Tree Components Programming: an Application to XML

Pascal Coupey, Christophe Fouqueré, and Jean-Vincent Loddo

LIPN – UMR7030
CNRS – Université Paris 13
99 av. J-B Clément, F–93430 Villetaneuse, France
firstname.lastname@lipn.univ-paris13.fr

Abstract. We present a new programming approach based on a contextual component specification. The language we propose integrates XML and functional aspects in a coherent and homogeneous framework. This enables us to fully have static typing and to specify formal properties with respect to interactions.
Our language FICX, Functional Interactive and Compositional XML, defines a new kind of data structure called Xobjects and relies on a statically typed functional language (currently OCaml). An Xobject is an abstract structure made in two parts: the Xdata part is an XML structure extended by means of triggers dedicated to interactions, the reaction part gives the code associated to triggers that is evaluated on demand. The modularity is ensured by a parameterization of Xobjects: compound Xobjects form a tree structure, rendering a complex XML tree together with appropriate reactions for triggers. A program is a set of structures, each structure being a tree of Xobjects.

1 Introduction

Classic object oriented programming languages offer class/subclass relationship with inheritance mechanism. It is not well suited when applications need "part-of" relationship. Of course, this may be encoded using the object paradigm but no facility is given to the programmer since she has to build by herself the partonomy beside the class/-subclass hierarchy. This is true in the semi-structured data field and in particular XML-like languages where many recent works extend XML language in order to describe documents as a composition of various parts (pure XML, scripts, database requests, web service requests [11, 13, 5]). Our purpose is to propose a programming language whose core principle is that basic objects are components of a tree and to apply it to XML language. In fact, tree structures may be obtained by merging partial trees (instead of just composing them). Such (partially defined) structures are first-class citizens in our programming language. They encapsulate static and dynamic contents to allow for interactivity and expressiveness. Moreover the whole language is strongly typed to ensure error-free executions. This programming paradigm is applied here to XML. Examples are given wrt the web as this domain has at least the following features: use of semi-structured data, interactivity, needs for modular and safe programming. Our language FICX, Functional Interactive and Compositional XML, defines a new kind of

data structure called *Xobject* and relies on a statically typed functional language (currently OCaml). An Xobject is an abstract structure made of two parts: the *Xdata* part is an XML structure extended by means of triggers dedicated to interactions, the *reaction* part gives the code associated to triggers and that is evaluated on request. A *request* is a first-citizen expression of the language. Its value is the result of a reaction selected by a trigger. To take advantage of the tree structure of compound Xobjects, a delegation mechanism is offered: a request may contain an (abstract) path to be followed to find an adequate reaction. FICX uses in fact extensively the concept of *abstract path*. An abstract path is a sequence of labels for addressing Xobjects in a tree, e.g. the root, the value at label Y of a parent, ... To summarize, FICX has the main following characteristics:

Modularity: Compound Xobjects form a tree structure, rendering a complex XML tree together with appropriate reactions for triggers. A program is a set of *Xstructures*. Each Xstructure is a fully defined tree of Xobjects and plays the role of an entry point to the program.

Interaction: Each reaction describes a possible evolution: the result of requesting a reaction to some Xobject is a new Xobject (possibly with new Xdata, new triggers, new reactions).

Static typing: The type of an Xobject is given by the type of its Xdata part together with the type of its reactions. An Xobject defines a set of triggers (usable for interaction) and a set of reactions (called either by an expression of the language or by means of an interaction). These two sets should coincide in case of Xstructures: the type of reaction patterns should cover the type of the XML structure associated to a trigger (soundness), and, a reaction being given, a corresponding trigger should have been defined for interaction (completeness).

These peculiarities offer the user means to develop modular and type checked programs. In the framework of web applications, triggers may be viewed as web service names or anchors in web sites. However, contrarily to most web languages, triggers and reactions should be related in a program in such a way that controls may occur. The toy example given in example 1 on the left defines the variable `link` to be a function with one parameter `msg` which returns an Xobject[1] and `home` whose value is an Xobject with one parameter `Y`. The Xdata part is written in CDuce style [2] and is extended with a trigger `T`. The corresponding XML structure in `home` Xobject declaration is given on the right. The Xobject `link` has two reactions with trigger tag `T`. The first one creates a new Xobject `link` with the string `"Bonjour"` if the parameter given with the trigger contains the string `"Hello"`, the second reaction has the converse behaviour. The evolution consists in creating Xobjects that alternates `"Bonjour"` and `"Hello"` messages. `website` is a (compound) Xobject giving a value to the parameter `Y` in a copy of Xobject `home`. Note that Xobject `home` is unchanged. Its standard HTML presentation is given on the right (where $URL\_encoding\_of()$ is a built-in function). The interactive request has the same shape as the expression for defining `otherwebsite`. Its operational semantics uses the delegation mechanism. In `Y.T("Hello")@website`, `website` is called the *initial concrete*

[1] For the sake of simplicity, we consider that `link = xobject <> (msg:string) ...` is syntactic sugar for `link = fun (msg:string)→xobject <> ...` then collapsing the name of the function which returns the Xobject with the name of the Xobject itself.

```
link = fun (msg:string) →
   xobject <>
      T<h1 align="center">[msg]
   ▶
      T(<h1 align="center">["Hello"])
         ⇒ (link "Bonjour")
      T(<h1 align="center">["Bonjour"])
         ⇒ (link "Hello")
   xend;

home = xobject <Y>
      <html>[
         <head>[<title>["Welcome"]]
         <body>[Y]
      ]
   ▶ xend;

website = home[Y ↦ (link "Hello")];
otherwebsite = Y.T("Hello")@website;
```

Data extracted from `website` in standard XML style:

```
<html>
   <head>
      <title>"Welcome"</title>
   </head>
   <body>
      <a href=
         URL_encoding_of(
            Y.T("Hello")@website
         )>
      <h1 align="center">["Hello"]
      </a>
   </body>
</html>
```

Ex. 1: Xobject definitions and XML data

receiver as it is the Xobject that should at first react. As it has no appropriate reaction, the request is delegated wrt the path, here `Y`. Let o be the value of `(link "Hello")`, o responds by `(link "Bonjour")`. The Xobject `otherwebsite` may then have been defined equivalently by the expression `home[Y↦(link "Bonjour")]`. This is also the result sent back in case of interactive request.

