Programmation fonctionnelle et XML : PaX

D. Plaindoux[1]

1: plaindoux@mipnet.fr

Résumé

Face à l'émergence de XML comme standard pour la formulation de documents, d'échanges de données, comme support pour les bases de données semi-structurées, un ensemble de travaux adoptent une approche formelle dans le sens de la validation afin de proposer des outils sûrs intégrés dans des processus de transformation. PaX s'inscrit dans ce mouvement en appliquant une approche simple et consistante à partir de principes bien connus dans le monde du fonctionnel et de les unifier avec des principes du monde XML. Cela couvre le langage des types par l'introduction de la régularité et donc nécessite une révision du principe du typage et du principe de filtrage afin d'aboutir à un outil de manipulation expressif et ouvert.

L'objectif de ce document est une présentation du typage et du filtrage du langage PaX et propose donc une possibilité d'intégration de la structuration XML au sein d'un langage fonctionnel avec une ébauche de formalisation.

1. Introduction

L'utilisation des types réguliers n'est pas une chose nouvelle. En fait, si nous faisons un petit retour en arrière leur utilisation a été entre autres introduite pour la spécification de protocoles de communication [19]. Le but était alors la définition de processus communiquants véhiculants leur propre méthode de communication de données. Parallèlement, le monde SGML intégra la régularité pour la description de types de documents. Une version épurée de SGML a conduit à la définition de XML qui est actuellement le formalisme phare pour la représentation de documents [24], la communication d'informations [8], la définition de base de données semi-structurées [17] ou pour l'introduction de principes de programmation [11].

Un terme XML, ou instance, doit vérifier une seule propriété : être bien formé. A ceci peut être ajouté de la notion de spécification introduisant la conformité de documents relativement à une DTD[1]. En terme de langage, XML introduit le langage des termes et les DTD le langage des types. La difficulté est alors la possibilité de manipuler des termes pouvant être typés. La DTD actuellement est utilisée essentiellement lors de la validation de documents et donc dans un processus dynamique i.e. analyseur, éditeurs spécialisés.

L'intérêt actuel pour XML porte sur le principe de validation de processus de transformation dédiés. Ces travaux consistent à avoir une approche formelle des principes appliqués dans XML soit par la reformulation de propositions du W3C [26] soit par la définition d'outils sophistiqués de manipulation comme XDuce [12], XML-QL [1], XMλ [10] et YATL [17].

2. Transformation XML

Quand il s'agit de transformer un document XML deux approches sont actuellement proposées. La première est basée sur la notion d'analyse du document au fil de l'eau et la seconde sur la notion de représentation d'un document par un arbre abstrait en mémoire et par un principe de réécriture bien connu.

[1]Document Type Definition

2.1. Transformation par événements

Les premiers outils en fait, découlent du monde SGML et sont bâtis sur les analyseurs qui pilotent la transformation par le biais d'événements tels que l'ouverture, la fermeture de balise et la capture de texte libre i.e. `pcdata`. De cette méthodologie sont nés des langages tels que Omnimark [20], Balise [4] ou l'analyseur générique SP de J.Clark [7]. Actuellement une volonté d'uniformisation des événements a conduit à la norme SAX [22] proposant une approche homogène de la programmation par événements basée sur la notion d'analyse au fil de l'eau et ne proposant donc qu'une vue partielle du document à un instant donné de la transformation. La gestion d'un environnement d'analyse est alors faite par la couche dédiée à la lecture et la validation de documents permettant d'avoir une vue plus précise du document par la mise à disposition d'un ensemble de prédicats utilisés pour la sélection de règles de réécriture. Dans cette catégorie, nous retrouvons le langage de transformation XSL [11] et la notion d'exploitation d'environnement de l'analyseur introduite par XPath [16].

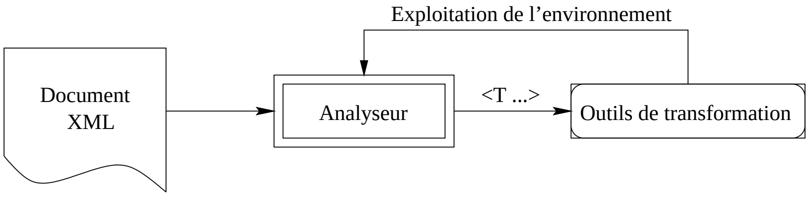

FIG. 1 – Transformation conduite par les événements

Cette vue partielle et le caractère implicite de l'environnement rendent toute vérification statique de processus de transformation difficile ; ceci en fait découle du caractère sensitif au contexte de ces outils. Par contre, l'analyse étant au fil de l'eau de tels outils proposent des temps d'exécution faibles ce qui peut être souhaitable quand la taille du document excède la centaine de méga-octets.

2.2. Transformation d'arbres

La seconde approche consiste à proposer la transformation d'un arbre abstrait dénotant une instance XML. L'environnement d'analyse doit être alors géré et explicité lors de l'écriture des translateurs occultant donc le caractère sensitif cité précédemment.

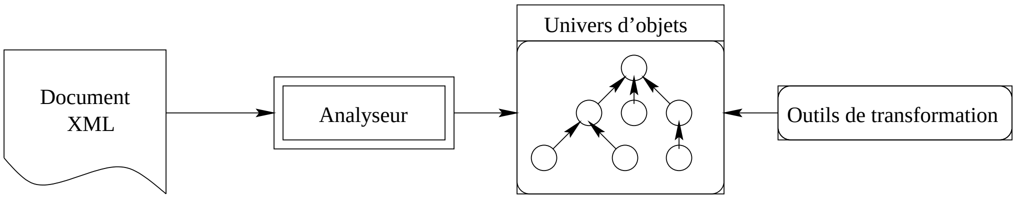

FIG. 2 – Transformation basée sur les arbres abstraits

Ce type de transformation peut être effectué par le biais d'une représentation d'arbre en mémoire et d'une interface de navigation dans cet arbre comme le propose la couche DOM [3], composant CORBA. Ce type de manipulation reste cependant de trop bas niveau pour pouvoir lui appliquer simplement tout principe de validation.

Actuellement, un ensemble d'outils de plus haut niveau sont à l'étude et proposés dans un but de validation statique de transformation d'arbre abstrait. Cette approche est adoptée plus particulièrement pour QL [1] XDuce [12] Xmλ [10] et PaX. Ces outils intègrent :

. la régularité dans le langage des types,
. les balises XML dans le langage des termes et

. un paradigme bien connu dans le monde des langages fonctionnels : le filtrage de forme ou pattern matching.

3. Le langage PaX

Dans le suite, nous présentons le langage des types et des termes. L'accent est mis sur la puissance d'expression du langage tout en gardant à l'esprit une maîtrise totale des processus associés à XML et en restant *fidèle* à l'approche fonctionnelle.

3.1. Définition de types

Notre point de départ est l'introduction de la régularité dans le langage des types tout en essayant de montrer la correspondance dans l'univers des DTDs.

3.1.1. De la DTD aux types PaX

Dans une DTD, la spécification de type est caractérisée par la définition de règles dites `ELEMENT` enrichies par des règles dites `ATTLIST`. La première famille de règles permet de caractériser le nom d'une balise ou constructeur et le type de son contenu. La seconde famille permet d'adjoindre des attributs aux balises. A noter que ces attributs sont sujets à une notion dite de présence : ils peuvent être optionnels ou obligatoires.

