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 (LIF), CNRS and Université de Provence, 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 [9]. A typical example is the computation of a *reverse web-link graph*, that is a list of web pages which contain a link to a common target URL. 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 needed in this situation. 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

[⋆] The first and third author are supported by the French government research grant ACI TRALALA. The second author is supported by the EU within the FET-GC2 initiative, project SENSORIA.

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 (sub-)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. [2,6,11], 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 [9], that is used to write programs to be executed on Google's large clusters of computers in a simple functional style. Continuing with the "reverse web-link graph example" above (developed in Section 5), assume that the documents of interest are cached on different, perhaps 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 sift the local documents and transmit to the central reference cell sequences of pages with a link to the target URL, so as to eventually produce the global reverse web-link graph. 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*, an approach that matches well 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 [16] and proposed semantics for concurrent-ML [10]; 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 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 and an example of pattern-matching evaluation. 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, we develop the example of the reverse web-link graph (Section 5) and we study possible extensions of our model (Section 6). Appendice A contains proof of Theorem 1. 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 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 [8], 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 [15] 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 $\mathtt{a}, \mathtt{b}, \ldots$. 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 operator "$,$" is associative with identity `()`).

In the following we consider distributed documents, meaning that each element $\mathtt{a}_j[v_j]$ 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 simplicity, locations and indexes are the only values handled in our calculus and we leave aside atomic data values such as characters or integers.

Document Types. Applications that exchange and process XML documents rely on type information, such as DTD, to describe structural constraints on the occurrences of elements in a "valid" document. In our model, types take the form of regular tree expressions [8], which are sets of recursive definitions of the form $A := \mathit{Reg}(\mathtt{a}_i[A_i])_{i \in 1..n}$, where Reg is a regular expression and $A, A_1, \ldots, A_n$ are type variables. 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 people such that each person has a `name` element, and two elements, `d` and `s`, for the list of his daughters and sons:

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

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 [13,14,15] or the review in [7], 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 $\boldsymbol{x}$ for tuples of names. The declaration $p(\boldsymbol{x}) := \big(\mathit{Reg}(\mathtt{a}_i[p_i(\boldsymbol{y}_i)])_{i\in 1..n}\big)$ $\mathtt{as}$ y defines a pattern called p, with parameters $\boldsymbol{x}$, that collects matched documents in the reference y (where y, the *capture variable* of p, should occur in $\boldsymbol{x}$). For instance, the following patterns can be used to collect the names of male and female people occurring in a document of type L (see example in the next subsection):

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

In its most general form, a pattern declaration also allows `let` definitions and setting continuations to be evaluated upon success or failure of the pattern. (These optional continuations make it possible to add basic exception and transaction mechanisms to the calculus.) Hence, a pattern declaration is of the following form, where S is a *selector* $\mathit{Reg}(\mathtt{a}_i[p_i(\boldsymbol{y}_i)])_{i\in 1..n}$.

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

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. Also, types appears clearly as a particular kind of patterns (patterns declared without parameters, `let` definitions and continuations), and every pattern p can be associated with the type A obtained by erasing these extra information. In this case, A is exactly the type of all documents that are matched by p.

Witness and Unambiguous Patterns. It is standard in XML to restrict to expressions that denote sequences of elements unequivocally. 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 $\mathit{Reg}(\mathtt{a}_i[p_i(\boldsymbol{v_i})])_{i\in 1..n}$. The sequence $\mathtt{a}_{i_1}\ldots\mathtt{a}_{i_m}$ matches S if and only if it is a word in the language of $\mathit{Reg}(\mathtt{a}_i)_{i\in 1..n}$. This relation is denoted $\mathtt{a}_{i_1}\ldots\mathtt{a}_{i_m} \vdash_S p_{i_1}(\boldsymbol{v}_{i_1})\ldots p_{i_m}(\boldsymbol{v}_{i_m})$ and we call the sequence $(p_{i_j}(\boldsymbol{v}_{i_j}))_{j\in 1..m}$ a *witness* for S of $\mathtt{a}_{i_1}\ldots\mathtt{a}_{i_m}$. We write $\mathtt{a}_{i_1}\ldots\mathtt{a}_{i_m} \not\vdash_S$ if the sequence has no witness for S.

We say that a pattern with selector S is *unambiguous* if each sequence of tags has at most one witness for S. Assume that $(p_{i_j}(\boldsymbol{v}_{i_j}))_{j\in 1..m}$ is the witness of S for $\mathtt{b}_1\ldots\mathtt{b}_m$. When a document $\mathtt{b}_1[v_1]\ldots\mathtt{b}_m[v_m]$ is matched against a pattern with selector S, each sub-document v_j is matched against $p_{i_j}(\boldsymbol{v}_{i_j})$. If $\mathtt{b}_1\ldots\mathtt{b}_m$ has no witness then pattern-matching fails.

For instance, when matching the "pattern-call" $\mathit{names}(\imath,\ell)$ against a list of people, the contents of elements tagged `man` is matched by $p(\imath,\ell,\imath)$, which involves that the value of the `name` element inside `man` is matched by $\mathit{all}(\imath)$. From the capture variable in all, this results in storing the name in (the reference located at) $\imath$. More generally, a call to $\mathit{names}(\imath,\ell)$ stores the names of men in $\imath$ and women in ℓ. A call to $\mathit{names}(\ell,\ell)$ stores all the names in ℓ.

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

A result is either a name or an index. Expressions include results, operators for creating new elements $\mathtt{a}[u]$, for concatenating indexes $u\boldsymbol{,}\, v$, 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
e	expression
$\mathtt{let}\ x = P\ \mathtt{in}\ Q$	let
$\langle \imath \mapsto d \rangle$	location
$P \Rsh Q$	parallel composition
$(\nu \imath) P$	restriction
$d ::=$	resources
$\mathtt{ref}\ u$	reference with value u
$\mathtt{node}\ \mathtt{a}(u)$	node, element tagged $\mathtt{a}$ with index u
$\mathtt{try}\ \imath\ p(u_1, \ldots, u_n)$	try matching
$\mathtt{test}\ \imath\ u\ v_k$	test matching
$\mathtt{ok}\ \imath$	successful match
$\mathtt{fail}\ \imath$	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 $P \Rsh Q$ represents the concurrent evaluation of P and Q. Overall, a process is a sequence of `let` expressions, describing threads execution, and locations $\langle\, \imath \mapsto d \,\rangle$, that describes a *resource* d located in $\imath$. Hence the syntax of configurations is very expressive as it unifies the notions of expression, store, thread and processes.