We present in the next section the syntax and the operational semantics of the language FICX, focussing on its main features: abstract paths, Xobjects, requests. We define in particular a specific class of trees and show in which extent a set of abstract paths is a representation of such a tree. We give in section 3 the type system . We end comparing FICX to other works in this domain and present a few extensions under study.

2 Language FICX: syntax and operational semantics

We use a functional programming language, currently OCaml, as a core language for functions, definitions, ... that we de not detail here (the reader may find descriptions of OCaml in [6]). This core is extended by means of an *Xobject* data type that integrates an extended XML structure called *Xdata* to publish data and triggers, and a functional part called *reaction* intended to answer requests built from triggers. Moreover Xobjects may be parameterized by *abstract paths* defined in the following subsection. Finally, an *Xstructure* is a specific top-level definition that is used to declare interactive data and functionalities. The grammar of the language FICX, specific to our aim, is given in figure 1. We use the following notations throughout the paper: e is an expression and p is a pattern, a, A, C, x, y are identifiers, τ is a trigger, Y, Z are abstract paths, finally r states for a reaction.

The operational semantics follows standard functional programming operational semantics: it is given as an evaluation judgment on programs, expressions, ... to be computed with respect to a given environment. An environment is an *evaluation environ-*

Program

P	$::= \epsilon$		*empty program*
	$\mid S\ P \mid d\ P$		*Xstructure or definition followed by a program*

Xstructure

S	$::= \texttt{xstruc}\ d\ \texttt{begin}\ w = e$	*where d is a definition, w an identifier, e an expression*

Xobject

e	$::= \texttt{xobject}\langle Y_1, \ldots, Y_n \rangle$	*Xobject definition with abstract paths parameters*
	$\quad e \blacktriangleright sr$	*Xdata ► reactions*
	$\quad \texttt{xend}$	
	$\mid e_1[Y \mapsto e_2]$	*parameter assignment*
	$\mid \tau(e_2)@e_1$	*request evaluation*
	$\mid Y$	*Abstract path name Y*

Reactions

sr	$::= \epsilon$	r	$::= \tau_p(p) \Rightarrow e$	*reaction conditioned by trigger and parameter patterns*
	$\mid r\ sr$			

Triggers

τ	$::= Y.C$	*abstract path followed by a tag*

Fig. 1: Grammar of FICX

ment together with a *handler environment*. An evaluation environment is a partial function from the set of variable names and abstract paths to values, either ground values or handlers to such values (supposing a domain of handlers). A handler environment is a partial function from the set of handlers to values. Handlers are used to denote Xobject parameter values. The evaluation judgment for expressions is of the following form:

$$\mathcal{E}, \mathcal{H} \vdash e \Downarrow v, \mathcal{H}'$$

read as: the evaluation of expression e in an evaluation environment $\mathcal{E}$ with a handler environment $\mathcal{H}$ leads to a value v together with a new handler environment $\mathcal{H}'$.

2.1 *Abstract paths*

Abstract paths are defined according to the following grammar, where y is an identifier, `parent`, `root` and `self` are keywords:

$$\text{Y} ::= \texttt{parent} \mid \texttt{root} \mid \texttt{self} \mid \text{y} \mid \text{Y.Y}$$

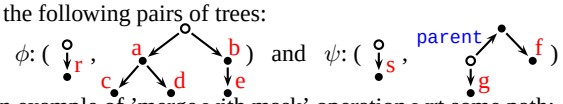
We suppose further that abstract paths (and abstract path patterns or path types in the same way[2]) are always in normal form with respect to the rewriting $\rightarrow_{AP}$ applied to `self`.Y, where Y is the abstract path to be normalized ($y \neq$ `self`, `root`, `parent`):

$$\text{y.}\texttt{parent} \rightarrow_{AP} \epsilon \qquad \text{Y.}\texttt{root} \rightarrow_{AP} \texttt{root} \qquad \text{Y.}\texttt{self} \rightarrow_{AP} \texttt{self}$$

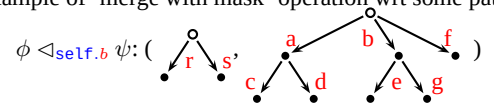
Thus $\rightarrow_{AP}$ gives rise to two kinds of normal forms: `root`.Y and `self`.Y with `self` and `root` not in Y. The intended meaning is that a set of such abstract paths should partially define two rooted trees, one with abstract root `root` and another 'centered' on `self` where `self.parent`....`parent` should represent a path 'up' to some concrete root (see example 2 where orientation is given as the root is not always at top). Abusively, `self` may be omitted in the following from abstract paths writings. We give

[2] Grammars for path patterns and path types are similar to the grammar given for abstract paths, except a '_' added for patterns. '_' matches an abstract path.

Let $\phi = \{\texttt{root}.r, \texttt{self}.a.c, \texttt{self}.a.d, \texttt{self}.b.e\}$
and $\psi = \{\texttt{root}.s, \texttt{self}.g, \texttt{self.parent}.f\}$,
ϕ and ψ represent the following pairs of trees:



The following is an example of 'merge with mask' operation wrt some path:



Ex. 2: Abstract paths and trees.

below a few simple definitions and properties that characterize the particular trees and operations we need. We then relate such trees to a specification given by abstract paths. We do not first consider values (say Xdata and reactions) attached to nodes and we fix a non-empty set of symbols $\mathcal{L}$.

Definition 1.

