Risoluzione efficiente di interrogazioni XPath su documenti XML con attributi e riferimenti

Enrico Zimuel

enrico@zimuel.it

Università degli Studi ”G.D’Annunzio” Chieti - Pescara

Sommario

- Come implementare un elaboratore XPath?
- Elaborazione efficiente di Core XPath e XPatterns
- Il nostro engine
- Sintassi SXPath e EXPath
- Il modello dei dati
- Semantica EXPath e SXPath
- L'algoritmo (tecnica del *pre*/*post* e del *flag* numerico)
- La complessità computazionale
- Conclusioni

XPath 1.0

- **XPath** è un linguaggio di interrogazione per documenti XML che consente la ricerca di elementi XML a partire da una struttura ad albero del documento;
- In questa presentazione faremo riferimento alla versione **1.0** di XPath definita dal consorzio W3C
- Alcuni esempi di interrogazioni XPath:
  `/descendant::a/child::b`
  (ricerca tutti gli elementi etichettati con **a** che hanno come figlio un elemento etichettato con **b**)
  `/descendant::a[attribute::nome='Enrico']`
  (ricerca tutti gli elementi etichettati con **a** che hanno un attributo **nome** con valore uguale a **Enrico**).

Un elaboratore XPath naif (1)

- Pensiamo ad un documento XML come ad un **albero radicato etichettato**
- Una query XPath è un insieme (lista) di *location-step* (al momento non consideriamo gli operatori filtri [])
- Dato un nodo iniziale, tipicamente *root*, possiamo elaborare il primo *location-step* della query XPath su questo nodo ottenendo, come risultato, un insieme di nodi *S*
- Reiterando il procedimento per ogni nodo dell'insieme *S* è possibile ottenere l'elaborazione di tutta la query XPath
- Che complessità computazionale ha un simile algoritmo?

Un elaboratore XPath naif (2)

- Sia **Q** un'interrogazione XPath, $\mathbf{n}_0$ il nodo da elaborare (*context node*) e **T** l'albero del documento XML

ProcessLocationStep (n_0,Q,R)

```
1    S ← Process(Q.first,n0);
2    if (Q.next ≠ NULL) and (S ≠ NULL) then
3        foreach n ∈ S do
4            ProcessLocationStep(n,Q.next,R);
5        endfch
6    endif
7    R ← R ∪ S;
```

- Se $n=|T|$ e $k=|Q|$ ed ipotizzando che *Process()* ha complessità *O(n)* allora la complessità computazionale di *ProcessLocationStep()*, nel caso pessimo, è **O(n^k)**

Un elaboratore XPath naif (3)

- Il risultato precedente sulla complessità dell'algoritmo è stato calcolato utilizzando la seguente formula ricorsiva:

$$Time(k) = \begin{cases} n + n * Time(k-1) = & \\ n * Time(k-1) & se \quad k > 0 \\ 1 & se \quad k = 0 \end{cases}$$

- G. Gottlob et al. [2002] hanno dimostrato che alcune implementazioni di XPath (XALAN, XT e Internet Explorer 6) hanno dei tempi di calcolo esponenziale.

Un elaboratore XPath naif (4)

- E' possibile migliorare il risultato precedente?
- Proviamo ad evitare la ricorsione: avendo a disposizione un albero XML in memoria (*memory tree*) possiamo procedere analizzando uno *step* per volta, in un'interrogazione XPath, memorizzando temporaneamente il risultato, rappresentato da **insieme di nodi**, in una lista **C**
- All'i-esimo *step* elaboriamo i nodi presenti nella lista C_{i-1} e memorizziamo il risultato nella lista C_i
- All'inizio lo step 0 sarà costituito dall'insieme $C_0 = \{root\}$ ossia dalla sola radice dell'albero XML

Un elaboratore XPath naif (5)

- Supponiamo di avere **T** un albero XML e **Q** un'interrogazione XPath

ProcessXPath (T,Q)

```
1   C ← {root};
2   foreach step ∈ Q do
3       S ← NewList();
4       foreach c ∈ C do
5           ProcessStep(T,c,step,S);
6       endfch
7       C ← S;
8   endfch
9   return C;
```

- Se $n=|T|$ e con $k=|Q|$ ed ipotizzando che *ProcessStep()* ha complessità *O(n)* allora la complessità computazionale di *ProcessXPath()*, nel caso pessimo, è $\mathbf{O(k \cdot n^2)}$

Un elaboratore XPath naif (6)

- Se consideriamo la presenza degli operatori filtri, nelle interrogazioni XPath, il risultato precedente non è più valido.
- Un'interrogazione XPath del tipo `/axis::a[filter]` presuppone l'elaborazione dello step `/axis::a` e successivamente per ogni nodo risultato la verifica dell'operatore `filter`.
- Questo modo di procedere genera una complessità esponenziale in funzione del grado r di ricorsione degli operatori filtri `/axis::`a_1`[axis::`a_2`[...[axis::`a_r`]]...]`.
- Si può dimostrare che la complessità computazionale di un elaboratore XPath naif risulta essere $O(k \cdot n^{2r+2})$ (M. Franceschet, E.Zimuel [2005])

Core XPath (G. Gottlob et al. [2002])

- Consideriamo un sottoinsieme di XPath indicato con il nome di **Core XPath**
- La sintassi di Core XPath:

exp	=	locationpath | /locationpath
locationpath	=	locationstep (/locationstep)*
locationstep	=	χ :: t | χ :: t [pred]
pred	=	pred and pred | pred or pred | not(pred) | exp | (pred)
χ	=	self | child | parent | descendant | descendant-or-self | ancestor | ancestor-or-self | following | following-sibling | preceding | preceding-sibling

- G.Gottlob et al. hanno dimostrato che è possibile risolvere con complessità $O(|q| \cdot |D|)$ un'interrogazione $q \in$ Core XPath su D un documento XML (Teorema 8.5 dell'articolo di G.Gottlob et al. [2002]).

XPatterns (G. Gottlob et al. [2002])

- Partendo dal linguaggio Core XPath ed introducendo la funzione id() di XPath si ottiene un nuovo linguaggio indicato da G.Gottlob et al. con il nome di **XPatterns**.
- L'utilizzo della funzione XPath **id()** consente di 'saltare' all'interno del documento XML in un nodo prestabilito trasformando la struttura classica ad *albero* di un documento XML in una struttura a *grafo*.
- Si è dimostrato che è possibile risolvere con la stessa complessità computazionale di Core XPath, ossia $O(|q| \cdot |D|)$, interrogazioni $q \in$ XPatterns su D un documento XML (Teorema 8.8 dell'articolo di G.Gottlob et al. [2002]).