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 [4]. 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: $(\nu\imath_1\imath_2)\big(\langle\, \imath \mapsto \mathtt{node}\ \mathtt{a}(\imath_1\,\imath_2)\,\rangle \Rsh \langle\, \imath_1 \mapsto \mathtt{node}\ \mathtt{b}(\,)\,\rangle \Rsh \langle\, \imath_2 \mapsto \mathtt{node}\ \mathtt{c}(\,)\,\rangle\big)$. 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 $\boldsymbol{u} = (u_1,\dots,u_n)$. We write $f(\boldsymbol{u}) := e'$ if $f(\boldsymbol{x}) := e$ and $e' = \sigma(e)$ and we write $p(\boldsymbol{u}) := S'$ if the selector of $p(\boldsymbol{x})$ 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 $(\nu\ell)P \Rsh Q$ stands for $((\nu\ell)P) \Rsh Q$; the term $\mathtt{let}\ x = P\ \mathtt{in}\ Q \Rsh R$ stands for $(\mathtt{let}\ x = P\ \mathtt{in}\ Q) \Rsh R$; 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 [16]: 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)
$$\frac{}{(P \Rsh Q) \Rsh R \equiv P \Rsh (Q \Rsh R)}$$

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

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

(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)
$$\frac{\imath \notin \mathit{fn}(P)}{(\nu\imath)(P \Rsh Q) \equiv P \Rsh (\nu\imath)Q}$$

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

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

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 [10] and for concurrent extensions of object calculi [12]. An advantage of this approach is that we directly include sequential composition of processes: the term $P; Q$ can be interpreted by $\mathtt{let}\, x = P \,\mathtt{in}\, Q$, where $x \notin \mathit{fv}(Q)$. 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`) yield 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 $\langle\, \ell \mapsto \mathtt{ref}\ w \,\rangle$ 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[1].

A location $\langle\, \imath \mapsto \mathtt{node}\ \mathtt{a}(u) \,\rangle$ is created by the evaluation of an element creation expression $\mathtt{a}[u]$, where u is an index, (Red Node). A location $\langle\, \ell \mapsto \mathtt{try}\ \imath\ p(\boldsymbol{v}) \,\rangle$ is created by the evaluation of a `try` operator. The expression $\mathtt{try}\ u\ p(\boldsymbol{v})$ 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 $\langle\, \imath \mapsto \mathtt{node}\ \boldsymbol{o}(u) \,\rangle$ that points to the index u that is processed (we use the reserved name

[1] 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 $\mathit{names}(\ell, \ell)$ counts the number of people in a document of type L. For the sake of simplicity, we only consider index composition in this work.

(Red Fun)

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

(Red Let)

$$\frac{}{\mathtt{let}\ x = u\ \mathtt{in}\ P \to P\{x \leftarrow u\}}$$

(Red Struct)

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

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

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

(Red Ref)

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

(Red Read)

$$\frac{}{\langle\, \ell \mapsto \mathtt{ref}\ u \,\rangle \Rsh !\ell \to \langle\, \ell \mapsto \mathtt{ref}\ u \,\rangle \Rsh u}$$

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

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

(Red Node)

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

(Red Comp)

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

(Red Try)

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

(Red Try Match)

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

(Red Try Error)

$$\frac{P = \langle\, \imath \mapsto \mathtt{node}\ \mathsf{a}(\imath_1 \ldots \imath_n) \,\rangle \Rsh \prod_{k \in 1..n} \langle\, \imath_k \mapsto \mathtt{node}\ \mathsf{a}_k(w_k) \,\rangle \quad p(\boldsymbol{v}) := S\ \mathtt{as}\ v_k \quad \mathsf{a}_1 \ldots \mathsf{a}_n \not\vdash_S}{P \Rsh \langle\, \ell \mapsto \mathtt{try}\ \imath\ p(\boldsymbol{v}) \,\rangle \to P \Rsh \langle\, \ell \mapsto \mathtt{fail}\ \imath \,\rangle}$$

(Red Test Ok)

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

(Red Test Fail)

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

(Red Wait Ok)

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

(Red Wait Fail)

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

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

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

Table 1. Reductions

$\boldsymbol{o}$ 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.

The remaining rules are related to the evaluation of a `wait` expression. The status of a pattern evaluation can be checked with the expression $\texttt{wait}\,\ell(x)\ \texttt{then}\ e_1\ \texttt{else}\ e_2$, see rules (Red Wait Ok) and (Red Wait Fail). If the resource at ℓ is $\texttt{ok}\,\imath$ 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 $\texttt{fail}\,\imath$ 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 the sake of simplicity.

Example: pattern-matching evaluation. As an example of pattern-matching evaluation, consider the pattern p below, which extracts all the sub-elements tagged `a` and discards elements tagged `b`.

$$p(x)\ :=\ (\mathtt{a}[p(x)]\ \texttt{as}\ x \mid \mathtt{b}[p(x)])*$$

Let d be the document $\mathtt{a}[\mathtt{b}[\mathtt{a}[\,]]]\ \mathtt{b}[]$. We assume that the elements of d are stored at the indexes $(\imath_1\imath_4)$, that is d is represented by the process:

$$(\nu\imath_2\imath_3)\big(\langle\,\imath_1 \mapsto \texttt{node}\ \mathtt{a}(\imath_2)\,\rangle \Rsh \langle\,\imath_2 \mapsto \texttt{node}\ \mathtt{b}(\imath_3)\,\rangle \Rsh \langle\,\imath_3 \mapsto \texttt{node}\ \mathtt{a}()\,\rangle \Rsh \langle\,\imath_4 \mapsto \texttt{node}\ \mathtt{b}()\,\rangle\big).$$