- *An* unambiguous rooted $\mathcal{L}$-edge-labelled tree *is a tree with a root node, edges labelled by elements in $\mathcal{L}$, and such that for each node two distinct edges have distinct labels. Let $\mathcal{T}_\mathcal{L}$ be the set of such trees.*
- *Let $T_1, T_2 \in \mathcal{T}_\mathcal{L}$, $T_1 \leq T_2$ if there exists an injective mapping f from T_1 to T_2 such that if r is the root node of T_1 then $f(r)$ is the root node of T_2, and if (n_1, n_2) is an edge of T_1 labelled l, then $(f(n_1), f(n_2))$ is an edge of T_2 labelled l.*

Proposition 1. *Let $\mathcal{L}^*$ be the set of finite sequences of elements of $\mathcal{L}$ and $\mathcal{P}(\mathcal{L}^*)$ be the powerset of $\mathcal{L}^*$, $(\mathcal{T}_\mathcal{L}, \leq)$ is faithfully represented by $(\mathcal{P}(\mathcal{L}^*), \subset)$, hence $(\mathcal{T}_\mathcal{L}, \leq)$ is a lattice.*

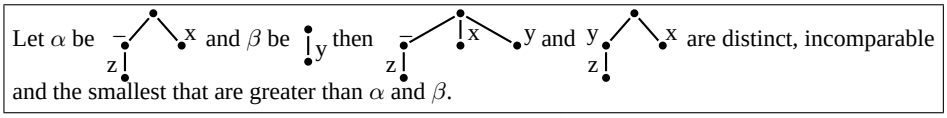
Let us now consider the set $\mathcal{T}_{\bar{\mathcal{L}}}$ of *partially defined unambiguous rooted $\mathcal{L}$-edge-labelled trees*, where a special symbol '_' plays the role of a variable:

Definition 2. *Let _ be a symbol not in $\mathcal{L}$, $\mathcal{T}_{\bar{\mathcal{L}}}$ is the set of unambiguous rooted $\mathcal{L} \cup \{\_\}$-edge-labelled trees such that paths $Y.l.\_$ cannot appear, where $l \in \mathcal{L}$. Let $T_1, T_2 \in \mathcal{T}_{\bar{\mathcal{L}}}$, $T_1 \leq T_2$ if there exists an injective mapping f from T_1 to T_2 such that if r is the root node of T_1 then $f(r)$ is the root node of T_2, if (n_1, n_2) is an edge of T_1, then $(f(n_1), f(n_2))$ is an edge of T_2, moreover if the label of (n_1, n_2) is in $\mathcal{L}$ then it is also the label of $(f(n_1), f(n_2))$.*

Proposition 2. *$(\mathcal{T}_{\bar{\mathcal{L}}}, \leq)$ is a poset, $(\mathcal{T}_\mathcal{L}, \leq)$ embeds in $(\mathcal{T}_{\bar{\mathcal{L}}}, \leq)$, $(\mathcal{T}_{\bar{\mathcal{L}}}, \leq)$ is not a lattice.*

In example 3, a counter-example for $(\mathcal{T}_{\bar{\mathcal{L}}}, \leq)$ to be a lattice is given. However the following proposition serves to control Xobject composition validity:

Proposition 3. *Let $T_1, T_2 \in \mathcal{T}_{\bar{\mathcal{L}}}$, (T_1, T_2) has a least upper bound (lub) iff either $T_1 = T_2$, or $T_1, T_2 \in \mathcal{T}_\mathcal{L}$, or if $T_1 \notin \mathcal{T}_\mathcal{L}$ then $T_2 \in \mathcal{T}_\mathcal{L}$ and the set of labels of edges from the root of T_2 is included in the set of labels of edges from the root of T_1, or the same last property interchanging T_1 and T_2.*



Ex. 3: Trees in $\mathcal{T}_{\bar{\mathcal{L}}}$

The proof follows a structural definition of these trees. The key element comes from the fact that if the two roots have each a daughter labelled _ then there are always at least two distinct minima: one merging the two edges labelled _, the other putting separately two branches, one labelled _, the other labelled by some element of $\mathcal{L}$ (that is a non-empty set).

Let us now add a value to a node, i.e. a value necessary for the operational semantics or the typing system. For simplicity, we keep notation $\mathcal{T}_{\bar{\mathcal{L}}}$ and we define a lub of two valued trees as the lub of the trees obtained forgetting the values. We consider moreover the following definitions: If $T \in \mathcal{T}_{\bar{\mathcal{L}}}$ and Z is a path in T then $Z(T)$ is the subtree of T at path Z, $val(T)$ is the value associated to the root node of T, $self(T)$ is defined as the subtree ending the _ path: $self(T) = T$ if $T \in \mathcal{T}_{\mathcal{L}}$, $self(T) = self(\_(T))$ otherwise. Taking care of previous properties, we define the following partial operations on $\mathcal{T}_{\bar{\mathcal{L}}}$:

Definition 3. *Let $T_1, T_2 \in \mathcal{T}_{\bar{\mathcal{L}}}$,*

- *'Merge with mask' operation $\lhd$: $T = T_1 \lhd T_2$ has the structure of the lub of T_1 and T_2 if it exists and, forall path Y, $val(Y(T)) = val(Y(T_2))$ if defined, $val(Y(T)) = val(Y(T_1))$ otherwise.*
- *'Merge with mask' operation wrt some path. $T = T_1 \lhd_Y T_2$ is defined in the following way: let n be the longest path including only _ in T_2 (i.e. the depth of $self(T_2)$ in T_2),*
  - *if $|Y| \geq n$, let Z be the prefix of Y of length $|Y| - n$ and T_1' be the tree T_1 after deleting the subtree $Z(T_1)$, if $Z(T_1) \lhd T_2$ exists, T is the tree T_1' appending $Z(T) \lhd T_2$ at the leaf Z, otherwise T is not defined ;*
  - *if $|Y| < n$, let Z be a sequence of _ of length $n - |Y|$ and T_2' be the tree T_2 after deleting the subtree $Z(T_2)$, if $T_1 \lhd Z(T_2)$ exists, T is the tree T_2' appending $T_1 \lhd Z(T_2)$ at the leaf Z, otherwise T is not defined.*

Note that, thanks to proposition 3, the safety of previous operations may be statically checked as soon as labels are known. The following proposition shows that such trees may be used to design an operational model for our language (and also a typing system as soon as a typing system is available for Xdata and reactions).

Proposition 4. *A finite set of abstract paths represents two trees: a tree in $\mathcal{T}_{\mathcal{L}}$ we called 'with abstract root `root`', and one in $\mathcal{T}_{\bar{\mathcal{L}}}$, we called 'centered on `self`'.*

In the following, we freely use the abstract path notation for operations on $\mathcal{T}_{\bar{\mathcal{L}}}$.