En résumé, une DTD permet de spécifier un langage pour une description ciblée de documents et ce langage est en fait une spécification de tous les arbres abstraits exprimables et se distingue donc des langages définis par les grammaires dites BNF [2]. En effet, les DTDs permettent de décrire une syntaxe concrète d'arbres et normalisent la définition de la syntaxe abstraite alors que les règles BNF ne définissent que la syntaxe abstraite.

Les spécifications `ELEMENT`

Afin d'illustrer la spécification de type dans PaX nous allons utiliser une DTD introduisant une formulation simple de documents bâtis sur la structure classique de `document`, `chapitre`, `section` comme le montre la définition suivante :

```
<!DOCTYPE DOC [
 <!ELEMENT Document     (Titre,Resume?,Chapitre+) >
 <!ELEMENT Resume       (Para+) >
 <!ELEMENT Chapitre     (Titre,Para*,Section*) >
 <!ELEMENT Section      (Titre,Para*) >
 <!ELEMENT (Titre|Para) (#PCDATA) >
]>
```

La spécification de type en PaX s'inspire largement de ce formalisme et s'exprime de la façon suivante :

```
type DOC =
  <Document Titre,Resume?,Chapitre+ >
| <Resume Para+ >
| <Chapitre Titre,Para*,Section* >
| <Section Titre,Para* >
| <Titre|Para string >
```

En XML la DTD est orientée fichier et introduit implicitement la disjonction de toutes les balises constituant le type de la DTD. Ceci est nécessaire dans la mesure où un document attaché à une DTD donnée peut avoir comme balise de plus haut niveau toutes celles exprimées. Ceci permet d'un point de vue langage d'avoir autant de règles de départ que de règles exprimées.

Implicitement donc, toute définition de types induit les types partiels comme l'illustre l'exemple suivant :

```
DOC.Document = <Document DOC.Titre,DOC.Resume?,DOC.Chapitre+ >
DOC.Resume   = <Resume DOC.Para+ >
DOC.Chapitre = <Chapitre DOC.Titre,DOC.Para*,DOC.Section* >
...
```

Il est aussi possible dans une DTD de spécialiser la règle de départ comme suit :

```
<!DOCTYPE DOC Document [ ... ]>
```

En PaX cela est concrétisé par l'utilisation du type associé i.e. `DOC.Document`.

Les spécifications `ATTLIST`

L'attachement d'attributs à une balise est fait par les règles dites `ATTLIST`. Un attribut possède un type, une notion de présence et parfois une valeur par défaut. En PaX, seule la notion de valeur par défaut ne peut actuellement être exprimée.

```
<!DOCTYPE DOC [
 <!ELEMENT Document Titre,Resume?,Chapitre+ >
 ...
 <!ELEMENT Para (#PCDATA|Ref)* >
 <!ELEMENT Ref  EMPTY >

 <!ATTLIST Document kind (ARTICLE|BOOK|DRAFT) #IMPLIED >
 <!ATTLIST (Chapitre|Section) id CDATA #REQUIRED >
 <!ATTLIST Ref idref CDATA #REQUIRED >
]>
```

Dans une DTD, les règles `ELEMENT` et `ATTLIST` proposent une définition complète de balise avec un point de vue déstructuré de par la séparation de ces déclarations. En fait ceci permet d'introduire la notion de réutilisabilité et de généricité au sein de la spécification par le biais des entités et des inclusions appelées sections marquées. Contrairement à XMλ qui introduit la notion d'entité, PaX impose la déclinaison de tous les types et induit donc la perte de la notion d'inclusion conditionnelle pour la spécification des types. De plus l'aspect déstructuré des déclarations est perdu comme le montre la dénotation du type précédent.

```
type DOC.kind = { "ARTICLE","BOOK","DRAFT" }

type DOC =
  <Document { kind : DOC.kind? } Titre,Resume?,Chapitre+ >
| <Resume Para+ >
| <Chapitre { id : string } Titre,Para*,Section* >
| <Section { id : string } Titre,Para* >
| <Para (string|Ref)+ >
| <Titre string >
| <Ref { idref : string } empty >
```

Par contre, l'expression des types d'attributs optionnels repose sur l'utilisation de l'opérateur d'occurrence '?'. En fait, l'ensemble des types exprimables pour les attributs dans PaX est un sur-ensemble des types exprimables dans les DTDs. En effet, il est possible de typer un attribut tout comme le contenu d'une balise tel que le propose l'exemple suivant :

```
type DOC =
  <Document { kind : DOC.kind? ; content : Titre,Resume?,Chapitre+ }>
| ...
```

Ceci implique la possibilité de plonger un type DTD dans PaX mais l'inverse n'est pas obligatoirement vrai. En fait, nous pouvons nous poser la question de la restriction syntaxique et sémantique des attributs dans XML. Cette contrainte conduit à définir un analyseur XML augmenté par de nouveaux analyseurs pour la reconnaissance des valeurs attachées aux attributs quand leur signification n'est autre qu'une simple chaîne de caractères comme cela est proposé pour XPath dans XSL ou plus simplement le langage JavaScript dans les pages HTML.

3.1.2. Parallèle avec les types inductifs des langages fonctionnels

Le langage des types tel qu'il est proposé intègre la notion de constructeurs par le biais des balises et la définition de types injectifs repose sur l'opérateur de disjonction. Ainsi, nous pouvons exprimer les types injectifs classiques par l'utilisation conjointe des constructeurs et de la disjonction comme le montre l'exemple suivant :

```
type Bool  = <true|false empty>
type Peano = <z0 empty> | <succ Peano>
```

Par contre, la déclaration de type tel que le propose PaX permet de mettre en évidence des structures de types plus complexes comme la notion de types encapsulés sans nécessairement être obligé de déclarer des types intermédiaires comme en OCaml (ici le type `list` simule la répétition) :

```
type html    = TABLE of htmltr list | ...
and  htmltr  = TR    of htmltd list
and  htmltd  = TD    of html   list
```

Le langage des types permet d'avoir une écriture plus synthétique :

```
type html = <table <tr <td html*>*>*> | ...
```

Ceci permet d'introduire simplement la notion de dépendance de type en terme d'encapsulation sans nécessairement déclarer tous les intermédiaires.

3.1.3. Définition de Types polymorphes

Tout comme XMλ, PaX dispose de la possibilité de définir des types polymorphes. Ce dernier point permet de retrouver le principe de généricité introduit par les inclusions conditionnelles des DTDs. C'est entre autres le cas de toutes les DTDs normalisées et utilisées comme par exemple la structuration par tableaux introduite par la norme `CALS`.

Le polymorphisme permet aussi de retrouver des structures traditionnelles telles que la liste homogène ou l'arbre n-aire homogène.

```
type List[X] = <nil empty> | <cons X,List[X]>
type Tree[X] = <tree (X,Tree[X]*)?>
```

Du point de vue spécialisation, il existe une contrainte importante. Un type polymorphe ne peut être ni spécialisé par le type empty ni par une séquence de types. Ceci en fait est introduit dans le but de conserver l'aspect atomique[1] de la donnée lors de typages de filtrage opérant par décomposition structurelle.

