

UNIVERSITÀ DI PISA



FACOLTÀ DI SCIENZE MATEMATICHE FISICHE E NATURALI
CORSO DI LAUREA SPECIALISTICA IN INFORMATICA

Tesi di laurea

Schemi XML con Interleaving e Conteggio: Inclusione e Validazione

Luca Pardini

Relatori

Prof. Giorgio Ghelli

Dott. Carlo Sartiani

Controrelatore

Prof. Paolo Ferragina

Anno Accademico 2007/2008

Riassunto

L'inserimento dell'operatore di interleaving in XML trasforma l'operazione di inclusione da PSPACE a EXPSPACE-Completo e l'operazione di validazione da PTIME a NP-Arduo. Rispettando alcune limitazioni è possibile costruire algoritmi polinomiali che si comportano molto bene anche in fase sperimentale.

Indice

Riassunto	i
1 Introduzione	1
2 Background	3
2.1 Espressioni regolari	3
2.2 Il linguaggio XML	4
2.3 Tipi in XML	5
2.4 XML Schema	6
2.5 Parser XML	9
2.6 Inclusione e validazione	10
2.7 Il problema LCA	10
2.8 Generatori XML	12
2.8.1 XMark	13
2.8.2 TAXI	14
2.9 Strumenti per la validazione	15
2.9.1 Harmony	15
2.9.2 Xerces	19
3 Tipi e vincoli	21
3.1 Il linguaggio dei tipi	21
3.1.1 Sintassi	22
3.1.2 Semantica	22
3.2 Il linguaggio dei vincoli	24
3.3 Estrazione dei vincoli	26

3.4	Correttezza e completezza dei vincoli	29
4	Inclusione	31
4.1	Sistema di deduzione	31
4.1.1	Vincoli piatti	31
4.1.2	Vincoli di co-occorrenza	32
4.1.3	Vincoli di ordine	34
4.1.4	Correttezza e completezza	34
4.2	Algoritmo base	35
4.2.1	Vincoli piatti	35
4.2.2	Vincoli di co-occorrenza	37
4.2.3	Vincoli di ordine	41
4.3	Algoritmo quadratico	43
4.4	Algoritmo strutturale	48
5	Validazione	55
5.1	Algoritmo base	55
5.2	Algoritmo con vincoli appiattiti	62
5.3	Algoritmo lineare	68
5.4	Validare sequenze di stringhe	70
5.5	Validazione per XML Schema	72
5.5.1	Validazione lazy	73
5.5.2	Validazione eager	74
5.5.3	Differenze tra gli algoritmi lazy e eager	76
6	Implementazione e sperimentazione	77
6.1	Aspetti implementativi	77
6.1.1	Tipi e algoritmi	77
6.1.2	Il parser	78
6.1.3	Test di regressione	79
6.2	Sperimentazione	80
6.2.1	Inclusione	80
6.2.2	Validazione	89

7 Conclusioni	93
A Schemi XML utilizzati	95
A.1 Schema DBLP	95
A.2 Schema expense-report	101
Bibliografia	107

Elenco delle figure

2.1	Tipica struttura di un documento XML Schema.	7
2.2	Esempio di dichiarazione di elementi con tipo semplice.	7
2.3	Esempio di dichiarazione di elementi con tipo complesso.	8
2.4	Esempio di dichiarazione di tipi personalizzati.	9
2.5	Algoritmo di preprocessing e di interrogazione per il problema RMQ.	13
2.6	Algoritmo per la validazione in Harmony.	18
4.1	Algoritmo di inclusione.	35
4.2	Algoritmo per la verifica dei vincoli piatti.	36
4.3	Rappresentazione grafica dei tipi dell'Esempio 4.16.	36
4.4	Algoritmo per la chiusura all'indietro.	38
4.5	Algoritmo per la verifica dei vincoli di co-occorrenza.	39
4.6	Algoritmo per la verifica dei vincoli di ordine.	42
4.7	Procedure di supporto all'algoritmo della Figura 4.8.	46
4.8	Algoritmo per la chiusura all'indietro veloce.	47
4.9	Procedure di supporto per l'algoritmo in Figura 4.10.	52
4.10	Algoritmo per l'inclusione di tipi simili.	53
5.1	Tabella di trasformazione dei vincoli interni in residuo.	56
5.2	Inizializzazione delle strutture per codificare i vincoli interni.	58
5.3	Procedure di supporto all'algoritmo della Figura 5.5.	59
5.4	Rappresentazione grafica del tipo dell'Esempio 5.5.	60
5.5	Algoritmo per l'appartenenza con costo $O(T + w * depth(T))$	61
5.6	Calcolo del residuo per operatori n-ari.	63
5.7	Algoritmo per l'appartenenza con costo $O(T + w * flatdepth(T))$	65

5.8	Rappresentazione grafica del tipo dell'Esempio 5.9.	67
5.9	Procedure di supporto all'algoritmo della Figura 5.7.	67
5.10	Algoritmo per l'appartenenza con costo $O(T + w)$	69
5.11	Procedure di supporto all'algoritmo della Figura 5.10.	70
5.12	Procedure il ripristino delle strutture dati.	71
6.1	Struttura delle classi per rappresentare un tipo.	78
6.2	Algoritmo per la generazione di tipi casuali.	82
6.3	Grafico dei tempi relativi ai diversi algoritmi di inclusione misurati sull'insieme casuale.	85
6.4	Grafici dei tempi parziali relativi agli algoritmi di inclusione misu- rati sull'insieme casuale.	86
6.5	Grafico dei tempi relativi ai diversi algoritmi di inclusione misurati sull'insieme modificato.	87
6.6	Grafici dei tempi parziali relativi agli algoritmi di inclusione misu- rati sull'insieme modificato.	88
6.7	Grafico dei tempi relativi agli algoritmi di validazione misurati sull'insieme generato da XMark.	89
6.8	Grafico dei tempi relativi agli algoritmi di validazione misurati sull'insieme DBLP.	90
6.9	Grafico dei tempi relativi agli algoritmi di validazione misurati sull'insieme expense-report.	91

Capitolo 1

Introduzione

Dieci anni fa nacque l'eXtensible Markup Language (XML), un metalinguaggio che permette di costruire dei linguaggi personalizzati di markup, sviluppato dal World Wide Web Consortium (W3C). L'obiettivo principale di XML è semplificare SGML focalizzandosi su un particolare problema: i documenti su internet. XML è utilizzato sia per codificare documenti sia per serializzare dati ed è in grado di esprimere semplicemente dati e documenti anche con una struttura molto complessa.

Gli schemi XML sono strumenti che vengono utilizzati per garantire la robustezza di applicazioni che manipolano e scambiano dati XML, descrivendo la struttura che tali dati devono avere. In queste applicazioni si rivela fondamentale l'utilizzo delle seguenti operazioni:

- verificare quando uno schema è incluso nell'altro;
- verificare quando un'istanza di dati XML rispetta la struttura definita da uno schema.