2.2 *Xobjects*

An *Xobject* is structured in two parts: an *Xdata* structure together with *reactions*, and is parameterized by means of abstract paths. Parameterization is a convenient way to

```
xstruc
link =   xobject <root.H1> (x:string)
      root.H1.T<a>[x]
   ▶ xend;
message = xobject <> (msg:string)
      <h1 align="center">[msg]
   ▶ xend;
phandler = xobject <M1> (k:int)
      M1 <p>["Visits for this session (cs): " k]
   ▶
      T(<a>[x]) ⇒ (let y=(if (x = "Hello") then "Salut" else "Hello") in
                   (phandler (k + 1))[M1↦(message y)])
   xend;
home = xobject <L1, L2, H1>
      <html>[ <head>[ <title>["Welcome"]] <body>[ H1 <br> L1 <br> L2]]
   ▶ xend;

m1 = (message "Hello");
h1 = (phandler 0)[M1 ↦ m1];
l1 = (link "Increment cs and reload with Hello");
l2 = (link "Increment cs and reload with Salut");
o = home[L1 ↦ l1][L2 ↦ l2][H1 ↦ h1];

begin website = o
```

Ex. 4: Xobjects with components

refer to yet unknown Xobjects while parameter assignment merges partial trees of components. Abstract paths that are used in an Xobject body are declared in the header: in example 4, Xobject `home` expects three subcomponents `L1`,`L2`,`H1`. Note the reference to **`root`**`.H1` in the definition of `link`: this Xobject is expected to be in a tree whose Xobject root has at least a subcomponent for label `H1`.

Composing Xobjects is done by assigning values to abstract path parameters: the expression $o = e_1[Y \mapsto e_2]$ states that o is a copy of the Xobject value of e_1 where the abstract path Y refers to the value of expression e_2 (that should also be an Xobject). Such compositions of Xobjects give rise to two partially defined trees of components as explained before. These partial trees are merged in one tree in the case of an Xstructure declaration, that is a top-level expression. An Xstructure specifies a completely defined Xobject: the tree of components is "closed" and the Xobject is considered as an interactive entry point to the program. A program may have several Xstructure declarations. In example 4, `website` is declared as an entry point. The tree of components is rooted at `o`, that has two links `l1` and `l2` and one phandler `h1` as subcomponents, `h1` having a child `m1`. Two triggers are declared, posted in `l1` and `l2`, authorizing interactive requests of the form `H1.T(x)@website`.

We are now able to precise the operational model. We extend a domain, supposed given for basic types and that includes handlers, by the following kinds of values:

– $\mathcal{IT}$ called a *handler tree value* is a map from abstract paths to handlers. It is a pair of trees $(U, V) \in \mathcal{T}_{\mathcal{L}} \times \mathcal{T}_{\overline{\mathcal{L}}}$, one rooted at abstract path `root` and one centered at `self` with the following partial operations:
  - An 'identifier' operation: let $\mathcal{IT} = (U, V)$, if $V \in \mathcal{T}_{\mathcal{L}}$ and $U \lhd V$ is defined then $\updownarrow\mathcal{IT} = (U \lhd V, U \lhd V)$.

$$\frac{m \ \texttt{fresh}, \mathcal{T} = \{self \mapsto m\}}{\mathcal{E}, \mathcal{H} \vdash \texttt{xobject}\langle Y_1, \ldots, Y_n\rangle \rightarrow e \blacktriangleright sr \ \texttt{xend} \Downarrow \mathcal{T}, \mathcal{H} \cup \{m \mapsto xval(\mathcal{E}, e \blacktriangleright sr)\}}$$

$$\frac{\mathcal{E}, \mathcal{H} \vdash e_2 \Downarrow \mathcal{T}_2, \mathcal{H}_2 \quad \mathcal{E}, \mathcal{H}_2 \vdash e_1 \Downarrow \mathcal{T}_1, \mathcal{H}_1}{\mathcal{E}, \mathcal{H} \vdash e_1[Y \mapsto e_2] \Downarrow \mathcal{T}_1 \lhd_Y \mathcal{T}_2, \mathcal{H}_1}$$

$$\frac{\mathcal{E}, \mathcal{H} \vdash d \Downarrow_d \mathcal{E}_1, \mathcal{H}_1 \quad \mathcal{E}_1, \mathcal{H}_1 \vdash e \Downarrow \mathcal{T}_2, \mathcal{H}_2}{\mathcal{E}, \mathcal{H} \vdash \texttt{xstruc}\ d\ \texttt{begin}\ w = e \Downarrow_d \mathcal{E}_2 \cup \{w \mapsto \updownarrow \mathcal{T}_2\}, \mathcal{H}_2}$$

Fig. 2: Operational semantics for Xobject and Xstructure declarations

- A 'merge with mask' operation $\lhd$: let $\mathcal{T}_1 = (U_1, V_1)$ and $\mathcal{T}_2 = (U_2, V_2)$ be handler tree values, $\mathcal{T} = \mathcal{T}_1 \lhd \mathcal{T}_2$ is the handler tree value (U, V) such that $U = U_1 \lhd U_2$ and $V = V_1 \lhd V_2$.
- A 'merge with mask' operation wrt some path: let $\mathcal{T}_1 = (U_1, V_1)$ and $\mathcal{T}_2 = (U_2, V_2)$ be handler tree values, Y be an abstract path in normal form, $\mathcal{T} = \mathcal{T}_1 \lhd_Y \mathcal{T}_2$ is the handler tree value (U, V) such that
  - if Y has form $self.Z$, $U = U_1 \lhd U_2$ and $V = V_1 \lhd_Y V_2$,
  - if Y has form $root.Z$, $V = V_1$ and $U = U_1 \lhd_Y W$ where $\updownarrow \mathcal{T}_2 = (W, W)$.

– $xval(\mathcal{E}, e \blacktriangleright r)$ serves to denote Xobject closures, where r is a sequence of values of the form $\tau_p(p) \Rightarrow e$, e is an expression and $\mathcal{E}$ is an evaluation environment.