3.1.4. Indéterminisme et spécifications de types

La norme XML ne permet pas la définition de DTDs dites indéterministes. En fait, tout analyseur de DTD XML est censé rejeter les spécifications construitent sur ce principe. Cette contrainte importante découle de SGML et permet de définir le processus validant en omettant le principe de retour en arrière lors de choix multiples.

Le langage des types PaX permet la spécification de types indéterministes comme suit :

[1]Par atomique nous caractérisons les types initiaux, la balise et la disjonction de ces deux premiers types

```
type Ind[X] =
  <Ind (<e empty>,<e X*>   )
     | (<e X*>   ,<e empty>)
     | (<e X+>   ,<e X+>   )
  >
```

Intuitivement, il existe une forme déterministe pouvant être obtenue, par l'application de la distributivité de la disjonction sur les balises, qui est la suivante :

```
type Ind[X] = <Ind <e X*>,<e X*>>
```

Dans la suite du document nous supposons que tous les types spécifiés sont déterministes en présence de balises. Nous verrons cependant que cette contrainte forte ne peut être établie aussi simplement et influe donc sur le principe de validation proposé.

3.2. Langage des types et sous-typage

Le langage des types qui suit est bâti sur trois sortes :
. $\mathcal{I}$ représente les identifiants XML à savoir les noms de balises et d'attributs,
. $\mathcal{P}$ représente les types dits initiaux tels que les entiers, les caractères et les chaînes de caractères et
. $\mathcal{X}$ est l'ensemble des variables de type.

Ty	$=$	$Ty \rightarrow Ty$	(type fonctionnel)
	$\mid$	$Ty \mid Ty$	(disjonction)
	$\mid$	Ty, Ty	(séquence)
	$\mid$	$Ty*$	(répétition)
	$\mid$	ϵ	(type vide)
	$\mid$	Any	(type atomique général)
	$\mid$	$\langle\!\langle \mathcal{I}; Ry; Ty \rangle\!\rangle$	(type de balise)
	$\mid$	$\mathcal{P}$	(type prédéfini i.e. int, string)
	$\mid$	$\Pi\mathcal{X}.Ty$	(type polymorphe)
	$\mid$	$\mathcal{X}$	(variable de type)
Ry	$=$	$\{\mathcal{I} : Ty; \cdots; \mathcal{I} : Ty\}$	(ensemble d'attributs)

[Langage des types]

Classiquement, les opérateurs '+' et '?' sont interprétés comme suit :
. $T+ = T \mid (T, T*)$
. $T? = \epsilon \mid T$

Le principe de sous-typage passe tout d'abord par la définition de la fonction de dépliage par nécessité des types réguliers. En fait, ce dépliage permet de donner toutes les formes possibles avec comme propriété l'atomicité des têtes de séquences. A partir de cette définition nous pouvons alors spécifier le sous-typage en procédant par induction sur l'ensemble des types calculés. Cette méthode découle du principe de calcul des automates de reconnaissance de types réguliers par opposition à la formalisation actuelle de XDuce qui repose lui sur le principe d'automates d'arbres.

3.2.1. Dépliage de type régulier

Les règles suivantes introduisent le principe de dépliage de type régulier à un pas. Ces dernières reposent en fait sur le principe de calcul du premier terme et de ses suivants appliqué lors de la définition d'automates de reconnaissance du langage associé.

$\mathcal{U}[\![\_]\!]\_ : Ty \rightarrow (Ty)\mathsf{set} \rightarrow (Ty)\mathsf{set}$

$$
\begin{array}{lll}
(1) & \mathcal{U}[\![T]\!]_\Delta & = \emptyset \text{ si } T \in \Delta \\
(2) & \mathcal{U}[\![p]\!]_\Delta & = \{p\} \text{ si } p \in \mathcal{P} \\
(3) & \mathcal{U}[\![X]\!]_\Delta & = \{X\} \text{ si } X \in \mathcal{X} \\
(4) & \mathcal{U}[\![\Pi X.T]\!]_\Delta & = \{\Pi X.T\} \\
(5) & \mathcal{U}[\![\epsilon]\!]_\Delta & = \{\epsilon\} \\
(6) & \mathcal{U}[\![\mathsf{Any}]\!]_\Delta & = \{\mathsf{Any}\} \\
(7) & \mathcal{U}[\![\langle\!\langle t; A; T\rangle\!\rangle]\!]_\Delta & = \{\langle\!\langle t; A; T\rangle\!\rangle\} \\
(8) & \mathcal{U}[\![T_1 \mid T_2]\!]_\Delta & = \mathcal{U}[\![T_1]\!]_\Delta \cup \mathcal{U}[\![T_2]\!]_\Delta \\
(9) & \mathcal{U}[\![T_1*]\!]_\Delta & = \mathcal{U}[\![\epsilon \mid (T_1, T_1*)]\!]_{\Delta\cup\{T_1*\}} \\
(10) & \mathcal{U}[\![T_1, \epsilon]\!]_\Delta & = \mathcal{U}[\![T_1]\!]_\Delta \\
(11) & \mathcal{U}[\![T_1*, T_2]\!]_\Delta & = \mathcal{U}[\![\epsilon \mid ((T_1, T_1*), T_2)]\!]_{\Delta\cup\{T_1*\}} \\
(12) & \mathcal{U}[\![(T_1, T_2), T_3]\!]_\Delta & = \mathcal{U}[\![T_1, (T_2, T_3)]\!]_\Delta \\
(13) & \mathcal{U}[\![(T_1 \mid T_2), T_3]\!]_\Delta & = \mathcal{U}[\![(T_1, T_3) \mid (T_2, T_3)]\!]_\Delta \\
(14) & \mathcal{U}[\![T_1, T_2]\!]_\Delta & = \left\{ \begin{array}{ll} \{t, T_2 \mid t \in \mathcal{U}[\![T_1]\!]_\Delta / \epsilon\} \cup \mathcal{U}[\![T_2]\!]_\Delta & \text{si } \epsilon \in \mathcal{U}[\![T_1]\!]_\Delta \\ \{t, T_2 \mid t \in \mathcal{U}[\![T_1]\!]_\Delta\} & \text{sinon} \end{array} \right.
\end{array}
$$

[Dépliage par nécessité de types réguliers]

La principale remarque concerne les variables de types. En effet, il est important de noter qu'une variable de type ne peut être ni spécialisée par le type ϵ ni par un séquence de type afin de préserver les propriétés introduites par les règles (11), (12) et (14).

La règle (1) est nécessaire pour la terminaison du calcul dans le cas de dépliage de types complexes comme `T**`.

$$
\begin{array}{lll}
\mathcal{U}[\![\mathrm{T}**]\!]_\emptyset & = \mathcal{U}[\![\epsilon \mid (\mathrm{T}*, \mathrm{T}**)]\!]_{\{\mathrm{T}**\}} & (9) \\
& = \mathcal{U}[\![\epsilon]\!]_{\{\mathrm{T}**\}} \cup \mathcal{U}[\![\mathrm{T}*, \mathrm{T}**]\!]_{\{\mathrm{T}**\}} & (8) \\
& = \{\epsilon\} \cup \{\mathrm{T}, (\mathrm{T}*, \mathrm{T}**)\} \cup \mathcal{U}[\![\mathrm{T}**]\!]_{\{\mathrm{T}**\}} & (14) \\
& = \{\epsilon; \mathrm{T}, (\mathrm{T}*, \mathrm{T}**)\} & (1)
\end{array}
$$

A partir de cette formulation du dépliage à un pas de formes régulières on peut en déduire des propriétés telles que l'associativité de la séquence introduite par la règle (12), l'élément neutre ϵ pour la séquence introduit par les règles (10) et (14) et finalement l'associativité et la commutativité de la disjonction introduites par la règle (8).