The following expression starts the pattern-matching evaluation of p against d:

$$\texttt{let}\ x = \texttt{newref ()}\ \texttt{in}\ \texttt{try}\ (\imath_1\imath_4)\ p(x).$$

In what follows we show the matching evaluation step-by-step. By rules (Red Ref) and (Red Let), a new location containing a new reference (a "capture" reference) is created and substituted to x in the pattern invocation. By applying structural equivalence we obtain

$$\rightarrow (\nu\ell')\big(\langle\,\ell' \mapsto \texttt{ref ()}\,\rangle \Rsh \texttt{try}\ (\imath_1\imath_4)\ p(\ell')\big).$$

By rule (Red Try), two fresh locations are created: ℓ, where the pattern-matching is evaluated, and $\imath$, where the index of the document to analyze is stored

$$\rightarrow (\nu\imath,\ell,\ell')\big(\langle\,\ell' \mapsto \texttt{ref ()}\,\rangle \Rsh \langle\,\imath \mapsto \texttt{node}\ \boldsymbol{o}(\imath_1\imath_4)\,\rangle \Rsh \langle\,\ell \mapsto \texttt{try}\ \imath\ p(\ell')\,\rangle \Rsh \ell\big).$$

Note that the "result" of the `try` evaluation is the location ℓ, which will contain `ok` or `fail` at the end of the evaluation. This location can be captured by using a `let` construct, and can be used e.g. in a `wait` expression.

Rule (Red Try Match) is now applied. Let's call S the selector of $p(\ell')$; $\mathtt{a}\ \mathtt{b} \vdash_S p(\ell')\ p(\ell')$, thus two sub-evaluations are started (between documents at $\imath_1$ and $\imath_4$ and $p(\ell')$). The `try` resource at ℓ becomes a `test` resource which waits for the sub-evaluation results

$$\begin{aligned}\to (\nu\imath, \ell, \ell', \ell_1, \ell_4)\big(&\langle\, \ell' \mapsto \mathtt{ref}\ \mathtt{()}\,\rangle \Rsh \langle\, \imath \mapsto \mathtt{node}\, \boldsymbol{o}(\imath_1\imath_4)\,\rangle \Rsh \langle\, \ell_1 \mapsto \mathtt{try}\, \imath_1\, p(\ell')\,\rangle \\ &\Rsh \langle\, \ell_4 \mapsto \mathtt{try}\, \imath_4\, p(\ell')\,\rangle \Rsh \langle\, \ell \mapsto \mathtt{test}\, \imath(\ell_1\ell_4)\,\rangle \Rsh \ell\big).\end{aligned}$$

Sub-evaluations are concurrently started. By rule (Red Try Match) applied twice, two sub-evaluations are triggered, because $\mathtt{b} \vdash_S p(\ell')$ and $\mathtt{a} \vdash_S p(\ell')$

$$\begin{aligned}\to^* (\nu\imath, \ell, \ell', \ell_1, \ell_2, \ell_4, \ell_5)\big(&\langle\, \ell' \mapsto \mathtt{ref}\ \mathtt{()}\,\rangle \Rsh \langle\, \imath \mapsto \mathtt{node}\, \boldsymbol{o}(\imath_1\imath_4)\,\rangle \Rsh \langle\, \ell_2 \mapsto \mathtt{try}\, \imath_2\, p(\ell')\,\rangle \\ &\Rsh \langle\, \ell_5 \mapsto \mathtt{try}\ \mathtt{()}\ p(\ell')\,\rangle \Rsh \langle\, \ell_4 \mapsto \mathtt{test}\, \imath_4(\ell_5)\,\rangle \\ &\Rsh \langle\, \ell_1 \mapsto \mathtt{test}\, \imath_1(\ell_2)\ell'\,\rangle \Rsh \langle\, \ell \mapsto \mathtt{test}\, \imath(\ell_1\ell_4)\,\rangle \Rsh \ell\big).\end{aligned}$$

Note that in location ℓ_1 we take note about the reference ℓ' where the index $\imath_1$ will be stored in case of successful evaluation.

The evaluation at ℓ_5 ends, because the empty document is accepted by p, and there are no triggered sub-evaluations. While evaluation at ℓ_2 continues

$$\begin{aligned}\to^* (\nu\imath, \ell, \ell', \ell_1, \ell_2, \ell_3, \ell_4, \ell_5)\big(&\langle\, \ell' \mapsto \mathtt{ref}\ \mathtt{()}\,\rangle \Rsh \langle\, \imath \mapsto \mathtt{node}\, \boldsymbol{o}(\imath_1\imath_4)\,\rangle \\ &\Rsh \langle\, \ell_3 \mapsto \mathtt{try}\, \imath_3\, p(\ell')\,\rangle \Rsh \langle\, \ell_2 \mapsto \mathtt{test}\, \imath_2(\imath_3)\,\rangle \\ &\Rsh \langle\, \ell_5 \mapsto \mathtt{ok}\ \mathtt{()}\,\rangle \Rsh \langle\, \ell_4 \mapsto \mathtt{test}\, \imath_4(\ell_5)\,\rangle \\ &\Rsh \langle\, \ell_1 \mapsto \mathtt{test}\, \imath_1(\ell_2)\ell'\,\rangle \Rsh \langle\, \ell \mapsto \mathtt{test}\, \imath(\ell_1\ell_4)\,\rangle \Rsh \ell\big).\end{aligned}$$

By (Red Test Ok) evaluation at ℓ_4 ends successfully. Moreover, for evaluation at ℓ_3 we can reason as previously seen for ℓ_4, and obtain a success. Note that $\imath_3$ contains a document tagged `a`, thus by (Red Test Ok) location ℓ' is updated by adding $\imath_3$ to its content

$$\begin{aligned}\to^* (\nu\imath, \ell, \ell', \ell_1, \ell_2, \ell_3, \ell_4, \ell_5)\big(&\langle\, \ell' \mapsto \mathtt{ref}\ (\imath_3)\,\rangle \Rsh \langle\, \imath \mapsto \mathtt{node}\, \boldsymbol{o}(\imath_1\imath_4)\,\rangle \Rsh \langle\, \ell_3 \mapsto \mathtt{ok}\, \imath_3\,\rangle \\ &\Rsh \langle\, \ell_2 \mapsto \mathtt{test}\, \imath_2(\imath_3)\,\rangle \Rsh \langle\, \ell_5 \mapsto \mathtt{ok}\ \mathtt{()}\,\rangle \Rsh \langle\, \ell_4 \mapsto \mathtt{ok}\, \imath_4\,\rangle \\ &\Rsh \langle\, \ell_1 \mapsto \mathtt{test}\, \imath_1(\ell_2)\ell'\,\rangle \Rsh \langle\, \ell \mapsto \mathtt{test}\, \imath(\ell_1\ell_4)\,\rangle \Rsh \ell\big).\end{aligned}$$

