations. Theor. Comput. Sci. 248(1-2), 2000.