I principali linguaggi di schematizzazione per dati XML sono le DTD e XML Schema ([BNdB04]). Tali linguaggi sono stati disegnati per descrivere dati ordinati, ma offrono un limitato supporto per i dati nel quale l'ordine non è necessario. Per avere un pieno supporto dei dati non ordinati è necessario introdurre un nuovo operatore, detto operatore di interleaving. Sfortunatamente, l'aggiunta di questo nuovo operatore rende le operazioni di inclusione e validazione non trattabili,

in quanto l'inclusione di due schemi XML diventa un problema EXPSPACE-Completo, mentre il problema di verificare l'appartenenza di un'istanza di dati a uno schema XML diventa NP-Arduo [MS94].

In questa tesi presenteremo una sottoclasse di tipi XML, arricchita con operatori per definire il conteggio e il disordine, che raccoglie una larga parte dei tipi XML usati in pratica, nel quale le operazioni di inclusione e validazione si possono eseguire in tempo polinomiale.

Contributo della tesi Questa tesi si pone l'obiettivo di esporre due algoritmi polinomiali per risolvere l'inclusione e la validazione con schemi che includono l'operatore di interleaving e di conteggio. A tal fine presenteremo una sottoclasse che contiene questi operatori. Da ogni tipo della nostra sottoclasse estrarremo un insieme di vincoli che rappresenteranno la base per costruire gli algoritmi: l'inclusione verrà risolta costruendo un sistema di derivazione, mentre la validazione verrà risolta modificando i vincoli dopo ogni elemento letto.

Concluderemo la tesi mostrando gli esperimenti fatti sugli algoritmi che abbiamo implementato e vedremo che essi sono efficienti anche in fase sperimentale e con risultati paragonabili a strumenti che non supportano l'operatore di interleaving.

Organizzazione della tesi La tesi è strutturata nel modo seguente:

- nel Capitolo 2 presenteremo i concetti necessari a comprendere i formalismi e i risultati ottenuti nei capitoli successivi;
- nel Capitolo 3 introdurremo i tipi formalizzandone la sintassi, la semantica, le restrizioni e l'estrazione dei vincoli;
- nel Capitolo 4 parleremo degli algoritmi di inclusione sui tipi;
- nel Capitolo 5 descriveremo gli algoritmi di validazione;
- nel Capitolo 6 vedremo alcuni aspetti implementativi e i risultati che abbiamo ottenuto dalle sperimentazioni.

Capitolo 2

Background

In questo capitolo introduciamo le nozioni fondamentali utilizzate nei capitoli successivi. Descriveremo le espressioni regolari, la rappresentazione di XML, dei tipi su esso definiti e introdurremo le problematiche da risolvere. Inoltre, presenteremo alcuni strumenti utilizzati al fine di confrontare le prestazioni dei vari algoritmi.

2.1 Espressioni regolari

Le espressioni regolari sono un formalismo molto usato nel campo dell'informatica per descrivere insiemi di stringhe su un alfabeto Σ . Una stringa (o parola) è definita come una sequenza finita di simboli $w = a_1, a_2, \dots, a_n$; indichiamo, inoltre, con ϵ la stringa vuota.

Definizione 2.1. L'insieme delle espressioni regolari E_Σ su un alfabeto Σ è definito ricorsivamente da:

$$\begin{aligned} \epsilon &\in E_\Sigma \\ a &\in E_\Sigma, \quad \forall a \in \Sigma \\ e_1 \cdot e_2 &\in E_\Sigma, \quad \forall e_1, e_2 \in E_\Sigma \\ e_1 + e_2 &\in E_\Sigma, \quad \forall e_1, e_2 \in E_\Sigma \\ e^i &\in E_\Sigma, \quad \forall i \in \mathbb{N}, \forall e \in E_\Sigma \\ e^* &\in E_\Sigma, \quad \forall e \in E_\Sigma \end{aligned}$$

Con $e_1 \cdot e_2$, $e_1 + e_2$ e e^* indichiamo rispettivamente l'operatore di concatenazione, unione e stella di Kleene.

Definizione 2.2. La semantica, definita mediante insiemi di stringhe, è data da:

$$\begin{aligned}
 \llbracket \epsilon \rrbracket &= \{ \epsilon \} \\
 \llbracket a \rrbracket &= \{ a \} \\
 \llbracket e_1 \cdot e_2 \rrbracket &= \{ uv \mid u \in \llbracket e_1 \rrbracket, v \in \llbracket e_2 \rrbracket \} \\
 \llbracket e_1 + e_2 \rrbracket &= \{ u \mid u \in \llbracket e_1 \rrbracket \cup \llbracket e_2 \rrbracket \} \\
 \llbracket e^0 \rrbracket &= \{ \epsilon \} \\
 \llbracket e^{n+1} \rrbracket &= \{ uv \mid u \in \llbracket e \rrbracket, v \in \llbracket e^n \rrbracket \} \\
 \llbracket e^* \rrbracket &= \{ u \mid u \in \cup_{i \in \mathbb{N}} \llbracket e^i \rrbracket \}
 \end{aligned}$$

Esempio 2.3. Consideriamo l'espressione regolare $a + (b \cdot c)$: essa rappresenta l'insieme delle stringhe $\{a, bc\}$, mentre l'espressione regolare $(a \cdot b)^*$ rappresenta l'insieme $\{\epsilon, ab, abab, ababab, \dots\}$.

2.2 Il linguaggio XML

XML è un linguaggio utilizzato per rappresentare dati tramite una struttura a forma di albero costituito da un'unica radice con un numero arbitrario di sottoalberi.

Ogni nodo dell'albero può contenere un numero arbitrario n di figli $x_1 \dots x_n$ delimitati da una etichetta a , dove $\langle a \rangle$ rappresenta l'inizio (o apertura) e $\langle /a \rangle$ rappresenta la fine (o chiusura) dell'etichetta; il nodo viene, quindi, indicato da $\langle a \rangle x_1 \dots x_n \langle /a \rangle$.

Un nodo può contenere valori e attributi. Con valori indichiamo stringhe, interi e altre tipologie di dati, che prendono il posto di tutti o alcuni dei figli del nodo: ad esempio, in $\langle a \rangle b \langle /a \rangle$ l'etichetta a contiene come valore la stringa b . Gli attributi sono identificatori dichiarati all'apertura di una etichetta contenenti valori, ad esempio in $\langle a \ id = '1' \ nome = 'b' \rangle \langle /a \rangle$ l'etichetta a contiene gli attributi id e $nome$ rispettivamente con i valori 1 e b .