By (Red Test Ok) applied twice, evaluation at ℓ_1 ends and location ℓ' is updated:

$$\begin{aligned}\to^* (\nu\imath, \ell, \ell', \ell_1, \ell_2, \ell_3, \ell_4, \ell_5)\big(&\langle\, \ell' \mapsto \mathtt{ref}\ (\imath_3)\,\rangle \Rsh \langle\, \imath \mapsto \mathtt{node}\, \boldsymbol{o}(\imath_1\imath_4)\,\rangle \Rsh \langle\, \ell_3 \mapsto \mathtt{ok}\, \imath_3\,\rangle \\ &\Rsh \langle\, \ell_2 \mapsto \mathtt{ok}\, \imath_2\,\rangle \Rsh \langle\, \ell_5 \mapsto \mathtt{ok}\ \mathtt{()}\,\rangle \Rsh \langle\, \ell_4 \mapsto \mathtt{ok}\, \imath_4\,\rangle \\ &\Rsh \mathtt{let}\ \_ = (\ell' \mathrel{+}= \imath_1)\ \mathtt{in} \\ &\qquad\qquad \langle\, \ell_1 \mapsto \mathtt{ok}\, \imath_1\,\rangle \Rsh \langle\, \ell \mapsto \mathtt{test}\, \imath(\ell_1\ell_4)\,\rangle \Rsh \ell\big).\end{aligned}$$

Finally, the evaluation ends by (Red Write), (Red Let) and (Red Test Ok)

$$\begin{aligned}\to^* (\nu\imath, \ell, \ell', \ell_1, \ell_2, \ell_3, \ell_4)\big(&\langle\, \ell' \mapsto \mathtt{ref}\ (\imath_3\imath_1)\,\rangle \Rsh \langle\, \imath \mapsto \mathtt{node}\, \boldsymbol{o}(\imath_1\imath_4)\,\rangle \Rsh \langle\, \ell_3 \mapsto \mathtt{ok}\, \imath_3\,\rangle \\ &\Rsh \ldots \Rsh \langle\, \ell_1 \mapsto \mathtt{ok}\, \imath_1\,\rangle \Rsh \langle\, \ell \mapsto \mathtt{ok}\, \imath\,\rangle \Rsh \ell\big).\end{aligned}$$

4 Static Semantics

The types of document indexes are the same as 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 $\texttt{ref}\ A$; a node type $\texttt{node}\ \mathtt{a}(u)$ (the type of a location holding an element $\mathtt{a}[u]$); or a try type $\texttt{loc}\ \mathtt{a}(A)$ (the type of locations 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
$\texttt{ref}\ A$	reference
$\texttt{node}\ \mathtt{a}(u)$	node location
$\texttt{loc}\ \mathtt{a}(A)$	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 $\mathtt{a}_{i_1} \dots \mathtt{a}_{i_m}$, denoted $\mathtt{a}_{i_1} \dots \mathtt{a}_{i_m} \vdash_A A_{i_1} \dots A_{i_m}$, if and only if the sequence of tags $\mathtt{a}_{i_1} \dots \mathtt{a}_{i_m}$ 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. (We will not need the type $\overline{A}$ when $A \doteq \texttt{All}$, which means that we do not need to introduce a type with an empty language.) In the case of type witness, we have $\mathtt{a}_{i_1} \dots \mathtt{a}_{i_m} \not\vdash_A$ if and only if there is a witness for $\overline{A}$ of $\mathtt{a}_{i_1} \dots \mathtt{a}_{i_m}$.

The type system is given in Table 2. A type environment E is a finite mapping $x_1 : t_1, \dots, x_n : t_n$ between names and types. The type system is based on a single 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 $A := \mathit{Reg}(\mathtt{a}_i[A_i])_{i\in 1..n}$. We assume that functions and patterns are (explicitly) well-typed, which is denoted $f : \boldsymbol{t} \to t_0$ and $p : \boldsymbol{t} \to A$. The types $t_1, \dots, t_n$ in $\boldsymbol{t}$ 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 $S = \mathit{Reg}(\mathtt{a}_i[p_i(\boldsymbol{x_i})])_{i\in 1..n}$ 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 $p(\boldsymbol{x}) := S\ \texttt{as}\ x_k$ has type $\boldsymbol{t} \to A$, then the type t_k is *compatible* with A, which means that $t_k = \texttt{ref}\ B$ 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 can be of type $\texttt{ref}A$ but not $\texttt{ref}t$). Note that, for every assignment of a value of type B into a reference of type $\texttt{ref}\ A$, rule (Type Write), we check that $A, B <: A$. This is to take into account that references combine the sequence of values that are assigned to them.

The remaining typing rules are for resources and pattern-matching operators. The type of an expression $\texttt{try}\ u\ p(\boldsymbol{v})$ is $\texttt{loc}\ \boldsymbol{o}(A)$ if the pattern p matches documents of

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

(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)
$$\frac{E \vdash \imath_k : \mathtt{node}\, \mathsf{a}_k(u_k) \quad E \vdash u_k : B_k \quad k \in 1..n}{E \vdash \imath_1 \dots \imath_n : \mathsf{a}_1[B_1], \dots, \mathsf{a}_n[B_n]}$$

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

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

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

(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)
$$\frac{E \vdash \ell : \mathtt{node}\, \mathsf{a}(\imath_1 \dots \imath_n)}{E \vdash \langle\, \ell \mapsto \mathtt{node}\, \mathsf{a}(\imath_1 \dots \imath_n) \,\rangle : \star}$$

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

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

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

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

Table 2. Typing Rules

type 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.

