

Streaming XML

Alain Frisch –  – Projet Gallium

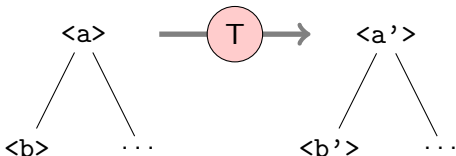


- Une syntaxe pour représenter des **données arborescentes**, de manière auto-descriptive.
 - Format de **stockage** : HTML, .doc, doc technique.
 - Format d'**échange** : EDI, protocoles (SOAP, Jabber).
- Les arbres ne sont pas nouveaux, mais le traitement de XML pose des problèmes intéressants de typage, compilation.
- Dans cet exposé : compilation de transformations XML en **streaming**.

- 1 Streaming XML : quoi ? Pourquoi ?
- 2 Pourquoi est-il difficile de faire du streaming ?
- 3 Comment faire du streaming quand même ?

Trois phases successives :

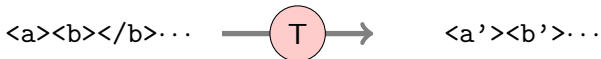
- **Charger** le document en mémoire, sous forme d'un arbre.
- Le **manipuler** (en fonctionnel ou en impératif).
- Produire un **nouveau document** modifié.



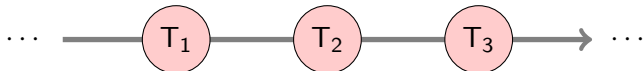
- Le document doit pouvoir tenir intégralement en **mémoire**.
- **Ressources sous-utilisées** : unités de calcul, canal de sortie.

↪ On voudrait **entrelacer** lecture, calcul, écriture.

- Principe : au lieu de transformer un arbre, on transforme un flux d'évènements séquentiels.
(évènements : ouverture/fermeture balise, texte)



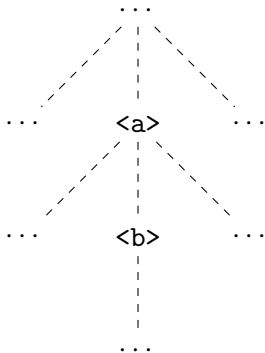
- Page web qui doit être transformée avant d'être affichée.
- Traitement distribué en pipeline.



- Extraction d'information d'un énorme document (doc technique Airbus).
- Conversion entre schémas (ex : renommer les tags).

- 1 Streaming XML : quoi ? Pourquoi ?
- 2 Pourquoi est-il difficile de faire du streaming ?**
- 3 Comment faire du streaming quand même ?

« Supprimer les éléments <a> sans sous-élément . »



```
if(true(),x,_) | if(false(),_,x) -> x  
or(true(),_) | or(_,true()) -> true()  
or(false(),x) | or(x,false()) -> x
```

```
hasb(b[_] _) -> true()  
hasb(%t[e1] e2) when t<>"b" -> or(hasb(e1),hasb(e2))  
hasb(()) -> false()
```

```
main(a[e1] e2) ->  
  if(hasb(e1),a[main(e1)] main(e2), main(e2))  
main(%t[e1] e2) when t<>"a" ->  
  %t[main(e1)] main(e2)  
main(()) -> ()
```

- Copier verbatim tant que l'on ne rencontre pas l'évènement `<a>`.
- Bufferiser les sous-arbres `<a>`. Conditions de sortie.
 - Évènement `` : copier le buffer.
 - Évènement ``, fin du sous-arbre : détruire le buffer.

... implémentation manuelle du streaming

```
type ev = Open of string | Close of string | End

let rec buffer buf bufs = match next() with
  | Open "b" as ev ->
      rev_iter (rev_iter out) ((ev::buf)::bufs)
  | Open "a" as ev -> buffer [ev] (buf::bufs)
  | Close "a" as ev ->
      (match bufs with
       | [] -> copy ()
       | buf::bufs -> buffer buf bufs)
  | ev -> buffer (ev::buf) bufs
and copy () = match next() with
  | End -> ()
  | Open "a" as ev -> buffer [ev] []
  | ev -> out ev; copy ()
```

- Programmation **à la main** : périlleuse et fragile, duplication de code, loin de la spécification fonctionnelle.
- Quelques prototypes pour aider le programmeur, mais ça reste quand même manuel, ou avec une expressivité réduite.

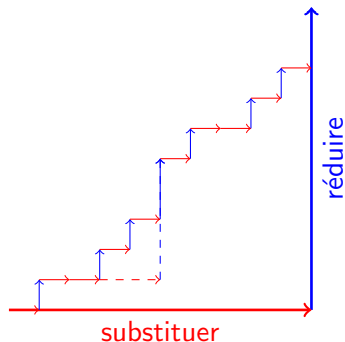
- 1 Streaming XML : quoi ? Pourquoi ?
- 2 Pourquoi est-il difficile de faire du streaming ?
- 3 Comment faire du streaming quand même ?**

Permettre au programmeur d'implémenter T dans un langage de **haut niveau** (fonctionnel, compositionnel) **sans se soucier du streaming** et laisser au compilateur le soin de produire du code qui :

- génère les évènements en sortie **dès que possible** ;
- garde le moins de choses en **mémoire** ;
- est **rapide**.