3.2.2. Types dépliés déterministes

Sur l'ensemble produit nous pouvons définir la forme déterministe en factorisant le premier élément des types calculés par le dépliage à un pas comme suit.

$\mathcal{D}[\![\_]\!]\_ : (Ty)\mathsf{set} \rightarrow (Ty)\mathsf{set} \rightarrow (Ty)\mathsf{set}$

$$
\begin{array}{lll}
(1) & \mathcal{D}[\![\emptyset]\!]_{s'} & = s' \\
(2) & \mathcal{D}[\![\{X, T_1\} \cup s]\!]_{\{X, T_2\}\cup s'} & = \mathcal{D}[\![s]\!]_{\{X, (T_1 \mid T_2)\}\cup s'} \\
(3) & \mathcal{D}[\![\{X\} \cup s]\!]_{\{X, T_2\}\cup s'} & = \mathcal{D}[\![s]\!]_{\{X, (\epsilon \mid T_2)\}\cup s'} \\
(4) & \mathcal{D}[\![\{X, T_1\} \cup s]\!]_{\{X\}\cup s'} & = \mathcal{D}[\![s]\!]_{\{X, (T_1 \mid \epsilon)\}\cup s'} \\
(5) & \mathcal{D}[\![\{T\} \cup s]\!]_{s'} & = \mathcal{D}[\![s]\!]_{\{T\}\cup s'} \text{ sinon}
\end{array}
$$

[Déterminise simple]

3.2.3. Règles de sous-typage

Les règlent suivantes permettent récurrence. En fait, le principe utilisé est similaire à celui introduit dans le cadre du sous-typage dans XDuce [13] et stipule la validité d'un sous-typage à une étape si l'interprétation en terme de dépliage par nécessité à l'étape suivante est vraie.

$$(\mathrm{Inter}_{\subseteq})\ \frac{\forall t_1 \in \mathcal{D}[\![\mathcal{U}[\![T_1]\!]_\emptyset]\!]_\emptyset,\ \exists t_2 \in \mathcal{D}[\![\mathcal{U}[\![T_2]\!]_\emptyset]\!]_\emptyset \quad \Gamma, T_1 \subseteq T_2 \vdash t_1 \subseteq t_2}{\Gamma \vdash_* \mathrm{T}_1 \subseteq \mathrm{T}_2} \qquad (\mathrm{Env}_{\subseteq})\ \frac{}{\Gamma, \mathrm{T}_1 \subseteq \mathrm{T}_2 \vdash_* \mathrm{T}_1 \subseteq \mathrm{T}_2}$$

Les règles suivantes introduisent le sous-typage sur les formes dépliées. En fait, celles-ci proviennent de système classique [21].

$$(\mathrm{Refl}_{\subseteq})\ \frac{}{\Gamma \vdash \mathrm{T} \subseteq \mathrm{T}} \qquad (\rightarrow_{\subseteq})\ \frac{\Gamma \vdash_* \mathrm{T}_3 \subseteq \mathrm{T}_1 \quad \Gamma \vdash_* \mathrm{T}_2 \subseteq \mathrm{T}_4}{\Gamma \vdash \mathrm{T}_1 \rightarrow \mathrm{T}_2 \subseteq \mathrm{T}_3 \rightarrow \mathrm{T}_4}$$

$$(\Pi_{\subseteq})\ \frac{\Gamma, \mathrm{X} \subseteq \mathsf{Any} \vdash_* \mathrm{T}_1 \subseteq \mathrm{T}_2}{\Gamma \vdash \Pi\mathrm{X}.\mathrm{T}_1 \subseteq \Pi\mathrm{X}.\mathrm{T}_2} \qquad (\mathrm{Seq}_{\subseteq})\ \frac{\Gamma \vdash \mathrm{T}_1 \subseteq \mathrm{T}_3 \quad \Gamma \vdash_* \mathrm{T}_2 \subseteq \mathrm{T}_4}{\Gamma \vdash \mathrm{T}_1, \mathrm{T}_2 \subseteq \mathrm{T}_3, \mathrm{T}_4}$$

Les règles qui suivent sont introduites pour le typage de constructeurs et l'introduction du type `Empty` dans le cas de sous-typage de séquences de longueurs différentes. En fait ceci vient du fait que le calcul des premiers et suivants exploite la propriété d'élément neutre de ϵ sur la séquence.

$$(\mathrm{Tag}_{\subseteq})\ \frac{\mathrm{n}_1 =_{\mathcal{I}} \mathrm{n}_2 \quad \Gamma \vdash_r \mathrm{R}_1 \subseteq \mathrm{R}_2 \quad \Gamma \vdash_* \mathrm{T}_1 \subseteq \mathrm{T}_2}{\Gamma \vdash \langle\!\langle \mathrm{n}_1; \mathrm{R}_1; \mathrm{T}_1 \rangle\!\rangle \subseteq \langle\!\langle \mathrm{n}_2; \mathrm{R}_2; \mathrm{T}_2 \rangle\!\rangle}$$

$$(\mathrm{Seq}^{\epsilon^l}_{\subseteq})\ \frac{\mathrm{T}_1 \neq \mathrm{T}', \mathrm{T}'' \quad \Gamma \vdash \mathrm{T}_1, \epsilon \subseteq \mathrm{T}_2, \mathrm{T}_3}{\Gamma \vdash \mathrm{T}_1 \subseteq \mathrm{T}_2, \mathrm{T}_3} \qquad (\mathrm{Seq}^{\epsilon^r}_{\subseteq})\ \frac{\mathrm{T}_3 \neq \mathrm{T}', \mathrm{T}'' \quad \Gamma \vdash \mathrm{T}_1, \mathrm{T}_2 \subseteq \mathrm{T}_3, \epsilon}{\Gamma \vdash \mathrm{T}_1, \mathrm{T}_2 \subseteq \mathrm{T}_3}$$

La règle suivante est la règle proposée dans le *record calculus* [6] dans laquelle est ajoutée la notion d'attribut optionnel caractérisé par le sous-typage $\Gamma \vdash \epsilon \subseteq T'_j$ pour tout attribut non défini dans le sous enregistrement.

$$(\mathrm{Rec}_{\subseteq})\ \frac{\forall \mathrm{j} \in \mathrm{J}, (\exists \mathrm{i} \in \mathrm{I}/\mathrm{m}_\mathrm{i} = \mathrm{m}'_\mathrm{j} \wedge \Gamma \vdash_* \mathrm{T}_\mathrm{i} \subseteq \mathrm{T}'_\mathrm{j}) \vee (\nexists \mathrm{i} \in \mathrm{I}/\mathrm{m}_\mathrm{i} = \mathrm{m}'_\mathrm{j} \wedge \Gamma \vdash_* \epsilon \subseteq \mathrm{T}'_\mathrm{j})}{\Gamma \vdash_r \{\mathrm{m}_\mathrm{i} : \mathrm{T}_\mathrm{i}\}_\mathrm{I} \subseteq \{\mathrm{m}'_\mathrm{j} : \mathrm{T}'_\mathrm{j}\}_\mathrm{J}}$$

$$(\mathrm{Any}_{\subseteq})\ \frac{\mathrm{T} \neq \mathrm{T}', \mathrm{T}''}{\Gamma \vdash \mathrm{T} \subseteq \mathsf{Any}}$$

Il est important de noter que le type Any n'est pas le type principal. La spécification du type maximal est en fait Any*. Ceci découle de l'introduction des types polymorphes dans le langage des types. La dénotation du type `ANY` dans une DTD est en fait Any∗.