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 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 a`(u) has only one possible type, namely `node 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 such a configuration is *well-formed* (see e.g. [12] for an example of how to extend the type system).

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. Since we have a strongly typed language, we need to prove that well-typedness of processes is preserved by reduction. The proof of this theorem is given in Appendix A.

Theorem 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 $P \rightarrow Q$ then $E \vdash Q : t$.*

The proof of Theorem 1 is more involved than in “traditional proofs” for subject reduction. A reason for this is the need to take into account complement types and the fact that it is not possible to reason on a whole document at once (its content is scattered across distinct resource locations.)

We do not state a *progress theorem* in connection with Theorem 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 trivial patterns (i.e. matching expressions that will always fail).

5 Example: the Reverse Web-Link Graph

We study the *reverse web-link graph* application [9], used e.g. in Google's search-engine to compute page ranks. The goal is to build a list of all pages containing a link to a given URL. We consider a calculus enriched with an atomic type for strings and a construct `if` $x = y$ `then`... to test equalities between strings, these extensions are straightforward to accommodate. We assume that web pages in the index are stored as documents of type $WP = \texttt{pg}[B]$, where B is the type $(\texttt{url}[\texttt{String}],\ \texttt{link}[\mathit{URL}*],\ \texttt{text}[\texttt{String}])$ and URL is a shorthand for $\texttt{url}[\texttt{String}]$, meaning that for each page we have its location (url), a list of its hyperlinks (link) and its textual content (text). For simplicity, assume that each list contains no duplicate hyperlinks. The following patterns are used for building a reverse web-link graph:

$$
\begin{array}{lcl}
\mathit{revWL}(t,r) & := & \big(\texttt{pg}[\mathit{revWL}'(t,r)]\big)* \\
\mathit{revWL}'(t,r) & := & \texttt{let}\ x = \texttt{newref ()}\ ,\ y = \texttt{newref ()}\ \texttt{in} \\
 & & \big(\texttt{url}[\texttt{String as}\ x], \texttt{link}[\mathit{URL}*\ \texttt{as}\ y], \texttt{text}[\texttt{String}]\big) \\
 & & \texttt{then}\ \big(\texttt{try}\ !y\ \mathit{sift}(t, !x, r)\big) \\
\mathit{sift}(t,t',r) & := & \big(\texttt{url}[\mathit{sift}'(t,t',r)]\big)* \\
\mathit{sift}'(t,t',r) & := & \texttt{let}\ z = \texttt{newref ()}\ \texttt{in}\ \big(\texttt{String as}\ z\big) \\
 & & \texttt{then}\ \big(\texttt{if}\ z = t\ \texttt{then}\ r \mathrel{+}= \texttt{url}[t']\big)\ .
\end{array}
$$

The main pattern is $\mathit{revWL}(t,r)$, where t is the string representing the target URL, and r is a (global) reference cell for t's reverse-index. revWL visits each indexed page and invokes revWL', which extracts the page's location and list of links, and stores them in two fresh references x and y. Then the pattern sift is used to test whether the list of URL in y contains the target location t. If true, the result r is updated by adding to it the value of x (that is passed as the second parameter of sift). In each pattern, the "location" parameters t and t' have type String while the final result, held in the parameter r, is a reference holding values of type $\mathit{URL}*$. Hence the pattern revWL has type $(\texttt{String}, \texttt{ref}\ (\mathit{URL}*)) \to \mathit{WP}*$ and sift has type $(\texttt{String}, \texttt{String}, \texttt{ref}\ (\mathit{URL}*)) \to \mathit{URL}*$. Assume $\imath_1 \ldots \imath_n$ are the indexes of the web pages of interest, possibly stored in different physical locations, we can create a reverse index for the target location ta with the expression: $\texttt{let}\ z = \texttt{newref ()}\ \texttt{in}\,\texttt{try}\ (\imath_1 \ldots \imath_n)\ \mathit{revWL}(ta, z)$. Note that patterns and functions are evaluated locally at each site, while the result reference z is "global" (it is local to the caller, but is accessed by every site for storing the results.)

6 Extensions

We study how to interpret two interesting programming idioms in our model: spawning an expression in a new thread, and handling user-defined exceptions.

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 $e_1 \upharpoonright e_2$. 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 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:

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

In the resulting process, e_1 and e_2 are evaluated concurrently and the resource $\langle\, \ell \mapsto \mathtt{ok}\ \imath\,\rangle$ cannot interact with e_2 until the evaluation of e_1 ends. Hence we can use the expression $(\mathtt{wait}\ x(y)\ \mathtt{then}\ e)$ in e_2 to block the execution 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.) 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 $(\langle\, \imath \mapsto \mathtt{node}\ \mathtt{a}(u)\,\rangle \Rsh \langle\, \ell \mapsto \mathtt{ok}\ \imath\,\rangle)$ to an (asynchronous) output action $\ell!\langle \mathtt{ok}, u\rangle$ as found in process calculi such as the π-calculus. Similarly, we can compare an expression $\mathtt{wait}\ \ell(x)\ \mathtt{then}\ e_1\ \mathtt{else}\ e_2$ with an input action.

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, see Section 2) contains only women. This can be achieved using the pattern declarations $p(\,) := \mathtt{woman}[q(\,)]*$ and $q(\,) := \mathtt{name}[\mathtt{All}], \mathtt{d}[p(\,)], \mathtt{s}[\mathtt{Empty}]$ and a matching expression $\mathtt{try}\ u\ p(\,)$. 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 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 $\langle\, \imath_e \mapsto \mathtt{node}\ \boldsymbol{o}(\,)\,\rangle$:

$\mathtt{exception}$	$=$	$(\nu\ell)\ell$	creates a fresh (location) exception
$\mathtt{throw}\ \ell$	$=$	$\langle\, \ell \mapsto \mathtt{ok}\ \imath_e\,\rangle \Rsh \mathtt{()}$	raises an exception at ℓ
$\mathtt{catch}\ \ell\ e$	$=$	$\mathtt{wait}\ \ell(x)\ \mathtt{then}\ e$	catches exception ℓ and runs e $(x \notin fv(e))$.