SXPath

- Il nostro obiettivo è quello di estendere il linguaggio XPatterns introducendo la gestione degli *attributi* e dell'operatore di uguaglianza all'interno dei filtri.
- Esempio: `/descendant::a[attribute::c='100']`, in questo caso l'operatore filtro è `[attribute::c='100']`
- Indichiamo questo nuovo linguaggio con **SXPath** (Standard XPath), risulta che Core XPath $\subset$ XPatterns $\subset$ SXPath $\subset$ XPath

Sintassi SXPath

Sia Σ l'insieme delle etichette (*tag*) relative agli elementi e agli attributi di un documento XML. Un'interrogazione SXPath è una formula (query) generata dalla prima clausola della seguente definizione ricorsiva:

query	=	/path
path	=	step$(/step)^*$ | pointer$(/step)^*$ | pointer[filter]$(/step)^*$
pointer	=	id('s') | id(path)
step	=	axis::a | axis::a[filter]
filter	=	path | filter = 's' | filter and filter | filter or filter | not filter
axis	=	self | attribute | child | parent | descendant | descendant-or-self | ancestor | ancestor-or-self | following | following-sibling | preceding | preceding-sibling

dove a $\in \Sigma \cup \{*\}$ e s $\in$ String, l'insieme delle stringhe alfanumeriche.

Il nostro engine SXPath (1)

- Abbiamo visto che la complessità di un elaboratore naif di interrogazioni XPath è al più $O(k \cdot n^{2r+2})$. Il termine quadratico n^2 deriva dal fatto che si effettua un'elaborazione con costo *O(n)* per ogni nodo dell'insieme *C*, risultato dello *step* precedente, il tutto al più *n* volte.
- Una prima idea è quella di ridurre questo termine quadratico elaborando, in un unico passo, tutto l'insieme *C* senza dover valutare un nodo per volta, in modo da ottenere una complessità al più *O(n)* e non più *O(*n^2*)*.
- Nel caso di query con la presenza dell'operatore filtro, come possiamo ridurre il fattore $O(n^r)$?

Il nostro engine SXPath (2)

- L'idea per l'elaborazione dei filtri è di tradurre un'interrogazione SXPath `/q[p]` in un'altra interrogazione SXPath, $/q \cap p^{-1}$ (G. Gottlob et al. [2002])
- L'operatore *intersezione* $\cap$ non è presente nel linguaggio SXPath per cui al posto di implementarlo all'interno dell'engine abbiamo pensato di estendere il linguaggio SXPath in un nuovo linguaggio **EXPath** (Extended XPath)

Sintassi EXPath (1)

Un'interrogazione EXPath è una formula (query) generata dalla prima clausola della seguente definizione ricorsiva:

query	=	/path
path	=	step $(/step)^*$ | id('s')$(/step)^*$ | id('s')[path]$(/step)^*$ |
		id('s_1')='s_2'$(/step)^*$ | id('s_1')[path]='s_2'$(/step)^*$ |
		path and path | **path or path** | **not path**
step	=	axis::a | axis::a[path] | **axis::a='s'** | **axis::a[path]='s'**
axis	=	self | child | parent | **self-attribute** | attribute | **parent-attribute** |
		descendant | descendant-or-self | ancestor | ancestor-or-self |
		following | following-sibling | preceding | preceding-sibling |
		next | **next-sibling** | **previous** | **previous-sibling** | **id** | **id**$^{-1}$

In **neretto** sono evidenziate le novità rispetto a SXPath.

EXPath vs. SXPath (1)

- Presenza degli *operatori logici* sui percorsi (*path*) che consentono di scrivere, ad esempio, interrogazioni del tipo $/q_1$ `and` p_1/q_2 `or` p_2`/not(`p_3`)`
- L'operatore d'uguaglianza all'interno degli *step*, `axis::a='s'`, `axis::a[path]='s'`
- La trasformazione dell'operatore *id(path)* nell'asse *id*, `id(path)=/path/id::*`
- La presenza dei nuovi assi *self-attribute*, *parent-attribute*, *next* (il primo following), *next-sibling* (il fratello successivo), *previous* (il primo preceding), *previous-sibling* (il fratello precedente)

EXPath vs. SXPath (2)

- La presenza dei nuovi assi consente di ottenere un linguaggio più simmetrico rispetto a SXPath e quindi anche allo standard XPath dove, ad esempio, un nodo attributo non è figlio di un nodo elemento ma il padre di un nodo attributo è un nodo elemento (vedremo successivamente che in EXPath ogni asse ha un suo inverso).
- Il linguaggio EXPath è un linguaggio più potente di SXPath, ed in alcuni casi anche di XPath, ad esempio la query `/descendant::*/attribute::*='s'` che consente di ottenere tutti gli attributi di un documento XML che hanno il valore uguale alla stringa `s` non può essere tradotta in XPath.

Da SXPath a EXPath

- Per elaborare un'interrogazione in SXPath introduciamo una funzione di traduzione $\phi : SXPath \mapsto EXPath$ che traduce un'interrogazione SXPath in un'interrogazione EXPath
- Con l'utilizzo di questa funzione di traduzione ϕ è possibile risolvere interrogazioni SXPath utilizzando un engine per EXPath
- E' possibile dimostrare che la funzione di traduzione ϕ ha una complessità computazionale lineare nella dimensione dell'interrogazione SXPath
- Dunque il nostro obiettivo è quello di implementare un algoritmo efficiente per EXPath

Il modello dei dati (1)

- Rappresentiamo un documento XML come un albero radicato etichettato con elementi (*tag*) , attributi e relativi valori di testo
- Ogni nodo dell'albero rappresenta un *elemento*, un *attributo* o un *testo* del documento XML. Per *testo* intendiamo le stringhe associate agli elementi o agli attributi di un documento XML
- L'albero è costruito rispettando l'ordine degli elementi nel documento XML (*document order*), in particolare applicando l'algoritmo di visita anticipata sui nodi dell'albero si riottiene il documento XML originale