A handler tree value $\mathcal{T}$ may also be part of an evaluation environment as it is a map from abstract paths to values. $dom(\mathcal{T})$ is then the set of abstract paths of $\mathcal{T}$. We consider available in that case an operation $\mathcal{T}^{Y \rightarrow Y'}$ that changes the reference frame of the domain of $\mathcal{T}$ wrt the change from Y to Y'.

The operational semantics of Xobjects is given in figure 2. The semantics corresponding to an Xobject declaration is straightforwardly a closure (as for functions) and assignment of abstract path parameters is similar to the standard treatment of handlers in functional programming: the rule is nothing else but a new value given for the reference (the typing system should ensure that v_2 is an Xobject value). The semantics of an Xstructure declaration follows the semantics of a (top level) definition ($\Downarrow_d$ evaluates definitions).

2.3 *Xdata* and *reactions*

The language for the Xdata part extends XML. XML is basically expressed by means of tree structures where nodes are of the form $<a \ \ l_e>[s]$ where a is the markup of the node, l_e is a list of attribute-value pairs (the value may be the result of the evaluation of an expression), and s is a sequence of XML constituents. XML syntax is extended in the following way: an abstract path may be used in place of an XML node and each node in an XML structure may be labelled by a trigger τ. A trigger τ has the general form $Y.C$ where C is an *interaction tag* (tag for short) and the abstract path Y is the path to the *abstract last possible receiver*. Setting down a trigger means that a functionality should be available, as a GET in HTML or the description of a service. In case of Xstructures, a built-in function `get_Xdata` is available that extracts the Xdatum to build a (standard)

$$\frac{\mathcal{E},\mathcal{H}\vdash e_2\Downarrow v_2,\mathcal{H}_2 \quad \mathcal{E},\mathcal{H}_2\vdash e_1\Downarrow \mathcal{I}_1,\mathcal{H}_1 \quad \mathcal{H}_1(self(\mathcal{I}_1))=xval(\mathcal{E}_1,e\blacktriangleright sr) \quad match(\tau,v_2,sr)=(e_0,\mathcal{E}_0) \quad \mathcal{E}_1\setminus(dom(\mathcal{E}_0)\cup dom(\mathcal{I}_1))\cup\mathcal{E}_0\cup\mathcal{I}_1,\mathcal{H}_1\vdash e_0\Downarrow\mathcal{I}_0,\mathcal{H}_0}{\mathcal{E},\mathcal{H}\vdash\tau(e_2)@e_1\Downarrow\mathcal{I}_0,\mathcal{H}_0}$$

$$\frac{\mathcal{E},\mathcal{H}\vdash e_2\Downarrow v_2,\mathcal{H}_2 \quad \mathcal{E},\mathcal{H}_2\vdash e_1\Downarrow \mathcal{I}_1,\mathcal{H}_1 \quad \mathcal{H}_1(self(\mathcal{I}_1))=xval(\mathcal{E}_1,e\blacktriangleright sr) \quad match(y.Z.C,v_2,sr)=undef \quad \mathcal{E}_1\cup\mathcal{I}_1^{y\to self},\mathcal{H}_1\vdash Z.C(e_2)@\texttt{self}\Downarrow\mathcal{I}_0,\mathcal{H}_0}{\mathcal{E},\mathcal{H}\vdash y.Z.C(e_2)@e_1\Downarrow\mathcal{I}_1\lhd_y\mathcal{I}_0,\mathcal{H}_0}$$

where the function $match(\_,\_,\_)$ checks if a trigger fires a reaction and in this case sends back the expression to be evaluated together with the capture variables given by a standard pattern matching function $matchPatt$ (not given here):

$$\begin{array}{ll} match(Y.C,v,\epsilon) & = undef,\\ match(Y.C,v,Z.D\Rightarrow e\ sr) & = match(Y.C,v,sr)\\ & \quad \texttt{if}\ Y\neq Z\ \texttt{or}\ C\neq D\ \texttt{or}\ matchPatt(v,p)=undef\\ match(Y.C,v,Y.C(p)\Rightarrow e\ sr) & = (e,\Gamma)\ \texttt{if}\ matchPatt(v,p)=\Gamma \end{array}$$

Fig. 3: Operational semantics of requests

XML structure that can be sent interactively. When encountering an abstract path in place of an XML node, the function is recursively called on the value at the abstract path. In case of a trigger, a request is prepared that includes the address of the value of `root` (called the *initial concrete receiver*[3]), the abstract path from it to the last possible receiver, the interaction tag and an XML structure (the parameter of the request). Note that the abstract path to the last possible receiver is now given from `root`, then may be different from the abstract path set up in the Xdata. Such an interactive request is at first received (and executed) by the initial concrete receiver. In fact a request is a first-class citizen that has the general form $\tau(e_2)@e_1$: e_1 is the concrete receiver of the request, $\tau(e_2)$ gives the trigger and the parameter value for the request. The operational semantics is given in figure 3. Due to lack of space, the second rule considers only an identifier beginning the abstract path but similar rules may be given with keywords `parent`, `root` and `self`. The semantics may be rephrased in the following way: if the concrete receiver has an adequate reaction (the reaction matches trigger and parameter of the request), the reaction is evaluated ; otherwise the request is delegated following the path to the last possible receiver until some Xobject has an adequate reaction. It is the type system that is responsible for checking that there cannot be run-time errors. Note that an Xobject value is rebuilt when a delegation occurs. In this paper, we suppose that capture variables are available in standard and XML patterns. However, this may be extended to tag and abstract paths patterns.

Going back to example 4, a tag `T` is set down in the Xdata part of `link`: **`root`**`.H1.T<a>[x]` states that the tag `T` is an anchor. Requests available in this case may be for example `T("Hello")@h1` or `H1.T("Hello")@website`. This last request is the only one that can be used interactively. As `website` has no appropriate reaction, the request is delegated to `h1`, value of `H1` as it is given in `website`. The reaction part of this phandler contains a reaction that is

[3] This is generalized in section 4 where the initial receiver may be different from `root`.