Par le jeu de règles précédant, nous pouvons montrer simplement l'équivalence de type $\mathrm{T}{*}{*} \approx \mathrm{T}{*}$ en montrant $\emptyset \vdash_* \mathrm{T}{*}{*} \subseteq \mathrm{T}{*}$ et $\emptyset \vdash_* \mathrm{T}{*} \subseteq \mathrm{T}{*}{*}$ comme suit :

$$\dfrac{\dfrac{}{\Gamma_0 \vdash \epsilon \subseteq \epsilon} \quad \dfrac{\dfrac{}{\Gamma_0 \vdash \mathrm{T} \subseteq \mathrm{T}} \quad \dfrac{\dfrac{}{\Gamma_1 \vdash \epsilon \subseteq \epsilon} \quad \dfrac{\dfrac{}{\Gamma_1 \vdash_* \mathrm{T} \subseteq \mathrm{T}} \quad \dfrac{}{\Gamma_1 \vdash_* \mathrm{T}{*}, \mathrm{T}{*}{*} \subseteq \mathrm{T}{*}}}{\Gamma_1 \vdash_* \mathrm{T}, (\mathrm{T}{*}, \mathrm{T}{*}{*}) \subseteq \mathrm{T}, \mathrm{T}{*}}}{\Gamma_0 \vdash_* \mathrm{T}{*}, \mathrm{T}{*}{*} \subseteq \mathrm{T}{*}}}{\Gamma_0 \vdash \mathrm{T}, (\mathrm{T}{*}, \mathrm{T}{*}{*}) \subseteq \mathrm{T}, \mathrm{T}{*}}}{\emptyset \vdash_* \mathrm{T}{*}{*} \subseteq \mathrm{T}{*}}$$

avec $\Gamma_0 = \{T{*}{*} \subseteq T{*}\}$ et $\Gamma_1 = \Gamma_0 \cup \{T{*}, T{*}{*} \subseteq T{*}\}$. La seconde preuve est similaire.

Un remarque importante concernant le système porte sur la nature du sous-typage entres constructeurs. En fait, cette relation repose entièrement sur l'aspect structurel des types et permet donc sous des dénominations de mettre en évidence des propriétés d'inclusions entre types sans qu'il soit nécessaire d'intervenir.

3.3. De l'instance XML au terme PaX

La plupart des langages événementiels dédiés à XML [20] [4] ne disposent pas de la notion de balise ce qui interdit toute vérification de types résultants. La notion de balise doit faire partie intégrante du langage ; ceci est rédhibitoire dans la mesure où un mécanisme de vérification de type doit être intégré.

Ce dernier point constituant en fait le principal défaut des langages cités précédemment, les nouvelles définitions de langage telles que XSL, XDuce, XML-QL, XMλ et PaX intègrent le balisage XML comme un constructeur de base et le possède donc au sein du langage des termes associés.

```
Instance XML                          terme PaX
<Document type='ARTICLE'              <Document { type="ARTICLE" }
 <Titre>Le titre du document</Titre>   <Titre "Le titre du document" >,
 <Resume>le résumé ...</Resume>        <Resume "le résumé ..." >,
 <Chapitre id='1'>                     <Chapitre { id="1" }
  <Titre>Chapitre 1</Titre>             <Titre "Chapitre 1" >,
  <Para>...</Para>                      <Para "..." >,
  <Section>                             <Section
   <Titre>Section 1</Chapitre>           <Titre "Section 1" >,
   <Para><Ref idref='1'/>...</Para>      <Para <Ref { idref="1"} > "..." >
  </Section>                            >
 </Chapitre>                           >
</Document>                           >
```

[Correspondance XML / terme PaX]

3.4. Manipulation de termes PaX : le filtrage de forme

Afin d'illustrer les différentes formes de filtrage nous allons définir la fonction permettant d'extraire les titres inclus dans un document. De façon générale le filtrage est la capacité à reconnaître un fragment et à lui associer un calcul. Ceci revient à la définition de règles de réécritures déclenchées par la reconnaissance d'une forme donnée. Ce mécanisme est beaucoup plus puissant que la programmation événementielle telle qu'elle est proposée et mise en oeuvre par la norme SAX car il permet de mettre en évidence un calcul exécutable lors de la reconnaissance non pas d'une simple balise mais plutôt d'un fragment complet.

Dans le cadre de PaX, le filtrage de forme opère par décomposition structurelle gauche. Ainsi le filtrage adopte un schéma classique et ne repose pas sur la reconnaissance par automate d'arbres tel que cela est proposé pour XDuce [14]. Ceci évite de poser la question de l'interprétation des formes étoilées et évite l'introduction d'un mécanisme de filtrage trop complexe.

3.4.1. Filtrage simple

Le premier exemple définit un itérateur sur une séquence homogène de données. Contrairement à SAX où la notion de séquence est en faite invisible au niveau événement, le filtrage impose le traitement de toutes les formes possibles caractérisées par le type. Cela permet au programmeur d'avoir une maîtrise du traitement lors de la reconnaissance. Ainsi contrairement aux langages fondés sur les événements il est possible de décomposer une séquence comme suit :

```
let rec seq_iter (f:X -> Y) (s:X*) : Y* =
   match s with
     <>  -> <>
   | h,t -> f h,seq_iter f t
   end
```

Le principe de filtrage dans PaX repose sur le dépliage de type vu précédemment. Donc intuitivement `h,t` représente la séquence avec `h` dénotant les types dits premiers ici `X` et `t` les types dits seconds ici `X*`.

En utilisant ce principe, nous pouvons écrire la fonction d'extraction exhaustive suivante :

```
let rec extraire_titres (e:DOC) : DOC.Titre* =
   match e with
```

```
  <Document b> -> seq_iter extraire_titres b
| <Chapitre b> -> seq_iter extraire_titres b
| <Section b>  -> seq_iter extraire_titres b
| <Resume _>   -> <>
| <Titre b>    -> <Titre b>
| <Para _>     -> <>
| <Ref>        -> <>
end
```

3.4.2. Filtrage générique

En fait, la fonction peut être plus simple du fait qu'un titre est toujours le premier à apparaître dans le corps de `Document`, `Chapitre` et `Section`. De cette propriété particulière nous pouvons établir la fonction suivante en ignorant la définition de la balise englobante :

```
let rec extraire_titres (e:DOC) : DOC.Titre* =
   match e with
     <_ <Titre t>,b> -> <Titre t>,seq_iter extraire_titres b
   | _               -> <>
   end
```

Cette dernière fonction impose toutefois une condition forte sur la structure même du document mise à part la propriété citée : une balise intitulée n'est pas visitée si la balise englobante ne possède pas de titre.