For instance, it is possible to raise the exception in the compensation part of a pattern declaration, to catch this exception and avoid to wait the end of the pattern-matching evaluation. E.g. the pattern p above can be redefined in: $p'(x) := \mathtt{woman}[q(\,)]*\ \mathtt{else}\ \mathtt{throw}\ x$.

7 Conclusions and related work

We study a formal model for computing over large (even dynamic) distributed XML documents. We extend the functional approach taken in e.g. XDuce and define a typed process calculus which supports a first-order type system with subtyping based on regular expression types, a system compatible with DTD and other schema languages for XML.

This work may be compared with recent proposals for integrating XML data into π-calculus, where pattern-matching plays a fundamental role: Iota [5] 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 [6] 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.

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. To this end, we will also need to answer questions concerning observational equivalences that we intend to study in future work. Our approach 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]. Although, for this, it will be necessary to add an "`eval`/`quote`" mechanism, as in e.g. LISP, and to revise our static type checking approach. Finally, another avenue to investigate is the encoding of other concurrency primitives, especially channel-based synchronization and distributed transactions.

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 29, 2006.
4. Amadio R.: An Asynchronous Model of Locality, Failure And Process Mobility. In *Proc. of COORDINATION*, LNCS vol. 1282, Springer, 1997.
5. Bierman G., Sewell P.: Iota: A concurrent XML scripting language with applications to Home Area Networking. TR 577, Computer Lab., Cambridge, 2003.
6. 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.
7. Castagna G.: Pattern and types for querying XML documents. In *Proc. of DBPL*, XSYM 2005 joint keynote talk, 2005.
8. 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/`
9. Dean J., Ghemawat, S.: MapReduce: Simplified Data Processing on Large Cluster. In *Proc. of OSDI*, 2004.
10. Ferreira W., Hennessy M., Jeffrey A.S.: A theory of weak bisimulation for core CML. J. Functional Programming 8(5), 1998.
11. Gardner P., Maffeis S.: Modelling dynamic web data. Theor. Comput. Sci. 342(1) (2005).
12. 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.
13. Hosoya H., Vouillon J., Pierce B.J.: Regular expression types for XML. ACM Transactions on Programming Languages and Systems, 27(1), 2004.
14. Hosoya H., Pierce B.J.: Regular expression pattern matching for XML. In *Proc. of POPL*, 2001.
15. Hosoya H., Pierce B.J.: XDuce: A Statically Typed XML Processing Language. In *Proc. of ACM Transaction on Internet Technology*, 2003.
16. Milner R.: Communicating and Mobile Systems: The π-Calculus. CUP , 1999.

A Proof of Theorem 1

We start by introducing some few preliminary results.

Proposition 1 (weakening). *If $E, x{:}t \vdash P : t'$ and $x \notin fn(P)$ then $E \vdash P : t'$ and vice versa.*

Proposition 2. *Assume $S = Reg(\mathtt{a}_i[p_i(\boldsymbol{v_i})])_{i \in 1..k}$ is a unambiguous pattern with type A. If $\mathtt{a}_1 \dots \mathtt{a}_n \vdash_S p_1(\boldsymbol{v_1}) \dots p_n(\boldsymbol{v_n})$ then we also have $\mathtt{a}_1 \dots \mathtt{a}_n \vdash_A A_1 \dots A_n$.*

Proposition 3. *Assume A is a an unambiguous type. If $\mathtt{a}_1 \dots \mathtt{a_n} \vdash_A A_1 \dots A_n$ then $\mathtt{a}_1[A_1], \dots, \mathtt{a}_n[A_n] <: A$ and if $\mathtt{a}_1 \dots \mathtt{a_n} \not\vdash_A$ then there is no $B_1, \dots, B_n$ such that $\mathtt{a}_1[B_1], \dots, \mathtt{a}_n[B_n] <: A$.*

Proposition 4. *Suppose A unambiguous and $A \neq \underline{\ }\mathtt{All}$. $\mathtt{a}_1 \dots \mathtt{a_j} \dots \mathtt{a}_n \vdash_A A_1 \dots A_j \dots A_n \Rightarrow \mathtt{a}_1[A_1], \dots, \mathtt{a_j}[\overline{A_j}], \dots, \mathtt{a}_n[A_n] <: \overline{A}$.*

Proposition 5. *Suppose $E \vdash S : A$ and $u = \imath_1 \dots \imath_n$ with $E \vdash \imath_i : \mathtt{node}\ \mathtt{a_i}(u_i)$ for $i \in 1, \dots, n$. If $\mathtt{a}_1 \dots \mathtt{a}_n \not\vdash_S$ then $E \vdash u : \overline{A}$.*

Theorem 1 *Suppose that P is well formed and contains only unambiguous patterns and t contains only unambiguous types. If $E \vdash P : t$ and $P \rightarrow Q$ then $E \vdash Q : t$.*

Proof. By induction on reduction rules. We distinguish the last rule applied for deducing $P \rightarrow Q$ (remember that at every step we work with a well formed term). The most interesting cases are the following:

(Red Try Match) by rules (Type Par), (Type Try Doc), and (Type Loc Node) $E \vdash \prod_{k \in 1 \dots n} \langle\, \imath_k \mapsto \mathtt{node}\ \mathtt{a_k}(w_k) \,\rangle \Rsh \langle\, \imath \mapsto \mathtt{node}\ \mathtt{a}(\imath_1 \dots \imath_n) \,\rangle \Rsh \langle\, \ell \mapsto \mathtt{try}\ \imath\, p(\boldsymbol{v}) \,\rangle : \star$ implies:

- $E \vdash \imath : \mathtt{node}\ \mathtt{a}(\imath_1 \dots \imath_n)$;
- $E \vdash \imath_k : \mathtt{node}\ \mathtt{a_k}(w_k)$ and $w_k = (\imath_{1_k} \dots \imath_{n_k})$;
- $E \vdash \ell : \mathtt{loc}\ \mathtt{a}(A)$, $p : (\boldsymbol{t}) \rightarrow A$, and $E \vdash \boldsymbol{v} : \boldsymbol{t}$; thus if $p(\boldsymbol{x})$ **:=** S $\mathtt{as}$ x_k then $S : A$, $E \vdash v_k : t_k$ and t_k is compatible with A, that is $t_k = \mathtt{ref}\ B$ and $B, A <: B$ (see page 11).