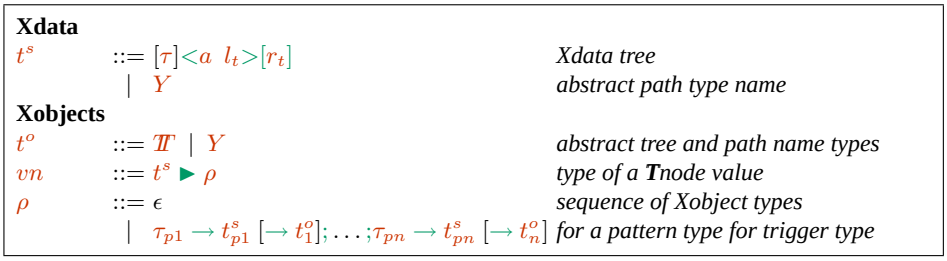


Fig. 4: Type language of FICX (except XML and functional type language)

fired with result the Xobject `(phandler 1)[M1↦(message "Salut")]`. The final value sent back to the requester is `home[L1↦l1][L2↦l2][H1↦(phandler 1)[M1↦(message "Salut")]]`.

3 Language FICX: the type system

FICX is strongly typed: the static typing offers the programmer a way to check its program before execution. Beside the usual benefits, it allows to check the completeness and soundness of the program with respect to interactions as given by request expressions. Obviously, requests included in the program are checked at compile time and interactive requests are checked only on the fly. However, one may control interactive requests by studying available triggers and reactions: e.g. clicks in a browser generate requests and are allowed by triggers set up on some Xdata, web services should answer to declared services. We refer in the later to completeness and soudness with respect to the cover of triggers and reactions in an Xstructure, i.e. a fully defined tree of Xobjects.

The type language is in two parts (see figure 4): t^s stands for standard or Xdata expressions, t^o for Xobjects. The type system for functional expressions is standard, it is extended for Xdata expressions by mimicking the structure. Abstract path variables are also defined as types (these are sequences of constants). The type of an Xobject is an abstraction of a handler tree value as defined in the operational semantics: it is a pair of trees in $\mathcal{T}_{\mathcal{L}} \times \mathcal{T}_{\bar{\mathcal{L}}}$ with abstract paths as labels and nodes valued by the type of its Xdata together with types for the reactions (pattern and result), these values are noted vn in the figure. ρ defines the types for possible reactions (supposed if given as triggers, explicit if given as reactions): this is a sequence possibly empty associating to a trigger pattern and to a pattern type the type of the result if it is defined (summing over the sets of patterns types, and of result types). Taking into account reactions in the type is possible because a program may only include a finite number of Xobject types. However, as reactions and triggers may not be defined in the same Xobject, the type system should propagate pieces of information and when possible merge them to satisfy coherence properties when an Xobject is used as a parameter value for another Xobject.

Due to lack of space, we limit the description of type judgments to Xobjects. The remainder is quite easy to define as it takes up techniques used for functional programming, XML, ... Note that in the following we suppose that variables newly typed were not typed before (type clash with respect to the environment is supposed implicit). Let

Δ be a type environment, i.e. a partial function $\mapsto$ from the set of variable names (including abstract path variables) to types, judgments for an expression and a sequence of reactions are given as follows:

$\Delta \vdash e : t$ where e is an expression and t its type

$\Delta \vdash_r sr : \rho$ where sr is a sequence of reactions with type ρ

3.1 Expression judgment rules: Xobjects

The type of an Xobject is of type $\mathcal{T}$ (`YJoin` builds the pair of trees) whose `self` node value summarizes types of reactions and of triggers set up in the Xdata part. These two data are merged in a single type ρ' considering that triggers not covered by reactions give 'partially defined' types. `Trig` computes the set of triggers set up in its argument. `RIT` (Reaction Intersection Type) retracts triggers (in its second argument) for which reactions are given in its first argument (a reaction type). Parameter assignment is typed by means of a merge and mask operation wrt an abstract path.

$$\frac{\Delta, \overrightarrow{Y_i \mapsto Y_i} \vdash e : t \quad \Delta, \overrightarrow{Y_i \mapsto Y_i} \vdash_r sr : \rho \quad \texttt{RIT}(\rho, \texttt{Trig}(t)) = \rho' \quad t_1^o = \texttt{YJoin}(Y_1, \ldots, Y_n, \texttt{self} \mapsto t \blacktriangleright \rho')}{\Delta \vdash \texttt{xobject} <Y_1, \ldots, Y_n> \rightarrow e \blacktriangleright sr \texttt{ xend} : t_1^o} \qquad \frac{\Delta \vdash e_1 : \mathcal{T}_1 \quad \Delta \vdash e_2 : \mathcal{T}_2}{\Delta \vdash e_1[Y \mapsto e_2] : \mathcal{T}_1 \lhd_Y \mathcal{T}_2}$$

Auxiliary functions for type computation:

$\texttt{Trig}(<a\ \ l_t>[r_t]) = \texttt{Trig}(r_t)$

$\texttt{Trig}(\tau <a\ \ l_t>[r_t]) = \{\tau \rightarrow <a\ \ l_t>[r_t]\} \cup \texttt{Trig}(r_t)$

$\texttt{Trig}(r_{t1}\ r_{t2}) = \texttt{Trig}(r_{t1}) \cup \texttt{Trig}(r_{t2})$

$\texttt{Trig}(t) = \emptyset \quad \texttt{otherwise}$

$\texttt{RIT}(\rho, \emptyset) = \rho$

$\texttt{RIT}(\rho, \{\tau \rightarrow t^s\} \cup T) = \texttt{RIT}(\texttt{RIT1}(\rho, \tau \rightarrow t^s), T)$

$\texttt{RIT1}(\epsilon, \tau \rightarrow t^s) = \tau \rightarrow t^s$

$\texttt{RIT1}(\tau_1 \rightarrow t_1^s\ [\rightarrow t_1^o]; \rho, \tau \rightarrow t^s) = \tau_1 \rightarrow t_1^s\ [\rightarrow t_1^o]; \texttt{RIT1}(\rho, \tau \rightarrow t^s \setminus \tau_1 \rightarrow t_1^s)$