La nostra rappresentazione è focalizzata sulla struttura di un documento XML senza occuparsi di valori o attributi, in quanto gli algoritmi che presenteremo nei capitoli successivi si concentrano esclusivamente sulla struttura.

Definizione 2.4. L'albero t , che rappresenta la struttura di un documento XML, è definito ricorsivamente come:

- un documento della forma $\langle a \rangle w \langle /a \rangle$, dove w contiene soltanto testo, ha un albero t rappresentato da un nodo con etichetta a , rappresentato con $a()$;
- un documento della forma $\langle a \rangle x_1 \dots x_n \langle /a \rangle$, con $t_1 \dots t_n$ alberi associati rispettivamente a $x_1 \dots x_n$, ha un albero t rappresentato da un nodo con la radice etichettata da a e con n figli ordinati da sinistra verso destra $t_1 \dots t_n$, rappresentato con $a(t_1 \dots t_n)$.
- l'insieme delle etichette di t è l'insieme dei nomi degli elementi del documento;

Esempio 2.5. Consideriamo il documento XML $\langle a \rangle \langle b \rangle c \langle /b \rangle \langle d \rangle e \langle /d \rangle \langle /a \rangle$: l'albero t che rappresenta la struttura del documento è dato da $a(b()d())$.

2.3 Tipi in XML

I formalismi che vedremo in questa sezione definiscono un insieme di alberi $L(d)$. Questi formalismi sono usati per indicare quali documenti XML sono *validi*, ovvero presenti nell'insieme di alberi definito, oppure *non validi*, ovvero non presente in tale insieme.

Definizione 2.6. Una DTD su un alfabeto Σ è una tripla (Σ, d, s_d) dove Σ è un insieme di simboli, $s_d \in \Sigma$ è il simbolo iniziale (o radice) e d è una funzione che associa ad ogni elemento di Σ una espressione di E_Σ detta *content model*.

Un albero t è valido per una DTD se la sua radice è etichettata con s_d e se, per ogni nodo n etichettato con a , la sequenza $a_1 \dots a_n$ delle etichette dei suoi figli è valida per l'espressione $d(a)$.

Esempio 2.7. Consideriamo la seguente funzione d con $\Sigma = \{a, b, c, d\}$ e $s_d = a$:

$$\begin{aligned} a &\rightarrow b + c \\ c &\rightarrow dd \end{aligned}$$

L'insieme di alberi denotato è $\{a(b()), a(c(d()d()))\}$.

Definizione 2.8. Una EDTD (Extended DTD) su un alfabeto Δ è una quintupla $(\Delta, \Sigma, d, s_d, \mu)$, dove Δ è un insieme di simboli, (Σ, d, s_d) è una DTD sull'alfabeto Σ e μ è una funzione che trasforma simboli di Σ in simboli di Δ .

La funzione μ può essere applicata ad un albero, etichettato con simboli di Σ , per rietichettare i nodi, ottenendo un albero etichettato con simboli di Δ . Un albero t è valido per una EDTD se t può essere riscritto come $\mu(t')$ dove t' è valido per la DTD (Σ, d, s_d) .

Esempio 2.9. Consideriamo la DTD dell'Esempio 2.7 con $\Delta = \{x, y, z\}$ e la funzione μ definita da:

$$\begin{aligned}\mu(a) &= x \\ \mu(b) = \mu(c) &= y \\ \mu(d) &= z\end{aligned}$$

L'insieme di alberi denotato è $\{x(y()), x(y(z()z()))\}$.

Definizione 2.10. Una *single-type* EDTD (EDTDst) su un alfabeto Σ è una EDTD $(\Sigma, \Delta, d, s_d, \mu)$ con la proprietà che, per ogni simbolo $a \in \Delta$, nell'espressione regolare $d(a)$ non possono comparire due elementi con lo stesso nome ($\mu(a)$) ma tipo differente.

Esempio 2.11. Consideriamo la EDTD dell'Esempio 2.9: essa non è single-type perché nell'espressione $a \rightarrow b + c$ i simboli b e c hanno lo stesso nome definito da $\mu(b) = \mu(c) = y$ ma hanno tipo differente, in particolare b è una foglia e c no.

Tale linguaggio può essere tuttavia generato dalla seguente EDTDst, dove $\Delta = \Sigma = \{x, y, z\}$, $s_d = x$, μ è la funzione identica e la funzione d è definita come:

$$\begin{aligned}x &\rightarrow y \\ y &\rightarrow zz + \epsilon\end{aligned}$$

2.4 XML Schema

XML Schema è un documento XML che utilizza un insieme di etichette speciali, che formano l'*XML Schema Language*, per definire la struttura di un documento XML. Inoltre, XML Schema ha la stessa espressività delle EDTDst.

Un XML Schema ha un elemento radice etichettato con $\langle xs : schema \rangle$, il quale specifica tramite l'attributo *xmlns* che nel documento sono utilizzate le etichette definite da uno specifico standard del W3C (World Wide Web Consortium), come illustrato in Figura 2.1.

```
<?xml version="1.0"?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">
... Definizione della grammatica ...
</xs:schema>
```

Figura 2.1: *Tipica struttura di un documento XML Schema.*

In XML Schema definiamo tramite l'etichetta $\langle xs : element \rangle$ gli elementi del nostro documento. Un elemento può avere come tipo di dato un tipo semplice oppure un tipo complesso.

I tipi di dato semplici sono relativi a quegli elementi che non possono contenere altri elementi (foglie), non prevedono attributi e sono definiti mediante l'etichetta $\langle xs : simpleType \rangle$. Sono previsti numerosi tipi di dato predefiniti come ad esempio i tipi classici: stringa, intero, booleano, ecc.

```
<xs:element name="lastname" type="xs:string"/>
<xs:element name="age" type="xs:integer"/>
<xs:element name="dateborn" type="xs:date"/>
<xs:element name="age">
<xs:simpleType>
  <xs:restriction base="xs:integer">
    <xs:minInclusive value="0"/>
    <xs:maxInclusive value="120"/>
  </xs:restriction>
</xs:simpleType>
</xs:element>
```

Figura 2.2: *Esempio di dichiarazione di elementi con tipo semplice.*

È possibile definire tipi semplici personalizzati derivandoli da quelli predefiniti.

Ad esempio, possiamo definire un tipo di dato come restrizione del tipo stringa, vincolando i valori ad uno specifico insieme di stringhe o a un insieme definito da un'espressione regolare, come mostrato in Figura 2.2.

```
<xs:element name="person">
  <xs:complexType>
    <xs:sequence>
      <xs:element name="full_name" type="xs:string"/>
      <xs:element name="child_name" type="xs:string"
        maxOccurs="10" minOccurs="0"/>
    </xs:sequence>
  </xs:complexType>
</xs:element>
```

Figura 2.3: Esempio di dichiarazione di elementi con tipo complesso.