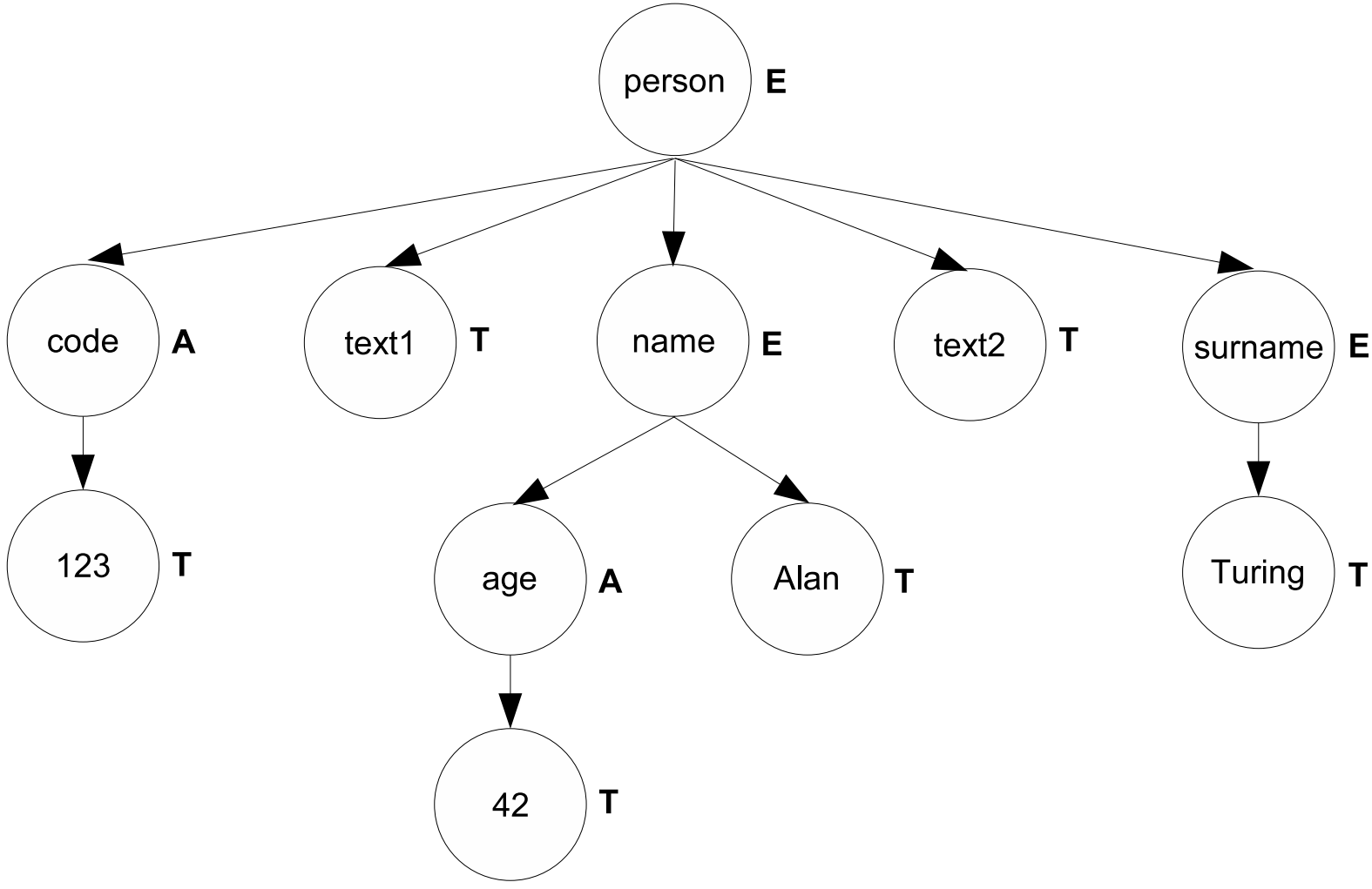
Il modello dei dati (2)

- I nodi di tipo attributo, di un nodo elemento, vengono inseriti nell'albero prima dei nodi figli di tipo elemento, l'ordine di inserimento dei nodi attributo non è rilevante
- I nodi di tipo testo vengono inseriti nell'albero nello stesso ordine in cui compaiono nel documento rispetto ai nodi figli
- Esempio, consideriamo il seguente documento XML:

```
<person code='123'>
    text1
    <name age='42'>Alan</name>
    text2
    <surname>Turing</surname>
</person>
```

Il modello dei dati (3)

- Esempio, rappresentazione tramite un albero radicato etichettato:



Semantica EXPath

- Sia T un albero XML e $root \in N$ la radice dell'albero T. Definiamo la semantica di EXPath attraverso la funzione $f(T, q, C) \mapsto 2^N$ dove q è l'interrogazione (query) EXPath e $C \subseteq N$ è un sottoinsieme dei nodi dell'albero T nel quale applicare l'interrogazione q
- Ad esempio $f(T, axis :: a, C)$ = $\{n \in N \mid n \in [C]_{axis,T} \wedge n \in L(a)\}$ dove $L(a)$ è l'insieme dei nodi etichettati con a
- $[C]_{axis,T} : C \times axis \times T \mapsto 2^N$ è la funzione che elabora l'asse ($axis$) a partire dall'insieme $C \subseteq N$
- Ad esempio $[C]_{child,T}$ = $\{n \in N \mid \exists c \in C.\ (c, n) \in R_{\downarrow} \wedge type(n) = element\}$ dove $R_{\downarrow}$ è l'insieme delle coppie di nodi (padre,figlio).

Relazioni tra assi e loro inversi

- Sia χ un asse EXPath. Per ogni coppia di nodi n,n' $\in N$, $n\ \chi\ n$' se e solo se n' $\chi^{-1}\ n$
- Ad esempio $n\ child\ n$' se e solo se n' $parent\ n$, ossia n è figlio di n' se e solo se n' è genitore di n
- In particolare si hanno le seguenti relazioni: self^{-1} = self, child^{-1} = parent, self-attribute^{-1} = self-attribute, attribute^{-1} = parent-attribute, descendant^{-1} = ancestor, descendant-or-self^{-1} = ancestor-or-self, following-sibling^{-1} = preceding-sibling, following^{-1} = preceding, next^{-1} = previous, next-sibling^{-1} = previous-sibling, (id^{-1})$^{-1}$ = id