3.4.3. Filtrage par factorisation

Par factorisation on caractérise la capacité à regrouper un ensemble de filtrage similaire au nom de balise près comme le propose la fonction suivante :

```
let rec extraire_titres (e:DOC) : DOC.Titre* =
   match e with
     <(Document|Chapitre|Section|Ref) b> -> seq_iter extraire_titres b
   | <Titre b>                           -> <Titre b>
   | <Para|Resume _>                     -> <>
   end
```

Le type de `b` est alors la disjonction des types de contenus associés.

3.4.4. Filtrage par négation

Par filtrage 'négatif', nous caractérisons la fonction de sélection de balise dont le nom n'appartient pas à l'ensemble des noms spécifiés.

```
let rec extraire_titres (e:DOC) : DOC.Titre* =
   match e with
     <~(Titre|Para) b> -> seq_iter extraire_titres b
   | <Titre b>         -> <Titre b>
   | _                 -> <>
   end
```

La première règle permet de filtrer toutes les balises de type `Document`, `Chapitre` et `Section`. Le type du corps, ici `b`, est alors par synthèse la disjonction des types des contenus de ces différentes balises. Si les balises `Para` ne sont pas traitées à part comme suit :

```
let rec extraire_titres (e:DOC) : DOC.Titre* =
   match e with
     <~Titre b> -> seq_iter extraire_titres b
   | <Titre b>  -> <Titre b>
   | _          -> <>
   end
```

Ceci conduit à une erreur de typage car la fonction n'admet qu'un paramètre de type `DOC`. Or `Para` conduit à appliquer la fonction sur des données de type `string` ce qui est donc faux.

3.4.5. Filtrage par typage dynamique

Afin de résoudre le problème précédant il est possible d'introduire le typage au sein de filtres comme suit :

```
let rec extraire_titres (e:DOC) : DOC.Titre* =
   match e with
     <Titre b>    -> <Titre b>
   | <_ b:[DOC*]> -> seq_iter extraire_titres b
   | _            -> <>
   end
```

De ce filtrage et de l'utilisation du type Any nous pouvons définir des fonctions générales manipulant des termes sans qu'il soit nécessaire de connaître leur type par avance. Cette dernière possibilité est importante dans la mesure où certaines classes de documents peuvent être caractérisées non pas par un type mais plutôt par un traitement commun. C'est le cas de la plupart des documents XSL qui ne peuvent avoir une DTD exhaustive.

```
let rec extraire_titres (e:any) : <Titre string>* =
    match e with
      <Titre b>  -> <Titre b>
    | <_ b>      -> seq_iter extraire_titres b
    | _          -> <>
```

Dans ce cas le type de `b` est `any*` ce qui explique la présence de l'itérateur dans le corps de la deuxième règle de filtrage.

Par opposition au filtrage de XDuce un filtre de la forme `r:[DOC.Para*],b:[DOC.Section]` est rejeté. Ceci découle de la méthode adoptée pour le filtrage i.e. décomposition structurelle standard. Ce choix influe donc sur le principe de typage établissant des formes dites simples pour la variable `r` (balise, donnée primitive ou disjonction de ces deux types).

3.4.6. Filtrage et ouverture vers l'extérieur

Le filtrage dynamique permet d'introduire la partie validation de documents. Ainsi nous pouvons à partir d'un terme donné activer un ensemble de calculs relativement au type sélectionné.

```
let valider_document (d:any) : DOC.Document =
   match d with
     d:[DOC.Document] -> d
   | _                -> raise <Error d>
   end
```

Afin d'avoir une ouverture vers l'extérieur PaX dispose de deux mécanismes. Le premier est en fait un analyseur XML intégré permettant d'obtenir le terme PaX correspondant à un document (`read_xml_from_file : string -> any`) et le second est la possibilité d'avoir les types paramétriques. De l'utilisation de ces deux principes et du typage dynamique, la fonction validante d'un document peut être simplement décrite comme suit :

```
let read_and_valid [T] (s:string) : T =
   match read_xml_from_file s with
     d:[T] -> d
   | _     -> raise <Xml::InvalidDocumentType>
   end
```

Ainsi, contrairement à la plupart des outils XML, PaX ne dispose pas de processus validant à proprement parler mais plutôt du concept de typage dynamique permettant d'exprimer cette validation.

3.4.7. Filtrage et polymorphisme

Lors de l'introduction du polymorphisme comme principe de définition de types, une contrainte forte à été établie. Un type polymorphe ne peut être que spécialisé par une balise, un type prédéfini ou une disjonction de ces derniers.

En fait cela découle du principe opératoire même du filtrage ainsi que de la propriété d'associativité de la séquence. Afin d'illustrer la problématique, soit la fonction suivante :

```
let rec list_vers_seq (l:List[X]) : X* =
    match l with
      <nil>      -> <>
    | <cons x,l> -> x,list_vers_seq l
    end
```

Logiquement par l'écriture du filtre `<cons x,l>`, nous supposons que `x` sera de type `X`. Si `X` est spécialisé par le type `int,string` le terme `<cons (1,"k1"),<nil>>` est alors à priori acceptable. Ce terme est équivalent par associativité à `<cons 1,("k1",<nil>)>`. La difficulté consiste alors, durant l'exécution du filtrage, à reconnaître la forme relativement à la spécialisation de `X`. Ceci implique donc la propagation du type spécialisé mais aussi une phase systématique de reconnaissance par typage dynamique et donc une compilation inefficace. Face aux implications sur le principe de compilation la restriction forte est établie. De même le type ne être spécialisé par ϵ ; C'est l'élément neutre pour la séquence.

Par contre l'utilisation de constructeurs englobants permet de palier à ce défaut comme cela peut être fait pour la définition des listes associatives à partir des listes comme le propose l'exemple suivant :

```
type AList[X][Y] == List[<entry X,Y>]
```

3.5. Langage des termes et typage

Nous nous limitons dans la suite à l'étude du filtrage de forme.

$$
\begin{array}{lcll}
Te & = & \mathsf{match}\ Te\ \mathsf{with}\ \{Pe \to Te; \cdots\} & \text{(Filtrage)} \\
 & | & \cdots & \\
 & & & \\
Re & = & \{\mathcal{I} = Pe; \cdots; \mathcal{I} = Pe\} & \text{(ensemble d'attributs)} \\
 & & & \\
Pe & = & \langle\rangle & \text{(forme pour le vide)} \\
 & | & \mathcal{V}_P & \text{(donnée primitive)} \\
 & | & \langle\{\mathcal{I}\}; Re; Pe\rangle & \text{(forme de balise)} \\
 & | & \langle\neg\{\mathcal{I}\}; Re; Pe\rangle & \text{(forme négative de balise)} \\
 & | & \mathcal{V} & \text{(variable)} \\
 & | & \mathcal{V} : [Ty] & \text{(variable typée)} \\
 & | & Pe, Pe & \text{(forme séquentielle)}
\end{array}
$$

[Langage des termes]

Les règles pour le processus de typage sont classiques et sont pour la plupart présentées dans [21]. Celles-ci opèrent par induction sur les structures.