I tipi di dato complessi si riferiscono alla definizione di elementi con attributi e che possono contenere altri elementi.

La definizione del tipo complesso consiste generalmente nella definizione della struttura prevista dall'elemento utilizzando l'etichetta $\langle xs : complexType \rangle$. Se l'elemento può contenere altri elementi, possiamo definire l'insieme degli elementi che possono stare al suo interno come una sequenza o come un insieme di valori alternativi specificando, con un intervallo, quanti elementi di quel tipo devono essere presenti (si veda, ad esempio, la Figura 2.3).

In XML Schema è possibile dichiarare tipi di dato personalizzati e fare riferimento a tali dichiarazioni quando si definisce un elemento, in modo analogo a come avviene per la definizione di tipi nei linguaggi di programmazione, come illustrato in Figura 2.4. Questo consente di rendere più leggibile lo schema e di concentrare in un unico punto la definizione di un tipo utilizzato diverse volte.

È possibile comporre schemi includendo o importando schemi diversi ed è possibile comporre documenti XML utilizzando tag definiti in schemi diversi. Ad esempio, possiamo integrare i nostri documenti XML che descrivono ricette di cucina con l'introduzione di nuovi elementi che definiscono gli utensili che servono per eseguire ciascuna ricetta. Supponendo che tali elementi siano definiti in un

```
<xs:complexType name="shipordertype">
  <xs:sequence>
    <xs:element name="orderperson" type="stringtype"/>
    <xs:element name="shipto" type="shiptotype"/>
    <xs:element name="item" maxOccurs="unbounded"
      type="itemtype"/>
  </xs:sequence>
  <xs:attribute name="orderid" type="orderidtype"
    use="required"/>
</xs:complexType>
<xs:element name="shiporder" type="shipordertype"/>
```

Figura 2.4: *Esempio di dichiarazione di tipi personalizzati.*

apposito XML Schema, possiamo combinare le etichette derivanti dai due schemi in un unico documento XML.

2.5 Parser XML

Un parser XML è uno strumento in grado leggere un documento XML al fine di determinarne la sua struttura. I parser XML si suddividono in due categorie: DOM (Document Object Model) e SAX (Simple API for XML):

I parser DOM vedono XML come un albero e come tale lo rappresentano in memoria. Con questo tipo di parser viene caricato un albero la cui struttura rispecchia quella del documento. Questo albero può essere letto e modificato a piacimento.

I parser SAX forniscono una visione del documento XML tramite eventi lanciati durante la lettura del file. A differenza del parser DOM, non costruiscono un albero in memoria, ma si limitano a richiamare delle funzioni definite dall'utente quando gli eventi sono generati. Gli eventi sono di vario tipo, come per esempio l'inizio o la fine del documento, l'apertura o chiusura di un etichetta, la lettura di un carattere, ecc.

Il vantaggio principale dei parser SAX rispetto a quelli DOM sta nel non dover

leggere tutto il file prima di iniziare a lavorare e di occupare una quantità inferiore di memoria. Risulta impossibile utilizzare un parser DOM per dati XML molto grandi, oppure per dati in *streaming*, ovvero con un flusso continuo.

2.6 Inclusione e validazione

Definiamo formalmente i problemi di *inclusione* tra due tipi e di *appartenenza* di un albero ad un tipo.

Definizione 2.12 (inclusione). Dati due tipi U e T che denotano, rispettivamente, l'insieme di alberi $L(U)$ e $L(T)$, diremo che U è incluso in T se $L(U) \subseteq L(T)$ e lo indicheremo con $U \subseteq T$.

Esempio 2.13. Consideriamo il tipo U definito dalla seguente DTD:

$$a \rightarrow b \cdot c$$

Consideriamo adesso il tipo T definito dalla seguente DTD:

$$a \rightarrow b \cdot c + d$$

Il tipo U rappresenta l'insieme $L(U) = \{a(b()c())\}$ mentre il tipo T rappresenta l'insieme $L(T) = \{a(b()c()); a(d())\}$ essendo $L(U) \subseteq L(T)$ possiamo dire che $U \subseteq T$.

Definizione 2.14 (validazione). Dato il tipo T che denota l'insieme di alberi $L(T)$, diremo che l'albero t è membro di T se $t \in L(T)$ e lo indicheremo con $t \in T$.

Esempio 2.15. Consideriamo il tipo T definito dalla DTD dell'Esempio 2.13 e consideriamo l'albero $t = a(b()c())$. L'albero t appartiene al tipo T in quanto risulta $t \in L(T)$.

2.7 Il problema LCA

Il problema di calcolare il *minimo antenato comune* tra due nodi di un albero t viene detto problema LCA (Least Common Ancestor).

In questa sezione vedremo un semplice algoritmo per risolvere questo problema strutturato in due fasi: una fase iniziale di preprocessing e una fase di risposta alle interrogazioni. Indicheremo il costo di questo algoritmo come $\langle f(n), g(n) \rangle$, dove $f(n)$ è il costo della fase di preprocessing, mentre $g(n)$ è il costo della fase di interrogazione.

Definizione 2.16. Dati due nodi u e v di un albero T , l'interrogazione $LCA_T(u, v)$ restituisce il minimo antenato comune tra u e v nell'albero T , ovvero restituisce il nodo più in lontano dalla radice che è contemporaneamente antenato sia di u che di v .

Esempio 2.17. Consideriamo l'albero $a(b(d()e())c())$; il minimo antenato comune tra il nodo c e il nodo d è il nodo a , mentre il minimo antenato comune tra il nodo d e il nodo e è il nodo b .

La soluzione di questo problema può essere derivata dalla soluzione al problema di trovare il minimo elemento di un intervallo in un array di numeri interi, detto problema RMQ (Range Minimum Query).

Definizione 2.18. Dati due indici i e j compresi tra 1 e n su un array A di dimensione n , l'interrogazione $RMQ_A(i, j)$ restituisce l'indice del minimo valore nel sottoarray $A[i \dots j]$.

Esempio 2.19. Consideriamo l'array $A = [3, 2, 1, 2, 3]$; il minimo tra l'elemento di indice 2 e l'elemento di indice 4 è l'elemento $A[3] = 1$, mentre il minimo tra l'elemento di indice 1 e l'elemento di indice 2 è l'elemento $A[2] = 2$.

La derivazione della soluzione del problema LCA dal problema RMQ è verificata dal seguente teorema dimostrato in [AM00].

Teorema 2.20. Se esiste una soluzione per il problema RMQ con costo in tempo $\langle f(n), g(n) \rangle$, allora esiste una soluzione per il problema LCA con costo in tempo $\langle f(2n - 1) + O(n), g(2n - 1) + O(1) \rangle$.