Semantica SXPath

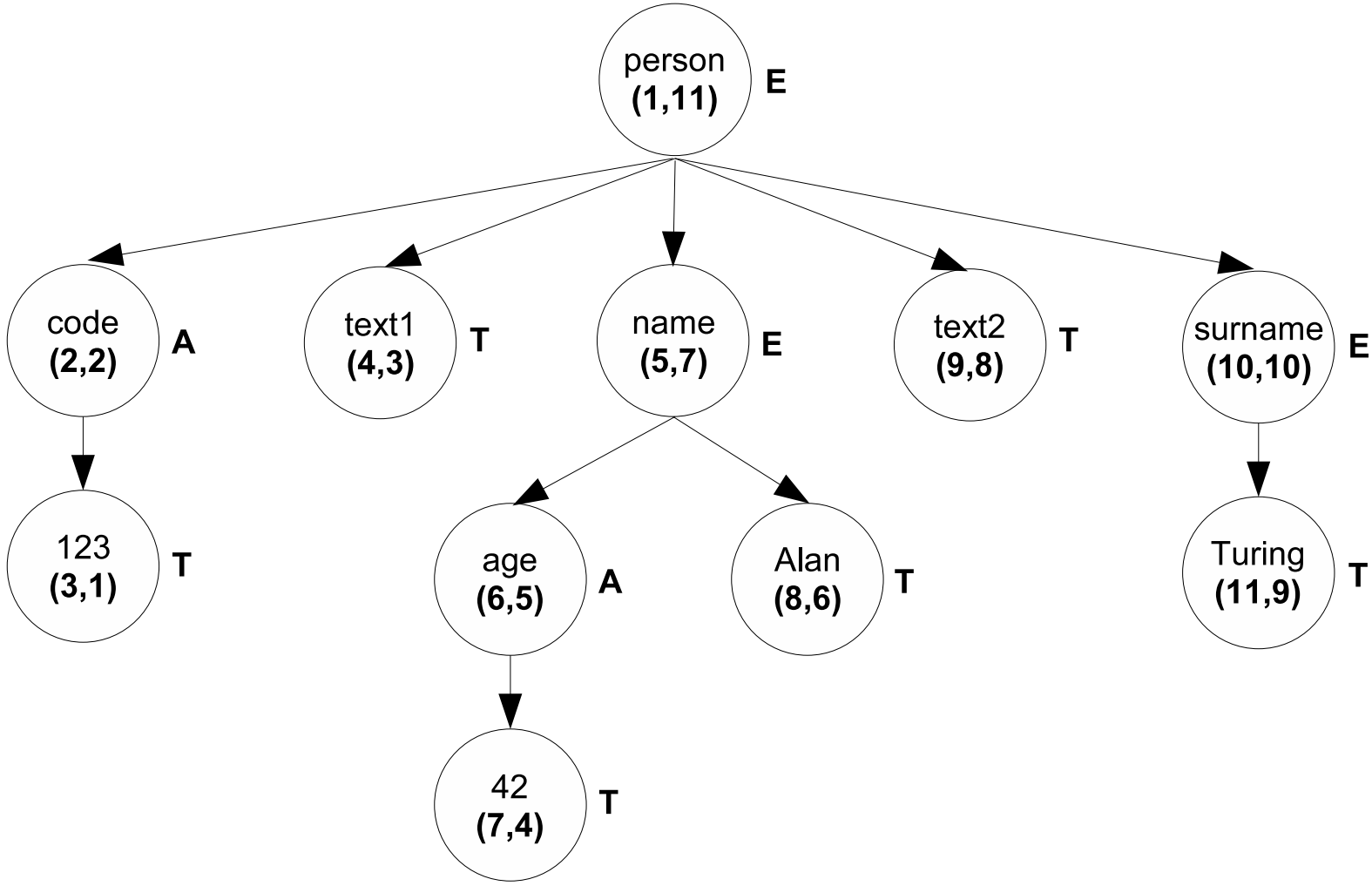
- Sia T un albero XML. Sia q un'interrogazione in SXPath e $\phi(q) : SXPath \mapsto EXPath$ la funzione di traduzione da SXPath a EXPath. Definiamo la semantica di SXPath attraverso la funzione $f(T, \phi(q), C) \mapsto 2^N$ definita, in precedenza, nella semantica di EXPath, dove $C \subseteq N$ è un sottoinsieme dei nodi dell'albero T nel quale applicare l'interrogazione $\phi(q)$
- In pratica definiamo la semantica del linguaggio SXPath attraverso la semantica del linguaggio EXPath utilizzando la funzione di traduzione $\phi(q) : SXPath \mapsto EXPath$

Calcolo dei valori *pre*/*post* (1)

- Per poter risolvere in maniera efficiente l'elaborazione di una query in XPath sono state presentate, in questi ultimi anni, diverse tecniche basate principalmente sull'utilizzo dei valori di *pre*/*post*.
- Questi valori di *pre*/*post* sono determinati dall'ordine della visita *anticipata* (*pre*) e *posticipata* (*post*) dei nodi di un albero
- Questi valori possono essere utilizzati per velocizzare l'elaborazione delle interrogazioni XPath soprattutto nel calcolo di alcuni assi (T.Grust et. al. [2002,2003], J.Hidders et. al. [2004]).

Calcolo dei valori *pre*/*post* (2)

- Esempio di albero XML con valori *pre*/*post* calcolati per ogni nodo:

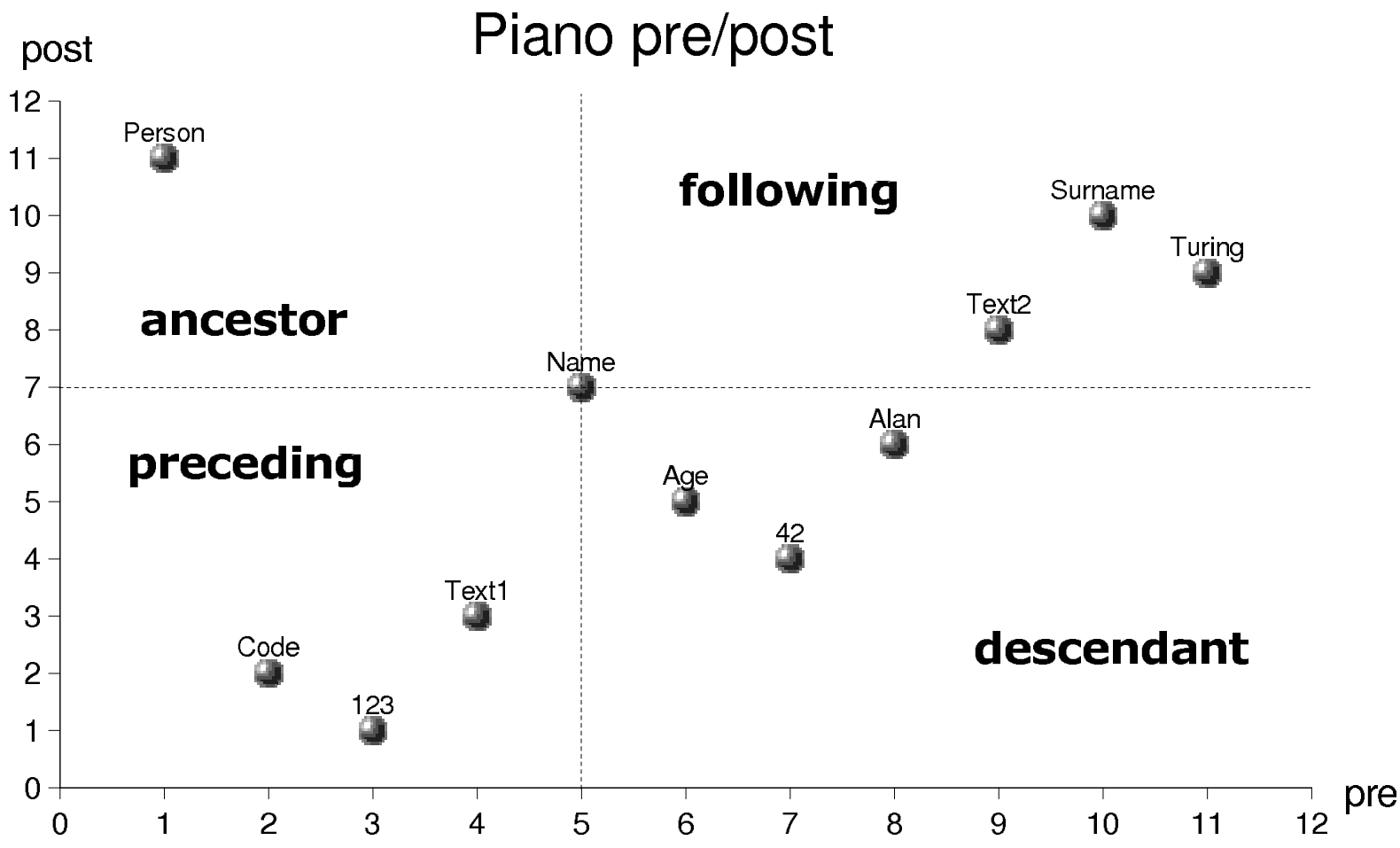


Calcolo dei valori *pre*/*post* (3)

- Una volta determinati i valori (*pre*, *post*) per ogni nodo dell'albero è possibile utilizzare questa coppia di numeri per rappresentare i nodi su di un **piano cartesiano** dove sull'asse delle ascisse vengono riportati i valori di *pre* e sull'asse delle ordinate i valori di *post*
- Osservando la posizione dei nodi all'interno di questo piano *pre*/*post* si possono trarre numerose informazioni utili per velocizzare l'elaborazione di alcuni assi XPath.