3.2 Expression judgment rules: requests

A request $\tau(e_2)@e_1$ has a type given by the result of the fired reaction. This reaction should be on the path (given in the trigger τ) beginning from the receiver (value of e_1). If the type of e_1 includes a compatible reaction (function $testreac$), then the request has the type of the result:

$$\frac{\Delta \vdash e_1 : \mathcal{T}_1 \quad \Delta \vdash e_2 : t_2 \quad testreac(\tau, t_2, \rho(\texttt{self}(\mathcal{T}_1))) = \mathcal{T}_2}{\Delta \vdash \tau(e_2)@e_1 : \mathcal{T}_1 \lhd_y \mathcal{T}_2}$$

otherwise the request is delegated to the first part of the trigger. The following rule concerns the case where this first part is some child y. The fact that the type of an Xobject is a global environment (not limited to local constituents) allows for similar rules when the first element of the path is `parent`, or `root`.

$$\frac{\Delta \vdash e_1 : \mathcal{T}_1 \quad \Delta \vdash e_2 : t_2 \quad testreac(y.Y.C, t_2, \rho(\texttt{self}(\mathcal{T}_1))) = \texttt{undef} \quad val(y(\mathcal{T}_1)) \textit{ defined} \quad \Delta \setminus dom(\mathcal{T}_1) \cup \mathcal{T}_1{}^{y \to \texttt{self}} \vdash Y.C(e_2)@\texttt{self} : \mathcal{T}_2}{\Delta \vdash y.Y.C(e_2)@e_1 : \mathcal{T}_1 \lhd_y \mathcal{T}_2}$$

where
$testreac(\tau, t, \epsilon) = \texttt{undef}$
$testreac(\tau, t, \tau_p \to t_1 \to t_2^o ; \rho) = t_2^o$ if $\tau <:_\tau \tau_p$ and $t <: t_1$
$testreac(\tau, t, \tau_p \to t_1 \to t_2^o ; \rho) = testreac(\tau, t, \rho)$ otherwise

It is not too difficult to prove a safety theorem stating that well-typed expressions are evaluable, i.e. there cannot be evaluation errors (provided for the functional language part an operational semantics safe with respect to a classic typing):

Theorem 1. *Let e be an expression of the language, if $\vdash e : t$ is provable, then there exist $v, \mathcal{H}'$ such that $\vdash e \Downarrow v, \mathcal{H}'$ is provable.*

The proof results from a careful study of the various rules. Note that the typing rules ensure that the Xobject parameters of a request have appropriate reactions, and that the operational semantics rules are in correspondence with the typing ones.

3.3 Xobject evolution and completeness

Interactive requests may be controlled by means of a static study of Xstructures: interactive requests that correspond to declared triggers on Xobjects may be executed without errors. This is particularly the case with web sites when requests are built by the browser after a user click, it is also the case with web services if clients conform a WSDL or BPEL declaration. However, a dynamic type checking has to be added as one cannot be sure that requests are well-formed with respect to some declaration. Besides this soundness property, the completeness stands for checking that reactions given in Xobjects are correctly declared. The typing rule for the Xstructure expression is given in figure 5 ($\Delta \vdash_d d : \Gamma$ is a type judgment for definitions, Γ is a type environment, $\vdash_S$ is used for top-level type judgments). The complexity of the rule comes from the fact that the pair of trees have now to be merged. This is done by the function FDX that also ensures that soundness and completeness properties are satisfied:

- 1st line does an 'identifier' operation,
- 2nd line ensures that reactions cover triggers (ρ is fully defined), and recursively through reactions,
- 3rd line ensures that Xobjects in the environment are known.

3.4 Subtyping

A subtyping system is supposed to be given for the XML part of the language. It is extended for XML and trigger patterns. The subtyping system for Xobjects has the following characteristics: let t_1^o and t_2^o be two Xobject types, $t_1^o <: t_2^o$ (t_1^o extends t_2^o) if

- For each abstract path Y, possibly with type t_Y in t_2^o, Y is present in t_1^o, if required with a type $t'_Y <: t_Y$.

Xstructure

$$\frac{\Delta \vdash_d d : \Gamma \quad \Gamma \vdash e : t^o \quad \mathsf{FDX}(t^o) = t'^o}{\Delta \vdash_S \texttt{xstruc}\ d\ \texttt{begin}\ w = e : w \mapsto t'^o}$$

where

$\mathsf{FDX}(\mathcal{T} <: t^o)$	$= \mathcal{T}' <: t^o$	iff $\mathcal{T}' = \updownarrow \mathcal{T}$ and $\rho(\texttt{self}(\mathcal{T}')) = \mathsf{FDX}_\rho(\rho(\texttt{self}(\mathcal{T})))$ and $\mathsf{FDX}_T(\mathcal{T}')$ is true
$\mathsf{FDX}_\rho(\epsilon)$	$= \epsilon$	
$\mathsf{FDX}_\rho(\tau_p \to t_p^s \to t^o;\rho)$	$= \tau_p \to t_p^s \to \mathsf{FDX}(t^o);\mathsf{FDX}_\rho(\rho)$	
$\mathsf{FDX}_T(\mathcal{T})$	is true	iff forall node n of $\mathcal{T}$, n has a value

Fig. 5: Soundness and completeness of Xstructures (entry points)

– XML subtyping should be satisfied as well as subtyping with respect to triggers (triggers in t_2^o should appear in t_1^o).
– Each reaction defined in t_2^o has its counterpart in t_1^o.

Typing constraints may then be added to the language as usual.

4 Extensions

In the current setting, requests should initially be sent to the (concrete) root of a tree of components, and, if necessary, delegated to some adequate Xobject wrt an abstract path to a final receiver. However, this constraint is neither formally necessary nor practically wishful. In fact, this delegation mechanism is safe as soon as the concrete receiver is known and the nodes in the abstract path have each a value. Moreover, sending a request directly to some node in a tree of components allows for an Ajax-like mechanism. Ajax [3] is a web development technique for creating interactive web applications and is intended to increase the web page's interactivity, speed, and usability. A response may be given as a part of an HTML document and it is the client responsability to replace the old value by the new one at the right place (a DOM-based mechanism generally), avoiding the page to be completely reloaded. This mechanism may be modelled in our framework as a request to some specific Xobject maybe different from a root, this Xobject being specified when setting up a trigger. To take care of this generalization, the syntax of a request does not change and a trigger should be set up as $Y'{:}Y.C{<}a\ \ l_e{>}[s]$ where the initial receiver is such that Y' is the path from `self` to it.