Il Teorema 2.20 permette di concentrarsi esclusivamente sul risolvere il problema RMQ, che ammette una soluzione con costo in tempo $\langle O(n), O(1) \rangle$. Questa soluzione rappresenta un algoritmo elaborato da implementare e, come vedremo

nei capitoli successivi, senza un impatto dominante dal punto di vista computazionale. Per i nostri scopi è sufficiente utilizzare un algoritmo molto più semplice ma con costo $\langle O(n^2), O(1) \rangle$.

Per ridurre il problema LCA al problema RMQ utilizziamo il concetto di *cammino euleriano* su un albero.

Definizione 2.21. Un cammino euleriano su un albero t è dato dalla sequenza di nodi incontrati durante una visita in profondità dell'albero tornando indietro dopo aver visitato un sottoalbero.

Esempio 2.22. Consideriamo l'albero $a(b(c())d())$; il cammino euleriano è dato dalla sequenza dei nodi $abcada$.

La riduzione dal problema LCA, relativa all'albero t , al problema RMQ avviene costruendo, in tempo lineare, le seguenti strutture:

- un array $E[]$, di lunghezza $2n - 1$, dove $E[i]$ contiene l' i -esimo nodo del cammino euleriano di t ;
- un array $L[]$, di lunghezza $2n - 1$, dove $L[i]$ contiene la profondità rispetto alla radice del nodo $E[i]$ nell'albero t ;
- una tabella hash $R[]$, dove per ogni nodo n , $R[n]$ contiene l'indice alla prima occorrenza del nodo n nel cammino euleriano di t .

L'algoritmo (Figura 2.5) utilizza queste strutture dati per costruire una tabella $results[][]$ contenente le risposte di tutte le $(2n-1)^2$ possibili interrogazioni relative al problema RMQ sull'array $L[]$ e utilizzata per accedere in tempo costante in fase di interrogazione. Per calcolare il minimo antenato comune tra due nodi u e v è sufficiente, quindi, recuperare rispettivamente i due indici $i = R[u]$ e $j = R[v]$ e prelevare dalla tabella l'indice k ; il risultato è contenuto in $E[k]$.

2.8 Generatori XML

I generatori automatici di dati XML sono strumenti che permettono, a partire da un tipo, di costruire grandi quantità di dati XML in un tempo molto limitato e in

```
BUILDRMQ( $L[]$ )
1   $n := \text{size of } L$ 
2  global  $results := \text{new int}[n, n]$ 
3  for  $i = 1$  to  $n$ 
4  do  $min := L[i]$ 
5      $minIndex := i$ 
6     for  $j = i$  to  $n$ 
7     do if  $L[j] < min$ 
8         then  $min := L[j]$ 
9              $minIndex := j$ 
10         $results[i, j] := minIndex$ 
11         $results[j, i] := minIndex$ 

QUERYRMQ( $i, j$ )
1  global  $results$ 
2  return  $results[i, j]$ 

QUERYLCA( $u, v$ )
1  return  $E[\text{QUERYRMQ}(R[u], R[v])]$ 
```

Figura 2.5: *Algoritmo di preprocessing e di interrogazione per il problema RMQ.*

accordo al tipo dal quale siamo partiti. Questi strumenti sono fondamentali per eseguire dei test su applicazioni che fanno uso di XML. Analizzeremo due strumenti con caratteristiche molto diverse tra loro e che abbiamo utilizzato durante il periodo di tesi.

2.8.1 XMark

XMark è un generatore di dati XML che modella una tipica applicazione commerciale, ovvero un sito per le aste online ([SWK⁺01]).

XMark è scritto utilizzando il linguaggio C e compilato per quattro piattaforme

tra cui Windows e Linux. Inoltre, XMark fornisce le seguenti funzionalità:

- generare documenti XML validi per lo schema predefinito che modella la base di dati di un sito di aste online;
- una generazione efficiente e scalabile di documenti XML anche su dimensioni dell'ordine dei gigabyte;
- un basso uso di memoria utilizzata per generare i documenti XML anche in caso di file molto grandi.

Il numero e il tipo degli elementi sono scelti secondo un template parametrizzato da alcune distribuzioni di probabilità, mentre i valori sono presi dalle opere di Shakespeare.

Il generatore di XMark ha un solo parametro di ingresso per personalizzare il documento XML generato, la dimensione del documento di output. Questo generatore può essere usato per vari scopi, ad esempio per generare un grande documento XML da interrogare per valutare la scalabilità di un'applicazione, oppure per generare una grossa quantità di dati di dimensioni variabili al fine di misurare l'andamento di un algoritmo.

2.8.2 TAXI

TAXI (Testing by Automatically generated XML Instances) è uno strumento che permette di generare dati XML (istanze) a partire da uno XML Schema definito dall'utente ([BGMP07a]). TAXI è scritto utilizzando il linguaggio Java e utilizzabile su qualsiasi piattaforma.

Differentemente da XMark, TAXI offre una vasta serie di opzioni per personalizzare i dati XML prodotti dal generatore:

- scegliere un XML Schema che verrà utilizzato per generare dei documenti XML validi: in questa fase verranno distribuiti uniformemente dei pesi ai tipi unione (choice);
- modificare i pesi assegnati di default durante la scelta dello schema per esprimere una maggiore importanza in fase di generazione del documento XML;

- scegliere una base di dati contenente l'insieme di valori da utilizzare oppure permettere a TAXI di inferire automaticamente dallo schema quale sono i valori corretti;
- generare documenti XML a partire da un sottoinsieme dello schema scelto;
- scegliere una strategia di selezione tra tutte le possibili istanze generabili potendo impostare un limite al numero di istanze e una percentuale di copertura rispetto agli elementi scelti nei tipi unione.

TAXI rappresenta uno strumento più personalizzabile rispetto a XMark ma, essendo ancora in fase di completamento, risulta meno scalabile su file di grandi dimensioni. Inoltre, le opzioni che TAXI permette di personalizzare offrono un controllo limitato sulla dimensione dei file generati. Per questo motivo, gli esperimenti presentati nei capitoli successivi, sono stati effettuati con istanze XML generate sia da TAXI che da XMark.

2.9 Strumenti per la validazione

In questa sezione vediamo due strumenti con il quale ci siamo confrontati per l'algoritmo di validazione.

2.9.1 Harmony

Harmony è uno strumento che lavora su dati XML che sono stati replicati su più file e permette di gestire la sincronizzazione tra questi dati quando si presentano dei disallineamenti. Esso può essere usato, ad esempio, per sincronizzare una piccola base di dati che è stata replicata su molti supporti dove ogni supporto presenta degli aggiornamenti da applicare a tutte le altre.

Harmony utilizza al suo interno un algoritmo per la validazione che ha l'obiettivo di essere efficiente in pratica, pur presentando un alto costo computazionale dal punto di vista teorico. Vediamo brevemente come è definito il modello dei dati di questo algoritmo, come sono rappresentate le formule di Presburger, il funzionamento gli automi sheave basati su queste formule e un algoritmo base.