Piano *pre*/*post* (1)

- Esempio di piano *pre*/*post* per il precedente albero XML:

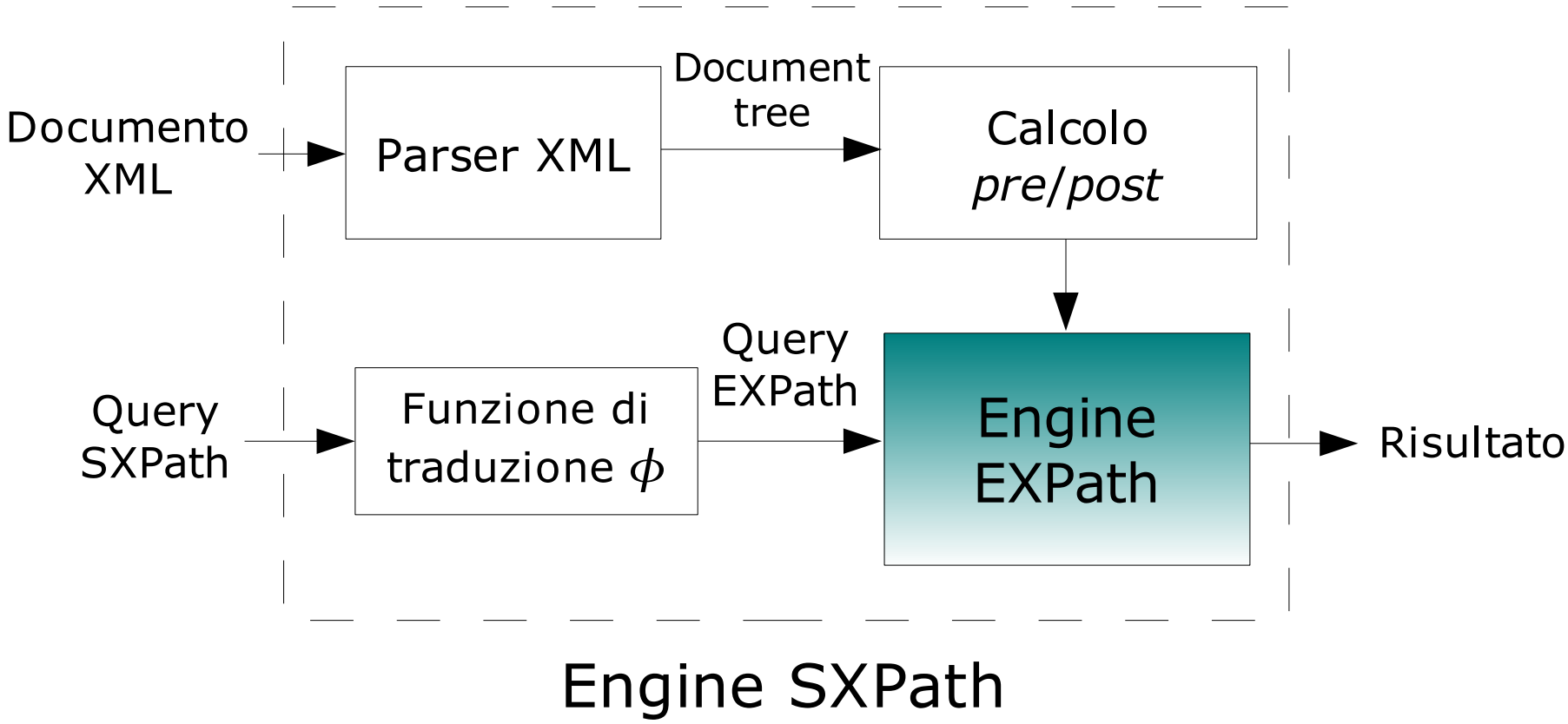


Piano *pre*/*post* (2)

- Considerando il nodo **Name**, nel piano precedente, con valori di (*pre*,*post*) pari a (5,7) si possono determinare i suoi nodi discendenti (*descendant*), antenati (*ancestor*), successori (*following*) e precedenti (*preceding*) suddividendo semplicemente il piano in quattro regioni
- In particolare per ogni nodo *n* si possono elaborare gli assi (*axes*) *descendant*, *ancestor*, *following* e *preceding* tramite un semplice confronto dei valori *pre*/*post* sui restanti nodi *x* dell'albero utilizzando la regola seguente:
  - se $pre(x) > pre(n)$ e $post(x) > post(n)$ allora *x* è *following* di *n*
  - se $pre(x) > pre(n)$ e $post(x) < post(n)$ allora *x* è *descendant* di *n*
  - se $pre(x) < pre(n)$ e $post(x) < post(n)$ allora *x* è *preceding* di *n*
  - se $pre(x) < pre(n)$ e $post(x) > post(n)$ allora *x* è *ancestor* di *n*

Schema a blocchi engine SXPath

- Schema a blocchi del nostro engine SXPath/EXPath:



La tecnica del *flag* numerico (1)

- Ipotizziamo di utilizzare un *flag* booleano (vero/falso) per marcare il passagio in un nodo. Questo *flag* potrebbe essere utilizzato per impedire di rielaborare nodi che sono stati già elaborati consentendo di risolvere l'elaborazione di uno step EXPath su un insieme di nodi in un tempo *O(n)*.
- Alla fine dell'elaborazione di ogni step è necessario riazzerare il valore del *flag* per tutti i nodi dell'albero. Tale operazione ha complessità *O(n)*.
- Utilizzando un flag booleano la complessità dell'elaborazione di ogni step risulta essere *O(2n)*
- L'idea è quella di utilizzare un *flag* con valore di tipo *numerico* per ogni nodo al posto di un valore di tipo booleano.

La tecnica del *flag* numerico (2)

- Durante l'elaborazione utilizziamo un contatore (denominato con *CONT*) che verrà incrementato ad ogni *step* dell'interrogazione ed il cui valore verrà utilizzato per marcare il *flag* del nodo appena elaborato.
- Inizialmente il valore del *flag* per tutti i nodi è posto uguale a zero.
- Per l'elaborazione di un singolo *step*, dato un nodo *n* dell'albero XML, è sufficiente verificare che *flag(n)* sia minore (oppure diverso) da *CONT* per poter stabilire che il nodo *n* non sia stato ancora elaborato.
- Questa tecnica consente di evitare di dover riazzerare ad ogni *step* i valori del *flag* per tutti i nodi dell'albero XML con conseguente risparmio di un fattore *O(n)* ad ogni step.