By (Red Try Match)

$$\begin{gathered}\textstyle\prod_{k \in 1 \dots n} \langle\, \imath_k \mapsto \mathtt{node}\ \mathtt{a_k}(w_k) \,\rangle \Rsh \langle\, \imath \mapsto \mathtt{node}\ \mathtt{a}(\imath_1 \dots \imath_n) \,\rangle \Rsh \langle\, \ell \mapsto \mathtt{try}\ \imath\, p(\boldsymbol{v}) \,\rangle \rightarrow \\ \textstyle\prod_{k \in 1 \dots n} \langle\, \imath_k \mapsto \mathtt{node}\ \mathtt{a_k}(w_k) \,\rangle \Rsh \langle\, \imath \mapsto \mathtt{node}\ \mathtt{a}(\imath_1 \dots \imath_n) \,\rangle \Rsh \\ (\nu w)(\textstyle\prod \langle\, \jmath_k \mapsto \mathtt{try}\ \imath_k\, p_k(\boldsymbol{v_k}) \,\rangle \Rsh \langle\, \ell \mapsto \mathtt{test}\ \imath\ w\ v_k \,\rangle)\end{gathered}$$

implies $w = \jmath_1 \dots \jmath_n$ fresh and $\mathtt{a}_1 \dots \mathtt{a}_n \vdash_S p_1(\boldsymbol{v_1}) \dots p_n(\boldsymbol{v_n})$.
If $p_k : (\boldsymbol{t_k}) \rightarrow A_k$ we choose $j_k : \mathtt{loc}\ \mathtt{a_k}(A_k)$ and $E, j_k : \mathtt{loc}\ \mathtt{a_k}(A_k)_{k=1,\dots,n} \vdash \prod_k \langle\, \jmath_k \mapsto \mathtt{try}\ \imath_k\, p_k(\boldsymbol{v_k}) \,\rangle : \star$.
We have to show that $E \vdash \langle\, \ell \mapsto \mathtt{test}\ \imath\ w\ v_k \,\rangle : \star$. We know that:

- $E \vdash \ell : \mathtt{loc}\ \mathtt{a}(A)$;
- $E \vdash \imath : \mathtt{node}\ \mathtt{a}(\imath_1 \dots \imath_n)$;
- $j_k : \mathtt{loc}\ \mathtt{a}_k(A_k)_{k=1,\dots,n}$;
- $E \vdash v_k : t_k$, $t_k = \mathtt{ref}\ B$ and $B, A <: B$.

We have to prove that $\mathtt{a}_1 \ldots \mathtt{a}_n \vdash_A A_1 \ldots A_n$. By the reduction we have $\mathtt{a}_1 \ldots \mathtt{a}_\mathtt{n} \vdash_S p_1(\boldsymbol{v_1}) \ldots p_n(\boldsymbol{v_n})$; moreover $p_i(\boldsymbol{v_i}) : A_i$, and $S : A$, so by Proposition 2 $\mathtt{a}_1 \ldots \mathtt{a}_\mathtt{n} \vdash_A A_1 \ldots A_n$, thus $E \vdash \langle\, \ell \mapsto \mathtt{test}\, \imath\, w\, v_k \,\rangle : \star$. In conclusion $E \vdash (\nu w)(\prod\langle\, \jmath_k \mapsto \mathtt{try}\, \imath_k\, p_k(\boldsymbol{v_k}) \,\rangle \Rsh \langle\, \ell \mapsto \mathtt{test}\, \imath\, w\, v_k \,\rangle) : \star$;

(Red Try Error) $E \vdash \prod_{k\in 1,\ldots,n}\langle\, \imath_k \mapsto \mathtt{node}\, \mathtt{a}_\mathtt{k}(v_k) \,\rangle \Rsh \langle\, \imath \mapsto \mathtt{node}\, \mathtt{a}(\imath_1 \ldots \imath_n \,\rangle \Rsh$ $\langle\, \ell \mapsto \mathtt{try}\, \imath\, p(\boldsymbol{v}) \,\rangle : \star$ implies $E \vdash \imath : \mathtt{node}\, \mathtt{a}(\imath_1 \ldots \imath_n)$, $E \vdash \imath_k : \mathtt{node}\, \mathtt{a}_\mathtt{k}(v_k)$, $v_k = (\imath_{1_k} \ldots \imath_{n_k})$ $E \vdash \ell : \mathtt{loc}\, \mathtt{a}(A)$, $p : (\boldsymbol{t}) \to A$, and $E \vdash \boldsymbol{v} : \boldsymbol{t}$. By the reduction $p(\boldsymbol{v}) \mathrel{\boldsymbol{:=}} S$ $\mathtt{as}$ v_k, thus $E \vdash S : A$. $\mathtt{a}_1 \ldots \mathtt{a}_n \not\vdash_S$, thus, by Proposition 5, $E \vdash \imath_1 \ldots \imath_n : \overline{A}$ so, by (Type Let) and (Type Loc Fail), $E \vdash \langle\, \ell \mapsto \mathtt{fail}\, \imath \,\rangle : \star$;