Modello dei dati

Il modello dei dati prevede una modellazione degli alberi e una per definirne degli insiemi. Indichiamo con $n \rightarrow t_n$ un figlio etichettato n con sottoalbero t_n , mentre con $\{\}$ indichiamo l'albero vuoto. Ogni albero viene rappresentato da una funzione parziale da etichette in alberi. Indicheremo con $\text{dom}(t)$ il dominio dell'albero t ovvero l'insieme dei nomi di tutti i suoi figli e con $t(n)$ indicheremo l'immediato sottoalbero etichettato con n .

Definizione 2.23. La sintassi degli tipi di alberi deterministici (DTTs) è la seguente:

$$T ::= \{\} \mid r[T] \mid r[T]^* \mid T + T \mid T \mid T \mid \neg T \mid X$$

dove r è una espressione regolare su un alfabeto Σ .

Il tipo $\{\}$ denota l'insieme che contiene l'albero vuoto. Il tipo $r[T]$ denota l'insieme degli alberi con un solo figlio dove l'etichetta appartiene all'insieme denotato da r e il sottoalbero all'insieme denotato da T . Il tipo $r[T]^*$ denota la chiusura di Kleene di $r[T]$. I tipi $T + T$ e $T \mid T$ denotano rispettivamente la concatenazione e l'unione su insiemi di alberi. Il tipo $\neg T$ denota il complemento dell'insieme denotato da T ovvero l'insieme di tutti gli alberi esclusi quelli di T . Il tipo X denota l'insieme denotato da X nell'ambiente $\Delta = \{X_1 = T_1 \dots X_k = T_k\}$.

Aritmetica di Presburger

L'aritmetica di Presburger è una teoria del primo ordine sui naturali con l'operatore di addizione ma senza quello di moltiplicazione, con la proprietà di essere decidibile.

Definizione 2.24. La sintassi di una espressione di Presburger ϕ è definita da:

$$\begin{aligned} \phi & ::= e = e \mid \phi \vee \phi \mid \phi \wedge \phi \mid \neg \phi \mid \exists. \phi \\ e & ::= i \mid x_j \mid e + e \end{aligned}$$

dove i è una costante e la variabile x_j rappresenta il j -esimo quantificatore. Se sono presenti meno di j quantificatori, x_j rappresenta la $(j - k)$ -esima variabile libera.

La semantica di una formula di Presburger è definita dal vettore v di naturali che la soddisfano. Scriveremo $v \models \phi$ se il vettore v soddisfa la formula ϕ e con $\models \phi$ indicheremo $\exists v \models \phi$.

Automati sheave

Un automa sheave è composto da un insieme finito di stati e da una funzione Γ che trasforma uno stato in una *formula sheave*. La transizione da uno stato all'altro è data dalla formula sheave associata allo stato di partenza in Γ .

Ogni formula sheave ha due componenti: una formula di Presburger ϕ e una lista di elementi della forma $r_i[S_i]$, dove r_i è un'espressione regolare che chiamiamo tag, mentre S_i è uno stato.

Il funzionamento di un automa è il seguente. Sia t un albero e S gli stati di un automa con $\Gamma(S) = (\phi, (r_0[S_0], \dots, r_k[S_k]))$; per ogni i compreso tra 0 e k sia c_i il numero di figli $n \in \text{dom}(t)$ per cui vale $n \in r_i$ e $t(n)$ è accettato da S_i . L'albero t è accettato se e solo se $\langle c_0, \dots, c_k \rangle \models \phi$.

Gli automi sheave devono essere *ben formati*. Un automa (ϕ, E) , dove abbiamo $E = (r_0[S_0], \dots, r_k[S_k])$, è detto ben formato se e solo se valgono le seguenti condizioni:

- le variabili libere di ϕ sono $\{x_0, \dots, x_{k-1}\}$;
- se esiste albero t accettato contemporaneamente da S_i e da S_j , con $i \neq j$, allora r_i e r_j sono espressioni regolari che denotano linguaggi disgiunti;
- per ogni albero t e ogni etichetta n esiste un elemento $r_i[S_i]$ dove vale $n \in r_i$ e t è accettato da S_i .

Se l'automata (ϕ, E) è ben formato, ogni albero ha un'unica decomposizione su E , questo significa che la semantica degli automi di sheave è ben definita.

Algoritmo

L'algoritmo proposto si basa sul seguente teorema la cui dimostrazione è presente in [FPS].

Teorema 2.25. Sia T un insieme di alberi descritto con la sintassi della Definizione 2.23 è possibile costruire un unico automa sheave che accetta lo stesso insieme di alberi.

Esempio 2.26. Consideriamo il tipo $(\{\} \vdash (a[T] + b[T]))$ esso è equivalente all'automata definito dallo stato S e dalla seguente funzione $\Gamma(S)$:

$$\Gamma(S) = \left(\begin{array}{l} ((x_0 = 0 \wedge x_1 = 0) \vee (x_0 = 1 \wedge x_1 = 1)) \wedge (x_2 = 0), \\ (a[T], b[T], \neg\{a, b\}[T]) \end{array} \right)$$

dove T rappresenta lo stato che accetta tutti gli alberi e $\neg\{a, b\}$ rappresenta l'insieme di tutte le stringhe escluse $\{a, b\}$.

L'algoritmo esplora non deterministicamente tutto lo spazio degli stati fino a quando non trova l'unico elemento che accetta un nodo interno di un albero. Esso prevede due cicli per accettare tutti i figli dell'albero tramite gli automi sheave; questi cicli sono seguiti dalla valutazione della formula di Presburger. L'algoritmo è mostrato in Figura 2.6.

```

MEM( $(n_0 \rightarrow t_0, \dots, n_k \rightarrow t_k), S$ )
1  let  $(\phi, (r_0[S_0], \dots, r_l[S_l])) = \Gamma(S)$ 
2  let  $a = \mathbf{new\ int}[l + 1]$ 
3  for  $i = 0$  to  $l$ 
4  do  $a[i] := 0$ 
5  for  $i = 0$  to  $k$ 
6  do for  $j = 0$  to  $l$ 
7      do if  $n_i \in r_j \wedge \text{MEM}(t_i, S_j)$ 
8          then  $a[j] := a[j] + 1$ 
9              break
10  $\langle c[0], \dots, c[l] \rangle \models \phi$ 

```

Figura 2.6: *Algoritmo per la validazione in Harmony.*

Il costo dell'algoritmo è esponenziale per due cause; la prima è dovuta alla valutazione della soddisfacibilità della formula di Presburger, mentre la seconda è dovuta all'esplorazione non deterministica degli stati.