L'algoritmo (1)

- Nel nostro algoritmo utilizziamo sia la tecnica del *pre*/*post* che quella del *flag* numerico.
- Consideriamo insiemi di nodi ordinati secondo l'ordine del documento (*document order*) e senza duplicati
- Il nostro engine ha come *input* un documento XML e un'interrogazione SXPath e come *output* un insieme ordinato di nodi e senza duplicati

L’algoritmo (2)

- Per la memorizzazione degli insiemi di nodi dell’albero XML utilizziamo una lista tramite le seguenti funzioni elementari:
  - NewList(); inizializza una nuova lista
  - DelFirst(C); restituisce ed elimina il primo elemento della lista C
  - DelLast(C); restituisce ed elimina l’ultimo elemento della lista C
  - AddAfter(C,n); aggiunge l’elemento n alla fine della lista C
  - AddListAfter(C,L); aggiunge la lista L alla fine della lista C
  - AddBefore(C,n); aggiunge l’elemento n all’inizio della lista C
  - AddListBefore(C,L); aggiunge la lista L all’inizio della lista C
  - First(C); restituisce il primo elemento della lista C
  - Last(C); restituisce l’ultimo elemento della lista C

L'algoritmo (3)

- Per l'elaborazione degli assi XPath utilizziamo le seguenti funzioni elementari sull'albero XML:
  - **first-child(n,type);** restituisce il primo figlio (a partire da sinistra) del nodo *n* di tipo *type*
  - **right-sibling(n,type);** restituisce il fratello destro del nodo *n* di tipo *type*
  - **left-sibling(n,type);** restituisce il fratello sinistro del nodo *n* di tipo *type*
  - **parent-node(n,type);** restituisce il nodo padre del nodo *n* di tipo *type*

  dove *type* $\in \{all, element, attribute\}$. Il parametro type consente quindi di ottenere nodi di tipo elemento (*element*), attributo (*attribute*) o di qualsiasi tipo (*all*).

Elaborazione dello *string-value* (1)

- Siamo interessati ad elaborare step del tipo `axis::a='s'`
- Secondo le specifiche XPath del consorzio W3C lo *string-value* di un nodo *elemento n* è la concatenazione delle stringhe associate ai nodi di tipo testo discendenti del nodo *n*; nel caso di nodo *attributo* lo *string-value* è semplicemente il valore (normalizzato) della stringa associata all'attributo.
- Per poter effettuare l'elaborazione dello *string-value* dei discendenti di un nodo *n* di tipo elemento introduciamo una procedura *AllText(n,value)* che restituisce nella variabile *value* la concatenazione di tutte le stringhe associate ai nodi di tipo testo discendenti del nodo *n*.

Elaborazione dello *string-value* (2)

```
AllText (n,value)
 1    n' ← first-child(n,all);
 2    while n' ≠ NULL do
 3        if type(n')=text then
 4            value ← concat(value,key[n']);
 5        else
 6            if type(n')=element then
 7                AllText(n',value);
 8            endif
 9        endif
10        n' ← right-sibling (n,all);
11    endw
```

La complessità computazionale è pari al più a *O(n)* poichè al massimo l'algoritmo effettua una visita completa dell'albero.

Elaborazione dello *string-value* (3)

- A questo punto possiamo introdurre la funzione **S(n)** che restituisce lo *string-value* di un nodo *n* qualsiasi.

```
S(n)
 1    value ⟵ ' ';
 2    if type(n) = text then
 3        value ⟵ key[n];
 4    else if type(n) = element then
 5        AllText(n,value);
 6    else
 7        n' ⟵ first-child(n,text);
 8        value ⟵ key[n'];
 9    endif
10    return value;
```

- Nella maggioranza dei casi il calcolo dello *string-value* di un nodo n, *S(n)*, avrà complessità costante *O(1)* perchè si utilizzano nodi terminali (le foglie dell'albero).

Elaborazione dell'asse *parent*

- Esempio: elaborazione dell'asse **parent::a**

```
parent (C,a)
 1    L ← NewList();
 2    while not(empty(C)) do
 3        n ← DelFirst(C);
 4        n' ← parent-node(n,element);
 5        if (flag(n') < CONT) and ((tag(n')=a) or (a='*')) then
 6            AddAfter(L,n');
 7            flag(n') ← CONT;
 8        endif
 9    endw
10    return L;
```

- Per eliminare il problema derivante da una duplicazione dei nodi genitori nel risultato dell'elaborazione utilizziamo la tecnica del *flag* numerico. La complessità di questo algoritmo è $O(n)$.

Elaborazione dell'asse *following* (1)

- Esempio: elaborazione dell'asse **following::a**
- Per poter elaborare l'asse **following::a** introduciamo una funzione ricorsiva *AddAllDescendant(L,n,a)* che restituisce i discendenti del nodo *n* con *tag(n)=a*

```
AddAllDescendant (L,n,a)
1    n' ← first-child(n,element);
2    while n' ≠ NULL do
3        if (tag(n')=a) or (a='*') then
4            AddAfter(L,n');
5        endif
6        AddAllDescendant(L,n',a);
7        n' ← right-sibling(n,element);
8    endw
```

- La complessità di questo algoritmo è *O(n)*.

Elaborazione dell'asse *following* (2)

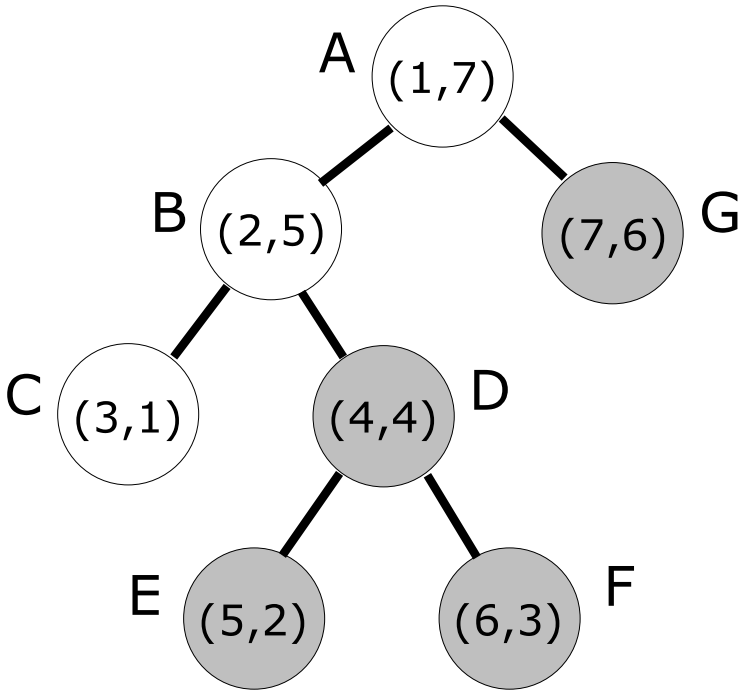
following (C,a)