La règle de typage dédiée au filtrage reste cependant à expliciter.

3.5.1. Typage des filtres

Les règles suivantes introduisent le principe de typage du filtrage de forme. Le formalisme adopté est celui utilisé pour la définition du filtrage de XDuce où les prémisses et conclusions sont de la forme : $\Gamma \vdash_* t : T \Rightarrow \Gamma'$. L'ensemble Γ' contient lors de la terminaison du processus le type de chaque variable introduite par la forme. Avant toute chose il faut que tout filtre remplisse la condition dite de linéarité interdisant l'occurrence multiple d'une même variable dans un filtre.

La règle principale pour le typage du filtrage est la suivante :

$$(\mathrm{Match})\ \frac{\Gamma \vdash_* \mathrm{e} : \mathrm{T_e} \quad \forall \mathrm{i} \in \mathrm{I}, \Gamma \vdash_* \mathrm{m_i} : \mathrm{T_e} \Rightarrow \Gamma'_\mathrm{i} \wedge \Gamma'_\mathrm{i} \vdash_* \mathrm{b_i} : \mathrm{T_i}}{\Gamma \vdash_* \mathsf{match\ e\ with}\ \{\mathrm{m_i} \rightarrow \mathrm{b_i}\}_\mathrm{I} :\mid \{\mathrm{T_i}\}_\mathrm{I}}$$

Le typage du corps de chacune des règles est réalisé dans l'environnement Γ' calculé lors du typage du filtre associé. Cet ensemble d'hypothèses contient entre autres, le type des variables introduites par le filtre. Reste à exprimer la règle principale pour le typage d'une forme.

$$(\mathrm{Or}_{match})\ \frac{\forall \mathrm{T_i} \in \mathcal{D}[\![\mathcal{U}[\![\mathrm{T}]\!]_\emptyset]\!]_\emptyset,\ \Gamma_\mathrm{i} \vdash^{|} \mathrm{t} : \mathrm{T} \Rightarrow \Gamma'_\mathrm{i} \quad \bigoplus_\mathrm{I} \Gamma'_\mathrm{I} \neq \bot}{\Gamma_0 \vdash_* \mathrm{t} : \mathrm{T} \Rightarrow \bigoplus_\mathrm{I} \Gamma'_\mathrm{I}}$$

$$\text{avec } \oplus : \begin{cases} \Gamma \oplus \bot = \Gamma \\ \Gamma \oplus \emptyset = \Gamma \\ \Gamma \oplus (\{v : T\} \cup \Gamma') = (\Gamma/v \cup \{v : T \mid \Gamma(v)\}) \oplus \Gamma' & \mathsf{si}\ v \in \Gamma \\ \Gamma \oplus (\{v : T\} \cup \Gamma') = (\Gamma \cup \{v : T\}) \oplus \Gamma' & \mathsf{si}\ v \notin \Gamma \end{cases}$$

Le filtrage repose en fait sur le principe de dépliage par nécessité introduit pour le sous-typage. La règle (Or_{match}) exprime l'existence au moins d'un type acceptant la forme proposée. Si ce n'est pas le cas l'ensemble résultant est alors $\bot$. Ceci est exprimé par les deux règles suivantes qui stipulent la validité d'un filtre lors d'une alternative ($\vdash^{|}$)

$$(\mathrm{Succ}_{match})\ \frac{\Gamma \vdash \mathrm{t} : \mathrm{T} \Rightarrow \Gamma'}{\Gamma \vdash^{|} \mathrm{t} : \mathrm{T} \Rightarrow \Gamma'} \qquad (\mathrm{Fail}_{match})\ \frac{\Gamma \not\vdash \mathrm{t} : \mathrm{T} \Rightarrow \Gamma'}{\Gamma \vdash^{|} \mathrm{t} : \mathrm{T} \Rightarrow \bot}$$

L'enrichissement de Γ est alors effectué par le typage des variables de formes comme suit :

$$(\mathrm{Env}^1_{match})\ \frac{\mathrm{v} \in \mathcal{V}}{\Gamma \vdash \mathrm{v} : \mathrm{T} \Rightarrow \Gamma \oplus \{\mathrm{v} : \mathrm{T}\}} \qquad (\mathrm{Env}^2_{match})\ \frac{\Gamma \vdash_* \mathrm{T_1} \subseteq \mathrm{T_2}}{\Gamma \vdash \mathrm{v} : [\mathrm{T_1}] : \mathrm{T_2} \Rightarrow \Gamma \oplus \{\mathrm{v} : \mathrm{T_1}\}}$$

Le typage du filtrage de constructeur procède par décomposition structurelle. La règle suivante traite le cas du filtrage par négation.

$$(\mathrm{Tag}^{\neg}_{match})\ \frac{\mathrm{t} \notin \{\mathrm{t_i}\}_\mathrm{I} \quad \Gamma \vdash^\mathrm{a} \mathrm{a} : \mathrm{A} \Rightarrow \Gamma' \quad \Gamma' \vdash_* \mathrm{t} : \mathrm{T} \Rightarrow \Gamma''}{\Gamma \vdash \langle \neg\{\mathrm{t_i}\}_\mathrm{I}; \mathrm{a}; \mathrm{t}\rangle : \langle\!\langle \mathrm{t}; \mathrm{A}; \mathrm{T}\rangle\!\rangle \Rightarrow \Gamma''}$$

3.5.2. Exhaustivité du filtrage

L'exhaustivité du filtrage pourrait être déduite par le sous-typage tout comme le propose XDuce [14] et non durant la phase de compilation comme cela est proposé dans ZINC [18].

Intuitivement la fonction suivante est exhaustive si la disjonction des types des filtres est un sur-type du type de la donnée filtrée. Prenons pour cela la fonction suivante :

```
let rec combine (l1:X*) (l2:Y*) : AList[X][Y] =
   match <e l1>,<e l2> with
     <e>,_             -> <nil>
   | _,<e>             -> <nil>
   | <e k,lk>,<e v,lv> -> <cons <entry k,v>,combine lk lv>
   end
```

Cependant, la validation de l'exhaustivité ne peut être déduite par sous-typage. En effet, le jeu de règles ne peut valider le sous-typage suivant :

$$\emptyset \vdash_* \langle e; \{\}; X*\rangle, \langle e; \{\}; X*\rangle \subseteq \langle e; \{\}; \epsilon\rangle, \langle e; \{\}; X*\rangle \mid \langle e; \{\}; X*\rangle, \langle e; \{\}; \epsilon\rangle \mid \langle e; \{\}; X+\rangle, \langle e; \{\}; X+\rangle$$

Ceci vient du fait que le type du filtrage est indéterministe. Pour remédier à ce problème, nous pouvons soit étendre le principe de la validation par un mécanisme d'équivalence de langages par le biais des automates d'arbres [15] soit étendre la déterminisation par l'application de la distributivité de la disjonction sur les constructeurs à savoir $\langle e; \{\}; T_1 \mid T_2 \rangle =_{def} \langle e; \{\}; T_1 \rangle \mid \langle e; \{\}; T_2 \rangle$. Ceci reste cependant encore à étudier dans le cadre du langage PaX.