Idée de base : faire tourner directement la spécification fonctionnelle, vue comme un système de **réécriture de termes**.

- Termes \equiv données (entrées, sorties, résultats intermédiaires), calculs.
- x_0 : entrée. **Parser** \equiv **substituer** x_0 .
- $\tau = \text{main}(x_0)$: résultat à calculer. **Évaluer** \equiv **réduire** τ .



→ Modèle classique

→ Normalisation gloutonne

Exemple pas à pas

Entrée	Substitution	Pile de parsing	Terme	Sortie
<a>	$x_0 := a[x_1]x_2$	$x_0, []$	$m(x_0)$	
<c>	$x_1 := c[x_3]x_4$	$x_1, x_2, []$	$if(h(x_1), a[m(x_1)]m(x_2), m(x_2))$	
	$x_3 := b[x_5]x_6$	$x_3, x_4, x_2, []$	$if(or(h(x_3), h(x_4)), a[c[m(x_3)]m(x_4)]m(x_2), m(x_2))$	<a><c>
	$x_5 := ()$	$x_5, x_6, x_4, x_2, []$	$a[c[b[m(x_5)]m(x_6)]m(x_4)]m(x_2)$	
</c>	$x_6 := ()$	$x_6, x_4, x_2, []$	$a[c[b[]m(x_6)]m(x_4)]m(x_2)$	
	$x_4 := ()$	$x_4, x_2, []$	$a[c[b[]]m(x_4)]m(x_2)$	</c>
<a>	$x_4 := a[x_7]x_8$	$x_7, x_8, x_2, []$	$a[c[b[]]if(h(x_7), a[m(x_7)]m(x_8), m(x_8))]m(x_2)$	
	$x_7 := ()$	$x_7, x_8, x_2, []$	$a[c[b[]]m(x_8)]m(x_2)$	
	$x_8 := ()$	$x_8, x_2, []$	$a[c[b[]]]m(x_2)$	
</>	$x_2 := ()$	$x_2, []$	$a[c[b[]]]()$	</>

```
if(true(),x,_) | if(false(),_,x) -> x
or(true(),_) | or(_,true()) -> true()
or(false(),x) | or(x,false()) -> x
```

```
h(b[_] _) -> true()
h(%t[e1] e2) when t <> "b" -> or(h(e1),h(e2))
h(()) -> false()
```

```
m(a[e1] e2) -> if(h(e1),a[m(e1)] m(e2), m(e2))
m(%t[e1] e2) when t <> "a" -> %t[m(e1)] m(e2)
m(()) -> ()
```

Résultats expérimentaux : vitesse

Résultats en Mb/s.

input size :	1Mb	2Mb	5Mb	10Mb	20Mb	40Mb	80Mb	160Mb	320Mb
XStream (OCaml)	4.45	5.30	6.12	6.01	6.25	6.47	7.24	7.23	7.25
CDuce (OCaml)	3.38	3.45	4.43	4.24	4.06	3.74	3.35	2.61	*
Saxon (Java,XQuery)	0.31	0.63	1.03	1.25	1.41	1.51	1.58	*	*
Qizx (Java,XQuery)	0.51	0.76	1.21	1.30	1.43	1.50	1.53	*	*
XSLTC (Java,XSLT)	0.85	1.41	2.48	3.05	3.41	3.18	2.72	0.88	*
Saxon (Java,XSLT)	0.42	0.83	1.53	1.94	2.29	2.55	2.76	*	*
xsltproc (C,XSLT)	2.13	2.54	3.43	2.47	1.59	1.04	0.60	*	*
Xalan (C++,XSLT)	1.07	1.23	1.43	1.19	1.08	0.68	0.46	0.24	*
Joost (Java,STX)	0.35	0.39	0.51	0.53	0.55	0.54	0.57	0.56	0.60

Même pour de petits documents, le streaming est utile.

STX : langage de streaming de bas niveau.

Mémoire maximale utilisée, résultats en Mb.

input size :	10Mb	20Mb	40Mb	80Mb
XStream	1.0	1.0	1.0	1.0
CDuce	38.7	69.5	165.8	348.6
Saxon (xquery)	76.8	134.1	258.8	499.8
Qizx (xquery)	65.8	112.4	205.9	398.7
XSLTC (xslt)	57.3	102.3	215.0	440.8
Saxon (xslt)	79.9	134.7	249.3	496.9
xsltproc (xslt)	109.5	216.3	430.6	852.8
Xalan (xslt)	42.1	77.9	149.4	290.2
Joost (STX)	16.7	16.7	16.7	16.7

- **Sémantique** : formalisme bien connu, utile pour raisonner.
- **Correction du schéma d'évaluation** en streaming : simples lemmes de commutation normalisation \leftrightarrow substitution.
- **Implémentation efficace** de la normalisation incrémentale.
 - Représentation DAG, pointeurs vers ancêtres.
 - Ensemble de nœuds suspects (redex possible).
 - Évaluateur fonctionnel + suspension.
- XStream peut servir de langage cible pour XSLT, CDuce, OCamlDuce, ...
- Lien avec évaluation paresseuse.

Merci !

- Compilateur XStream diffusé sous licence CeCILL.
Ordre supérieur fonctionnel, extensions Caml
<http://gallium.inria.fr/~frisch/xstream>