2.9.2 Xerces

Xerces è un progetto software che offre una robusta implementazione di un parser XML supportato da molti linguaggi di programmazione e piattaforme.

L'algoritmo di validazione di questo strumento è molto più semplice di quello visto in Harmony in quanto non supporta il disordine dei figli negli alberi XML. L'algoritmo è suddiviso in due fasi: la prima fase è caratterizzata da una costruzione interna del tipo XML tramite lettura della DTD o dello XML Schema di riferimento, mentre durante la seconda fase provvede a leggere i dati XML e a controllare tutti i vincoli definiti nel tipo.

Xerces è incluso nella libreria del linguaggio di programmazione Java largamente utilizzato tra i programmatori e con ottime prestazioni. Nonostante non sia presente l'operatore di disordine, che introdurremo nei capitoli seguenti, è stato utilizzato come punto di riferimento con cui confrontarci.

Capitolo 3

Tipi e vincoli

In questo capitolo parleremo dei due linguaggi utilizzati nei capitoli successivi. Il primo è utilizzato per definire le espressioni regolari che rappresentano i tipi, il secondo è utilizzato per definire i vincoli sui tipi. Le espressioni regolari, che vedremo, differiscono da quelle viste nel capitolo precedente per l'aggiunta di un nuovo operatore denominato *interleaving*, da queste espressioni ricaveremo un insieme di vincoli che, a sua volta, utilizzeremo nei capitoli successivi per delineare gli algoritmi di inclusione e validazione.

3.1 Il linguaggio dei tipi

Il linguaggio sulle espressioni regolari (che da ora in poi chiameremo tipi) comprende gli operatori di conteggio, unione, concatenazione e interleaving per stringhe (parole) di un alfabeto finito Σ .

L'operatore di interleaving è definito nel modo seguente.

Definizione 3.1. L'insieme risultante dall'interleaving di due parole $v, w \in \Sigma^*$, o due linguaggi $L_1, L_2 \subseteq \Sigma^*$ è definito come:

$$\begin{aligned} v \& w &=_{def} \{v_1 \cdot w_1 \cdot \dots \cdot v_n \cdot w_n \\ &\quad | v_1 \cdot \dots \cdot v_n = v, w_1 \cdot \dots \cdot w_n = w, v_i \in \Sigma^*, w_i \in \Sigma^*, n > 0\} \\ L_1 \& L_2 &=_{def} \bigcup_{w_1 \in L_1, w_2 \in L_2} w_1 \& w_2 \end{aligned}$$

dove ogni v_i o w_i può essere la stringa vuota.

Esempio 3.2. Consideriamo $(ab)\&(XY)$; esso contiene tutte le possibili permutazioni di $abXY$ dove a precede b e X precede Y , ovvero:

$$(ab)\&(XY) = \{abXY, aXbY, aXYb, XabY, XaYb, XYab\}$$

3.1.1 Sintassi

La sintassi dei tipi è data da:

$$T ::= \epsilon \mid a[m..n] \mid T + T \mid T \cdot T \mid T\&T$$

Con ϵ indichiamo l'insieme che contiene solo la stringa vuota. Con $a[m..n]$ indichiamo j ripetizioni del simbolo a dove $m \leq j \leq n$, $a \in \Sigma$, $m \in (\mathbb{N} \setminus \{0\})$, $n \in (\mathbb{N} \setminus \{0\} \cup \{*\})$, $m \leq n$. Con $\mathbb{N} \setminus \{0\} \cup \{*\}$ indichiamo l'insieme di tutti i numeri naturali, escluso lo zero, dove vale $\forall n \in \mathbb{N} : n \leq *$, mentre con $T + T$, $T \cdot T$ e $T\&T$ indichiamo rispettivamente l'operatore di unione, concatenazione e interleaving.

Esempio 3.3. Consideriamo l'espressione regolare $aa + (b \cdot cc)$ che rappresenta l'insieme delle parole $\{aa, bcc\}$; con la sintassi dei tipi appena definita scriveremo $a[2..2] + (b[1..1] \cdot c[2..2])$, mentre l'espressione regolare $a(a^*) + (b \cdot cc(c^*))$ che rappresenta l'insieme $\{a, aa, aaa, \dots\} \cup \{bcc, bccc, bbccccc, \dots\}$ è derivata da $a[1..*] + (b[1..1] \cdot c[2..*])$.

Questa sintassi non permette di scrivere $a[0..n]$, ma utilizzeremo questa notazione per indicare $a[1..n] + \epsilon$; ulteriore limitazione è costituita dal fatto che l'operatore di conteggio può essere applicato soltanto a simboli. Per rappresentare l'espressione regolare $(a_1 + \dots + a_n)^*$ è necessario scrivere $(a_1[0..*] \& \dots \& a_n[0..*])$.

Esempio 3.4. Consideriamo l'espressione regolare $a^* + b^*$ che rappresenta parole composte da un numero arbitrario di sole a oppure di sole b (anche di lunghezza zero); tale espressione è rappresentabile semplicemente con $a[0..*] + b[0..*]$ che significa $(a[1..*] + \epsilon) + (b[1..*] + \epsilon)$.

3.1.2 Semantica

Per definire la semantica di questo linguaggio utilizziamo le definizioni standard delle espressioni regolari di unione e concatenazione su stringhe e linguaggi.

Definizione 3.5. Per ogni stringa w , $S(w)$ è l'insieme di tutti i simboli che compaiono in w . Per ogni tipo T , $\text{Atoms}(T)$ è l'insieme di tutti gli atomi $a [m..n]$ contenuti in T e $S(T)$ è l'insieme di tutti i simboli che compaiono in T .

Esempio 3.6. Consideriamo la stringa $abba$; l'insieme definito da $S(w)$ è $\{a, b\}$.

Consideriamo adesso il tipo $(a [1..*] + \epsilon) \cdot b [2..*]$, l'insieme $S(T)$ è $\{a, b\}$, mentre l'insieme $\text{Atoms}(T)$ è $\{a [1..*], b [2..*]\}$.

La semantica dei tipi è definita come:

$$\begin{aligned} \llbracket \epsilon \rrbracket &= \{\epsilon\} \\ \llbracket a [m..n] \rrbracket &= \{w \mid S(w) = \{a\}, m \leq |w| \leq n\} \\ \llbracket T_1 + T_2 \rrbracket &= \llbracket T_1 \rrbracket \cup \llbracket T_2 \rrbracket \\ \llbracket T_1 \cdot T_2 \rrbracket &= \llbracket T_1 \rrbracket \cdot \llbracket T_2 \rrbracket \\ \llbracket T_1 \&T_2 \rrbracket &= \llbracket T_1 \rrbracket \&\llbracket T_2 \rrbracket \end{aligned}$$

Dalla semantica dei tipi si può osservare che non è possibile rappresentare l'insieme vuoto (che non contiene nessuna stringa), ma è possibile rappresentare la stringa vuota e l'insieme che la contiene.