(Red Test Ok) by rule (Type Loc Ok), (Type Loc Node), and (Type Test Loc) $E \vdash \langle\, \imath \mapsto \mathtt{node}\, \mathtt{a}(\imath_1 \ldots \imath_n) \,\rangle \Rsh \prod_{k\in 1,\ldots,n}\langle\, \jmath_k \mapsto \mathtt{ok}\, \imath_k \,\rangle \Rsh \langle\, \ell \mapsto \mathtt{test}\, \imath\, w\, v_k \,\rangle : \star$ (where $w = \jmath_1 \ldots \jmath_n$) implies $E \vdash \imath : \mathtt{node}\, \mathtt{a}(\imath_1 \ldots \imath_n)$, $\forall k \in 1, \ldots, n : E \vdash \jmath_k : \mathtt{loc}\, \mathtt{a}_k(A_k)$, $E \vdash \imath_k : \mathtt{node}\, \mathtt{a}_\mathtt{k}(u_k)$, and $E \vdash u_k : A_k$. Moreover $E \vdash \ell : \mathtt{loc}\, \mathtt{a}(A)$, $\mathtt{a}_1 \ldots \mathtt{a}_n \vdash_A A_1 \ldots A_n$, $E \vdash v_k : t_k$, $t_k = \mathtt{ref}\ B$ and $B, A <: B$.
$\mathtt{a}_1 \ldots \mathtt{a}_n \vdash_A A_1 \ldots A_n$ implies $\mathtt{a}_1[A_1], \ldots, \mathtt{a}_n[A_n] <: A$, by Proposition 3; thus by (Type Doc) $E \vdash \imath_k : \mathtt{node}\, \mathtt{a}_\mathtt{k}(u_k)$, and $E \vdash u_k : A_k$ we have $E \vdash \imath_1 \ldots \imath_n : \mathtt{a}_1[A_1], \ldots, \mathtt{a}_\mathtt{n}[A_n]$ and by (Type Sub) $E \vdash \imath_1 \ldots \imath_n : A$.
By rule (Type Write), $E \vdash \imath_1 \ldots \imath_n : A$, $E \vdash v_k : t_k$, $t_k = \mathtt{ref}\ B$ and $B, A <: B$ imply that $E \vdash v_k \mathrel{+}= (\imath_1 \ldots \imath_n) : \mathtt{Empty}$; x fresh name and Proposition 1 imply that $E \vdash \mathtt{let}\ x = (v_k \mathrel{+}= (\imath_1 \ldots \imath_n))\ \mathtt{in}\ \langle\, \ell \mapsto \mathtt{ok}\, \imath \,\rangle : \star$. Finally, by (Type Par), $E \vdash \langle\, \imath \mapsto \mathtt{node}\, \mathtt{a}(\imath_1 \ldots \imath_n) \,\rangle \Rsh \prod_{k\in 1,\ldots,n}\langle\, \jmath_k \mapsto \mathtt{ok}\, \imath_k \,\rangle \Rsh\ \mathtt{let}\ x = (v_k \mathrel{+}= (\imath_1 \ldots \imath_n))\ \mathtt{in}\ \langle\, \ell \mapsto \mathtt{ok}\, \imath \,\rangle : \star$;

(Red Test Fail) We have $E \vdash \langle\, \imath \mapsto \mathtt{node}\, \mathtt{a}(\imath_1 \ldots \imath_n) \,\rangle \Rsh \prod_{k\in 1,\ldots,n}\langle\, \jmath_k \mapsto d_k \,\rangle \Rsh$ $\langle\, \ell \mapsto \mathtt{test}\, \imath\, w\, v_k \,\rangle : \star$ (with $w = \jmath_1 \ldots \jmath_n$) implies:

- by rule (Type Loc Node) $E \vdash \imath : \mathtt{node}\, \mathtt{a}(\imath_1 \ldots \imath_n)$;
- by rule (Type Loc Ok) $\forall k \in 1, \ldots, n$: s.t. $d_k = \mathtt{ok}\, \imath_k$ we have $E \vdash \jmath_k : \mathtt{loc}\, \mathtt{a}_k(A_k)$, $E \vdash \imath_k : \mathtt{node}\, \mathtt{a}_\mathtt{k}(v_k)$, and $E \vdash v_k : A_k$;
- by rule (Type Loc Fail) $\forall k \in 1, \ldots, n$: s.t. $d_k = \mathtt{fail}\, \imath_k$ we have $E \vdash \jmath_k : \mathtt{loc}\, \mathtt{a}_\mathtt{k}(A_k)$, $E \vdash \imath_k : \mathtt{node}\, \mathtt{a}_\mathtt{k}(v_k)$, and $E \vdash v_k : \overline{A_k}$;
- by rule (Type Test Loc) $E \vdash \ell : \mathtt{loc}\, \mathtt{a}(A)$, $E \vdash \imath : \mathtt{node}\, \mathtt{a}(\imath_1 \ldots \imath_n)$, $E \vdash \jmath_k : \mathtt{loc}\, \mathtt{a}_k(A_k)$, $\mathtt{a}_1 \ldots \mathtt{a}_n \vdash_A A_1 \ldots A_n$ and $E \vdash t_k$, $t_k = \mathtt{ref}\ B$ and $B, A <: B$.

By (Red Test Fail) $\langle\, \imath \mapsto \mathtt{node}\, \mathtt{a}(\imath_1 \ldots \imath_n) \,\rangle \Rsh \prod_{k\in 1,\ldots,n}\langle\, \jmath_k \mapsto d_k \,\rangle \Rsh$
$\langle\, \ell \mapsto \mathtt{test}\, \imath\, w\, v_k \,\rangle \to \langle\, \imath \mapsto \mathtt{node}\, \mathtt{a}(\imath_1 \ldots \imath_n) \,\rangle \Rsh \prod_{k\in 1,\ldots,n}\langle\, \jmath_k \mapsto d_k \,\rangle \Rsh \langle\, \ell \mapsto \mathtt{fail}\, \imath \,\rangle$
if $\exists j \in 1, \ldots, n :$ $\langle\, \jmath_j \mapsto \mathtt{fail}\, \imath_j \,\rangle$ (note that for j we have $E \vdash v_j : \overline{A_j}$). Obviously $A \neq \mathtt{All}$ (because $\mathtt{All} : \mathtt{All}$ and every document satisfies the match with $\mathtt{All}$), so by Proposition 4 $\mathtt{a}_1[A_1], \ldots, \mathtt{a}_j[\overline{A_j}], \ldots, \mathtt{a}_n[A_n] <: \overline{A}$. By rule (Type Doc) $E \vdash \imath_1 \ldots \imath_n : \mathtt{a}_1[A_1], \ldots, \mathtt{a}_\mathtt{j}[\overline{A_j}], \ldots, \mathtt{a}_n[A_n]$ and by (Type Sub) $E \vdash \imath_1 \ldots \imath_n : \overline{A}$. In conclusion, by (Type Loc Fail), $E \vdash \langle\, \ell \mapsto \mathtt{fail}\, \imath \,\rangle : \star$. $\square$