Let us replace the definition of `link` in example 4 by:

```
link = xobject <root.H1,self.parent.H1.M1> (x:string)
     root.H1:self.parent.H1.M1.T<a>[x]
  ▶ xend;
```

The trigger `root.H1:self.parent.H1.M1.T<a>[x]` in `link` states that the initial (resp. final) receiver for the tag `T` should be the value at `root.H1` (resp. `self.parent.H1.M1`). A request corresponding to such a tag could be `M1.T("Hello")@h1`. Operational and typing rules are slightly more complex as one should manage delegation not only wrt direct subcomponents (daughters in the component tree) but also to parents of a node. Using such a

mechanism interactively requires more theoretical and practical investigations. When a request is received at first by the root of a tree of components, a new tree of components is created for the response, hence the tree structure is fully defined. This is no more the case when a request is initially sent to a node different from the root: either the tree is rebuilt but efficiency is lost, or a replacement is done but completeness and soundness wrt reactions may not be guaranteed.

More generally, we currently study carefuly the theoretical meaning of interaction, i.e. setting up triggers as a dual to requesting reactions. An operational semantics of interaction may be given in terms of processus calculus while keeping the semantics of the delegation process described in this paper. This may be fruitful for extending expressivity of interaction. For instance, associating multiple receivers to the same tag could be useful in pratice when one wants to fire reactions in different components. However, the operational semantics is not obvious if order of execution matters. This is not the case when replies concern disjoint parts of the concrete tree of components.

5 Related works

Our work concerns two different communities: object and XML programming as the real novelties of FICX are program modularity through Xobject component trees and static typing for structures that mix XML and functional parts. However our approach is uneasily comparable to the standard object-oriented paradigm in that modularity is got by partonomy rather than inheritance. Moreover Xobjects are in fact immutable as parameter instantiation and requests create each time new Xobjects. It is easier to relate our work to different areas in semi-structured data field (embedded calls, type checking, web services).

Embedded calls in XML documents: Concepts of triggers and reactions in FICX are close to the (not new) idea of embedded calls in XML structures. Previous works from this area can be classified in two categories: data oriented and code oriented. In the data-oriented approach, the XML structure is enriched with intensional data. In Macromedia [10], Appache Jelly [14], AXML [1, 5], database or web service queries help to dynamically complete XML documents and a declaration of services may be available. Including expressions and triggers in Xdata has the same objectives even if we do not focus on the problems of distributed stream data. The difference mainly relies on the fact that our language is strongly type-checked although works just cited extend loosely XML types or schemas. For example, type checking in AXML is based on an extension of XML schemas in order to describe data types needed in an exchange. The code-oriented approach, as popularized by PHP [13], JSP [12], ASP [11], tries to introduce code in XML structures in order to allow parameterization and dynamicity of websites. However, no static checking is proposed so there is no guarantee the resulting XML structure is correct before run-time.

Typed XML processing languages: Many works (see [9] for a general survey) exist that propose strongly typed languages for manipulating XML data: Xact [7] (an extension of JAVA), CDUCE [2] (a ML-like language), XSTATIC [4] (an extension of C#). These programming languages allow to manipulate, to create and to check XML documents thanks to a powerful parser and a type inference system. They extend a pro-

gramming language by means of a typed language for XML document manipulation. However they do not support code integration in XML data. On the other hand XMλ [8] is closer to our concerns. It uses a Haskell-like syntax and treats XML documents as native values. It allows to include typed expressions and embedded functions calls in XML documents while proposing a powerful type-checking. All these systems lack a general framework able to design software in a modular and homogeneous way.

6 Conclusion

FICX is a programming language that focuses on designing trees of components, close to part-of relationship. FICX is well suited to XML-like languages by integrating static and dynamic aspects in an homogeneous framework. A powerful delegation process for interactions is defined thanks to the tree structure. The study of FICX (type checking, operational semantics) is facilitated by the fact that XML data and Xobject evolutions are encoded in the same language. Such a tree components programming paradigm increases expressivity with respect to other works where XML values may be computed only by means of direct calls.

References

1. Serge Abiteboul, Omar Benjelloun, and Tova Milo. Positive active xml. In *ACM SIGMOD-/PODS 2004 Conference*, June 2004.
2. Véronique Benzaken, Giuseppe Castagna, and Alain Frisch. Cduce: An xml-centric general purpose language. In *Proc. 8th ACM SIGPLAN International Conference on Functional Programming (ICFP 2003)*, Uppsala, Sweden, August 2003.
3. Dave Crane, Eric Pascarello, and Darren James. *Ajax in Action*. Manning Publications, 2005.
4. Vladimir Gapeyev, Michael Y. Levin, Benjamin C. Pierce, and Alan Schmitt. XML goes native: Run-time representations for Xtatic. In *14th International Conference on Compiler Construction*, April 2005.
5. Active XML homepage. http://activexml.net.
6. Objective CAML homepage. http://caml.inria.fr/ocaml/index.en.html.
7. Christian Kirkegaard and Anders Møller. Type checking with XML Schema in Xact. Technical Report RS-05-31, BRICS, September 2005.
8. Erik Meijer and Mark Shields. XMλ: A functional language for constructing and manipulating XML documents. (Draft), 1999.
9. Anders Møller and Michael I. Schwartzbach. The design space of type checkers for XML transformation languages. In *Proc. 10th International Conference on Database Theory, ICDT '05*, volume 3363 of *LNCS*, pages 17–36. Springer-Verlag, January 2005.
10. Macromedia Coldfusion MX. http://www.macromedia.com/software/coldfusion.
11. Active Server pages. http://www.asp.net/.
12. Sun's JAVA Server Pages. http://java.sun.com/products/jsp.
13. The PHP Hypertext Preprocessor. http://www.php.net.
14. Jelly: Executable XML. http://jakarta.apache.org/commons/jelly.