3.6. Décidabilité, Sûreté et Prototypage : état des lieux

Bien que ces propriétés ne soient pas vérifiées, le problème majeur concerne la décidabilité. En fait, le typage s'inspire du mécanisme décidable d'inclusion d'automates par bisimulation. Une première étape consisterait à montrer que l'inspiration en question est formelle et ne sort pas de ce cadre. Malheureusement ceci n'est pas si simple car la déterminisation ne tient pas compte actuellement des propriétés d'inclusions des balises. Ceci découle du fait que l'approche n'est pas bâtie sur les automates d'arbres. Concernant la sureté l'effort à fournir pour prouver la sûreté doit être fait essentiellement sur la validation du filtrage et plus particulièrement sur sa résolution.

Pratiquement, un prototype du langage existe actuellement et dispose, en plus des mécanismes présentés, d'un ensemble d'outils pour une intégration complète dans le monde XML tels que des translateurs de DTD vers les types PaX et un translateur PaX vers OCaml [9] pour l'aspect compilation. Le principal constat concerne la difficulté d'élaboration de programmes dès lors qu'ils reposent sur un fragment polymorphe. En terme de compilation, le filtrage n'a pour l'instant fait l'objet d'aucune étude particulière pour son optimisation. En fait, dans la version actuelle, cette opération est partiellement interprétée n'impliquant donc pas des temps d'exécution fulgurants. De plus cette opération reste encore trop complexe du fait de l'existence du terme $\langle\rangle$ et plus particulièrement lors de la décomposition de séquences.

4. Conclusion et perspectives

Comme nous avons pu le constater, actuellement un ensemble de travaux propose une approche formelle de la problématique XML et ce en terme de validation de processus de transformation. PaX s'inscrit dans cette mouvance.

De ce point de départ deux pistes peuvent être suivies. La première concerne l'intégration de la programmation objet. En fait, XDuce propose déjà une notion de sous-typage d'objets par la mise en place d'une relation d'ordre partiel sur les balises et donc un premier pas pour l'intégration de mécanismes d'héritage. En fait, l'approche XML permet d'avoir une vue de la notion d'objets mais avec une approche structurelle du typage et non nominative comme la plupart des langages à objets. La seconde piste concerne l'analyse et l'interprétation dans PaX de différentes propositions du consortium W3C comme le couple XSL et XPath et plus particulièrement le principe de schémas. En fait, Le consortium W3C tente actuellement de pallier au principal défaut introduit par les DTD à savoir la définition de spécification. Pour cela, le principe de schémas [23] est en cours de définition introduisant non plus une spécification mais plutôt un ensemble de contraintes sur la composition de documents. Ceci revient donc à affaiblir le principe de DTD mais permet aussi l'introduction de types plus précis tels que les entiers. Les types PaX devraient permettre d'exprimer de telles contraintes sans qu'une révision des principes présentés soit nécessaire.

Remerciements

Je tiens tout particulièrement à remercier Mr Jean-Paul Bodeveix, maître de conférence à l'IRIT, pour son soutient et ses critiques durant ce travail de rédaction.

Références

[1] A. Deutsh, M. Fernandez, D. Florescu, A Levy and D. Suciu. Xml-ql : A query language for xml. 1999.

[2] Alfred Aho, Ravi Sethi and Jeffrey Ullman. *Compilers.* Addison-Wiley, 1986.

[3] Arnaud Le Hors. Document object model (dom) level 3 core specification. *W3C Working Draft.* `http ://www.w3c.org/TR/2000/WD-DOM-Level-3-Cor-20000901`.

[4] Balise. `http ://www.us.balise.com`.

[5] Kim Bruce. Typing in object-oriented languages : Achieving expressibility and safety. Technical report, Department of Computing, Williams College, 1996.

[6] Luca Cardelli. A Semantique of Multiple Inheritance. *Information and Computation*, 76 :138–164, 1988.

[7] James Clark. http ://www.jclarck.com/sp/.

[8] D. Box, D. Ehnebuske, G. Kakivaya, A. Layman, N. Mendelsohn, H. F. Nielsen, S.Thatte and D. Winer. SOAP : Simple Object Access Protocol.

[9] Emmanuel Chailloux, Pascal Manoury and Bruno Pagano. *Objective Caml.* O'Reilly, 2000.

[10] Erik Meijer Mark Shields. Xmλ a functionnal language for constructing and manipulating xml documents. 2000.

[11] James Clark. Xsl transformations. *W3C Working Draft.* `http :/www.w3c.org/TR/xslt`.

[12] Haruo Hasoya and Benjamin C. Pierce. XDuce : A typed XML Processing Language. In *Third International Workshop on the Web and Databases (WEBDB2000)*, 2000.

[13] Haruo Hasoya, Jérôme Vouillon and Benjamin C. Pierce. Regular expression types for XML. In *International Conference on Functional Programming*, 2000.

[14] Haruo Hasoya and Benjamin C. Pierce. Tree Automata and Pattern Matching. In *The $25^t h$ Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, 2001.

[15] Hubert Common, Max Dauchet, Rémy Gilleron, Florent Jacquemard, Denis Lugiez, Sophie Tison and Marc Tommasi. *Tree Automata techniques and applications.* Draft book `http ://www.grappa.univ-lille3.fr/tata`.

[16] James Clark and Steve DeRose. *XSL Path language (XPath).* W3C Working Draft. `http :/www.w3c.org/TR/xpath`.

[17] Sophie Cluet, Claude Delobel, Jérôme Siméon and Katarzyna Smaga Your Mediators Need Data Conversion ! *ACM SIGMOD'98 International Conference on Management of Data*, Seattle, Washington, June 1998.

[18] Xavier Leroy. The ZINC experiment : an economical Implementation of the ML language. Technical Report 117, INRIA, 1991.

[19] Oscar Nierstrasz. Regular Types for Active Objects. In O. Nierstrasz and D. Tsichritzis, editors, *Object-Oriented Software Composition*, pages 99–121. Prentice Hall, 1995.

[20] Omnimark. `http ://www.omnimark.com`.

[21] P.-L. Curien and G. Ghelli. Coherence of subsumption, minimum typing and typecheking in F$\leq$. *Mathematical Structures in Computer Science*, 2 :55–91, 1992.

[22] SAX 2.0 : The simple API for XML. `http ://www.megginson.com/SAX/sax.html`.

[23] David C. Fallside. *XML Schema Part 0 : Primer* W3C Recommendation. `http ://www.w3.org/TR/xmlschema-0`

[24] Tim Bray, Jean Paoli and C.M. Sperberg-McQueen. *Extensible Markup Language (XML) 1.0.* W3C Recommendation. `http :/www.w3c.org/TR/1998/REC-xml-19980210`.

[25] Gabriel M. Kuper and Jérôme Siméon. Subsumption for XML Types. In *International Conference on Database Theory (ICDT'2001)*, January 2001, London, UK.

[26] Phillip Wadler. A formal semantics of patterns in XSLT. In *Markup Technologies*, 1999.