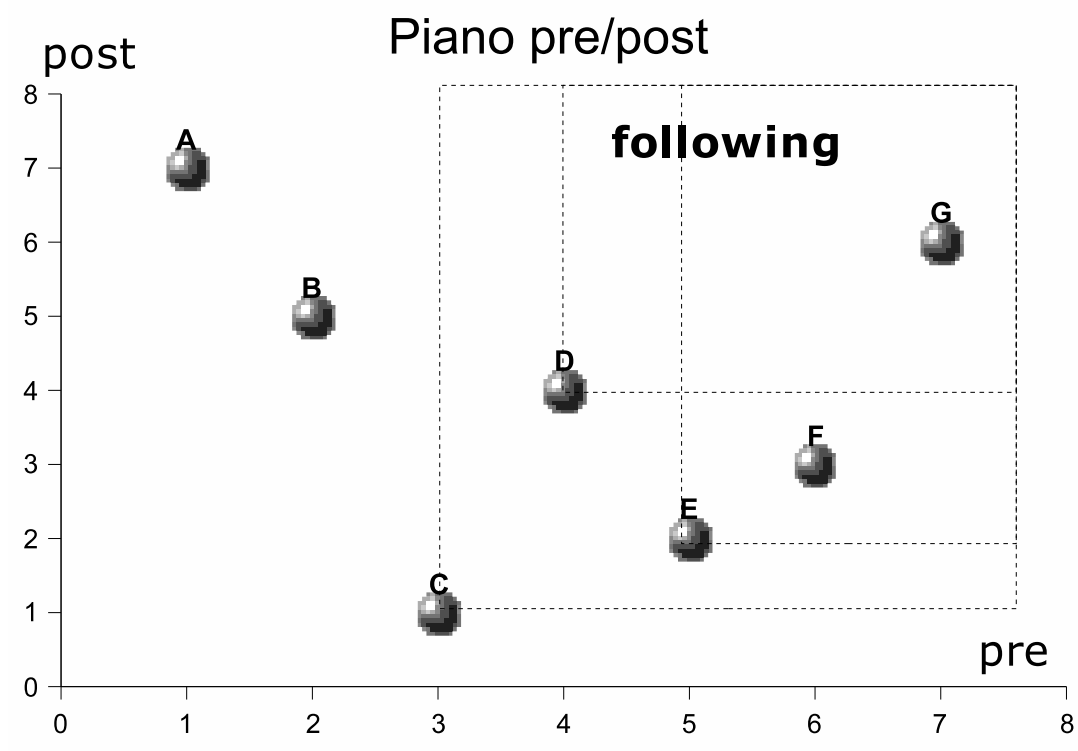
```
1   L ⟵ NewList();
2   if not(empty(C)) then
3       n ⟵ DelFirst(C);
4       while not(empty(C)) and post(First(C)) < post(n) do
5           n ⟵ DelFirst(C);
6       endw
7       while n ≠ NULL do
8           n' ⟵ right-sibling(n,element);
9           while n' ≠ NULL do
10              if (tag(n')=a) or (a='*') then
11                  AddAfter(L,n');
12              endif
13              AddAllDescendant (L,n',a);
14              n' ⟵ right-sibling(n',element);
15          endw
16          n ⟵ parent(n,element);
17      endw
18  endif
19  return L;
```

Per l'elaborazione dell'asse following abbiamo utilizzato i valori *post* dei nodi. La complessità di questo algoritmo è *O(n)*.

Elaborazione dell'asse *following* (3)

- Esempio: il following dell'insieme $\{C, D, E\}$ è lo stesso dell'insieme $\{C\}$ ossia è l'insieme $\{D, E, F, G\}$





Elaborazione degli assi *id* e id^{-1} (1)

- Esempio: elaborazione dell'asse **id::a** e **id^{-1}::a**
- Per l'elaborazione dell'asse **id::a** utilizzo una funzione *hash('s')* che restituisce, in media in tempo costante *O(1)*, il nodo dell'albero XML con attributo di tipo *ID* uguale ad *s*.
- Questa funzione *hash()* è implementata attraverso l'utilizzo di una tabella *hash* elaborata in fase di costruzione dell'albero XML.
- Anche per l'asse **id^{-1}::a** utilizzo una funzione $hash^{-1}$*('s')* che restituisce tutti i nodi che hanno un attributo *IDREF* o *IDREFS* contenente la stringa *s*.
- Anche la funzione $hash^{-1}$*()* è calcolata nella fase di costruzione dell'albero XML.

Elaborazione degli assi *id* e id^{-1} (2)

- Per elaborare gli assi **id** e $\mathbf{id}^{-1}$ su di un insieme di nodi C è necessario procedere in maniera iterativa su ogni nodo c $\in$ C.
- L'insieme risultato dell'elaborazione sarà, in generale, un insieme non ordinato di nodi poichè non esiste nessuna informazione preventiva sul 'salto' della funzione id() (in questo caso i valori *pre*/*post* ed il *flag* numerico non ci sono d'aiuto).
- E' possibile ordinare l'insieme in fase di elaborazione utilizzando un vettore *A* di cardinalità *n* inizializzato con valori uguale a zero.

Elaborazione degli assi *id* e id^{-1} (3)

- Per ogni nodo *c'* risultato dell'elaborazione dell'asse **id** o $\mathbf{id}^{-1}$ inseriamo in posizione *pre(c')* il puntatore del nodo *c'* nel vettore *A*.
- Al termine dell'elaborazione possiamo costruire la nostra lista *L* ordinata e senza duplicati andando ad effettuare la scansione del vettore A ed inserendo un elemento nella lista *L* se diverso da zero.
- In questo modo otteniamo un insieme risultato di nodi ordinato e senza duplicati.

Elaborazione degli assi *id* e *id*$^{-1}$ (4)

- La complessità dell'algoritmo è al più $O(n)$ per l'inizializzazione del vettore A, $O(n)$ per l'elaborazione degli assi **id** e **id**$^{-1}$ ed $O(n)$ per la scansione del vettore A e per la creazione della lista risultato, sommando questi valori di ottiene $O(3n) = O(n)$.
- Esiste un caso estremo nel quale ogni nodo di tipo IDREF di un documento XML ha cardinalità *n*, ossia quando ogni nodo del documento XML è 'puntato' da tutti gli altri nodi.
- Qui l'elaborazione degli assi *id* e *id*$^{-1}$ ha complessità $O(n^2)$, ma n^2 nel nostro modello di albero è proprio la dimensione del documento XML (poichè per noi un nodo di tipo IDREFS di dimensione *n* è costituito da un singolo nodo e non da *n* nodi di tipo IDREF).

Elaborazione degli assi

- Una volta elaborati tutti gli algoritmi sugli assi è possibile implementare una funzione *Evaluate(axis,a,C)* che consente di elaborare un asse generico `axis::a` su un insieme ordinato *C* di nodi.

Evaluate (axis,a,C)

```
1   switch axis do
2       case self : C ⟵ self(C,a);
3       case child : C ⟵ child(C,a);
4       . . .;
5   endsw
6   CONT ⟵ CONT + 1;
7   if CONT = MAX then
8       ResetTreeFlag();
9       CONT ⟵ 1;
10  endif
11  return C;
```

- Se *D* è il documento XML su cui elaborare la query, l'algoritmo *Evaluate()* ha complessità al più $O(|D|)$.

Elaborazione dei filtri (1)

- L'idea per l'elaborazione dei filtri è di tradurre un'interrogazione SXPath `/q[p]` in un'altra interrogazione SXPath, `/q` $\cap$ `p`$^{-1}$ (G.Gottlob et al. [2002])
- Introduciamo una funzione $\tau(p) : EXPath \mapsto EXPath$ che restituisce il percorso inverso di p
- Un filtro può essere elaborato con questa uguaglianza:

$$f(T, q[p], C) = f(T, q, C) \cap f(T, \tau(p), N)$$

  dove N è l'insieme di tutti i nodi dell'albero XML.
- Questa soluzione è onerosa dal punto di vista dello spazio di memoria occupato perchè ci obbliga ad utilizzare l'insieme *N*

Elaborazione dei filtri (2)

- Esiste una soluzione alternativa che consente di non utilizzare tutto l'insieme *N* ma solo la parte rilevante ai fini dell'interrogazione:

$$f(T, q[p], C) = f(T, q, C) \cap f(T, \tau(p), f(T, p, f(T, q, C)))$$

- E' possibile dimostrare che questa soluzione ha la stessa complessità computazionale della soluzione di G.Gottlob et. al [2002]. Il vantaggio è quello di non dover memorizzare tutto l'insieme *N* ma utilizzare al suo posto il risultato dell'interrogazione q/p (E.Zimuel [2005]).

Elaborazione di una query EXPath

- A questo punto abbiamo tutti gli strumenti per poter elaborare una query **q** EXPath dato **T** albero XML su di un contesto iniziale **C** (insieme dei nodi di partenza).

EXPath (T,q,C)

```
1   if not(empty(C)) then
2       if q = /path then
3           C ⟵ EXPath(T,path,{root});
4       else if q = q1/q2 then
5           C ⟵ EXPath(T,q2,EXPath(T,q1,C));
6       else if q = id('s') then
7           C ⟵ hash('s');
8       else if q = id('s')[path] then
9           C' ⟵ hash('s');
10          C ⟵ C' ∩ EXPath(T,Inverse(path),EXPath(T,path,C'));
11      . . .;
12  endif
13  return C;
```

Complessità

- Sia *D* il documento XML su cui elaborare la query *q*, l'algoritmo *EXPath(T,q,C)* ha, nel caso pessimo, complessità computazionale $O(|q| \cdot |D|)$.
- Abbiamo quindi dimostrato che è possibile elaborare una query *q* in EXPath su di un documento XML, che indichiamo con *D*, con complessità al più $O(|q| \cdot |D|)$.
- Utilizzando la funzione di traduzione $\phi(q) : SXPath \mapsto EXPath$ anche le interrogazioni in SXPath possono essere risolte con complessità al più $O(|q| \cdot |D|)$.

Metodo alternativo per EXPath (1)

- Ipotizziamo di non dover restituire per ogni *step* dell'elaborazione un insieme ordinato di nodi. L'ordinamento dei nodi viene effettuato solo alla fine dell'elaborazione.
- E' possibile risolvere un'interrogazione EXPath utilizzando un metodo alternativo basato esclusivamente sull'utilizzo del *flag* numerico
- In questo modo l'elaborazione degli assi **id** e $\mathbf{id}^{-1}$ è più efficiente perchè non è più necessario ordinare il risultato tramite l'utilizzo del vettore A di lunghezza *n*

Metodo alternativo per EXPath (2)

- Tutti gli algoritmi per l'elaborazione degli assi si semplificano notevolmente ed hanno tutti complessità al più $O(|D|)$ dove *D* è il documento XML.
- La complessità dell'elaborazione di una query EXPath rimane sempre lineare nella dimensione del documento *D* e della query *q* ossia $O(|q| \cdot |D|)$.
- Può essere interessante indagare con degli esperimenti l'efficienza di quest'ultima tecnica rispetto a quella presentata in precedenza che prevedeva l'utilizzo dei valori *pre*/*post* con l'ipotesi di ottenere sempre un insieme ordinato di nodi in ogni fase dell'elaborazione

Conclusioni

- In questa presentazione siamo partiti dal linguaggio **XPatterns** introdotto da G.Gottlob et al. [2002] e lo abbiamo esteso nel linguaggio **SXPath**
- Abbiamo esteso ulteriormente il linguaggio SXPath nel linguaggio **EXPath** per gestire in maniera efficiente l'operatore filtro all'interno delle query SXPath e per rendere più potente e simmetrico il linguaggio XPatterns
- Abbiamo dimostrato che è possibile risolvere in maniera efficiente interrogazioni EXPath e SXPath con la stessa complessità computazionale ottenuta da G.Gottlob et al. [2002] ossia $O(|q| \cdot |D|)$ dove *q* rappresenta l'interrogazione e *D* il documento XML.

Bibliografia

- L. Afanasiev, M. Franceschet, M. J. Marx e Rijke Rijke, *CTL model checking for processing simple XPath queries*, 2004, IEEE Computer Society Press
- M. Franceschet, E.Zimuel, *Comparing query evaluation strategies for navigational XPath*, 2005, Technical Report R-2005-001, Università degli Studi 'G.D'Annunzio' Chieti - Pescara, Dipartimento di Scienze
- G.Gottlob, C.Koch e R. Pichler, *Efficient Algorithms for Processing XPath Queries*, 2002, Very Large DataBases 2002 Conference
- G.Gottlob, C.Koch e R. Pichler, *XPath Query Evaluation: Improving Time and Space Efficiency*, 2003, IEEE International Conference on Data Engineering (ICDE)
- T.Grust, *Accelerating XPath Location Steps*, 2002, SIGMOD Conference
- T.Grust, M. van Keulen e J.Teubner, *Staircase Join: Teach a Relational DBMS to Watch its (Axis) Steps*, 2003, Very Large DataBases 2003 Conference
- J.Hidders e P.Michiels, *Efficient XPath Axis Evaluation for DOM Data Structures*, 2004, PLAN-X, Venice (Italy)
- E. Zimuel, *Risoluzione efficiente di interrogazioni XPath su documenti XML con attributi e riferimenti*, 2005, Tesi di Laurea, Università degli Studi 'G.D'Annunzio' Chieti - Pescara