Definizione 3.7. $N(T)$ è un predicato sui tipi che indica la presenza o meno della stringa vuota nell'insieme definito dal tipo. $N(T)$ è definito come:

$$\begin{aligned} N(\epsilon) &= \text{true} \\ N(a [m..n]) &= \text{false} \\ N(T_1 + T_2) &= N(T_1) \vee N(T_2) \\ N(T_1 \cdot T_2) &= N(T_1) \wedge N(T_2) \\ N(T_1 \&T_2) &= N(T_1) \wedge N(T_2) \end{aligned}$$

Lemma 3.8. $\epsilon \in \llbracket T \rrbracket$ se e solo se $N(T)$.

Esempio 3.9. Consideriamo il tipo $a [1..2] \cdot b [1..2]$; la sua semantica è rappresentata dall'insieme $\{ab, aab, abb, aabb\}$ e il predicato $N(T) = \text{false} \wedge \text{false} = \text{false}$. Non valendo il predicato $N(T)$ risulta $\epsilon \notin \llbracket T \rrbracket$.

Consideriamo adesso il tipo $a [0..2] \cdot b [0..2]$; la semantica è rappresentata dall'insieme $\{\epsilon, a, aa, b, ab, aab, bb, abb, aabb\}$ e il predicato $N(T) = \text{true} \wedge \text{true} = \text{true}$. Valendo il predicato $N(T)$ risulta $\epsilon \in \llbracket T \rrbracket$.

Diamo infine la definizione di *conflict-free* che assumeremo soddisfatta da ogni tipo di cui parleremo successivamente.

Definizione 3.10 (Tipi conflict-free). Un tipo T è detto *conflict-free* se per ogni sottoespressione $U + V$, $U \cdot V$ e $U \& V$ abbiamo $S(U) \cap S(V) = \emptyset$.

Equivalentemente possiamo dire che un tipo T è conflict-free se per ogni coppia di sottotermini $a [m..n]$ e $a' [m'..n']$ che occorrono in T abbiamo $a \neq a'$. Oppure che per ogni simbolo $s \in S(T)$, s occorre in esattamente un sottoterminale $a [m..n]$.

Esempio 3.11. Consideriamo il tipo $(a [1..1] \cdot b [1..1]) + a [1..1]$: esso non è conflict-free poiché $S(a [1..1] \cdot b [1..1]) \cap S(a [1..1]) = \{a\} \neq \emptyset$.

Consideriamo adesso il tipo $a [1..1] \cdot b [0..1]$, che genera lo stesso linguaggio: esso è conflict-free in quanto $S(a [1..1]) \cap S(b [0..1]) = \emptyset$.

3.2 Il linguaggio dei vincoli

I vincoli sono una rappresentazione corretta e completa di un tipo che utilizzeremo nei capitoli successivi per gli algoritmi di inclusione e validazione. I vincoli sono espressi con la seguente sintassi, dove $a, b \in \Sigma$, $A, B \subseteq \Sigma$, $m \in \mathbb{N} \setminus \{0\}$, $n \in (\mathbb{N} \setminus \{0\} \cup \{*\})$ e $m \leq n$:

$$F ::= A^+ \mid A^+ \Rightarrow B^+ \mid a?[m..n] \mid \text{upper}(A) \mid a \prec b \mid F \wedge F' \mid \mathbf{true}$$

Introduciamo una relazione di soddisfacibilità di una parola w rispetto a un vincolo F , che indicheremo come $w \models F$, e diremo che w soddisfa F quando:

- $w \models A^+$ se e solo se esiste un simbolo $a \in A$ che appare in w ;
- $w \models A^+ \Rightarrow B^+$ se e solo se, quando $w \models A^+$ allora $w \models B^+$;
- $w \models a?[m..n]$ se e solo se, quando il simbolo a appare in w , allora appare almeno m volte e, se $n \neq *$, al più n volte;
- $w \models \text{upper}(A)$ se e solo se tutti i simboli di w sono contenuti nell'insieme A ;

- $w \models a \prec b$ se e solo se, quando i simboli a e b compaiono entrambi in w , a non può comparire dopo b ;
- $w \models F \wedge F'$ se e solo se $w \models A^+$ e $w \models B^+$;
- $w \models \mathbf{true}$ per qualsiasi parola w .

Osservazione 3.12. Il vincolo A^+ è monotono; ciò significa che se prendiamo $w \models A^+$ con w una sottostringa di w' implica che $w' \models A^+$.

Il vincolo $\text{upper}(A)$ è anti-monotono; ciò significa che se prendiamo $w \models \text{upper}(A)$, con w' una sottostringa di w , implica che $w' \models \text{upper}(A)$. Anche il vincolo $a \prec b$ è anti-monotono; infatti se prendiamo $w \models a \prec b$, con w' una sottostringa di w , implica che $w' \models a \prec b$.

Vediamo una serie di esempi, sia positivi che negativi, di soddisfacibilità secondo le definizioni che abbiamo appena dato.

Esempio 3.13. $dab \models \{a, b, c\}^+$ e $ca \models \{a, b, c\}^+$ in quanto esiste almeno un simbolo (in questo caso a) che appartiene all'insieme $\{a, b, c\}$, mentre $fd \not\models \{a, b, c\}^+$ in quanto non esiste nessun simbolo della parola presente in $\{a, b, c\}$. Inoltre, abbiamo $\epsilon \not\models A^+$ e $w \not\models \emptyset^+$ perché, quando l'insieme di simboli che compone la parola w o l'insieme del vincolo A^+ è vuoto, non può esistere un simbolo che appare in w ed è contenuto in A .

Esempio 3.14. $ca \models \text{upper}(\{a, b, c\})$ in quanto l'insieme dei simboli $\{ca\}$, che compongono la parola, è incluso nell'insieme $\{a, b, c\}$, mentre $d \not\models \text{upper}(\{a, b, c\})$ e $dab \not\models \text{upper}(\{a, b, c\})$ perché esiste almeno un simbolo (in questo caso d) che non appartiene all'insieme $\{a, b, c\}$. Inoltre, abbiamo $\epsilon \models \text{upper}(A)$ e $\epsilon \models \text{upper}(\emptyset)$ perché l'insieme che compone la parola ϵ è l'insieme vuoto che è incluso in qualsiasi insieme.

Esempio 3.15. $ca \models b?[2..5]$ perché il simbolo b non compare nella parola, $cbab \models b?[2..5]$ e $bbcbab \models b?[2..5]$ perché compare un numero di volte compreso tra due e cinque, mentre $cba \not\models b?[2..5]$ e $bbcbabb \not\models b?[2..5]$ poiché b compare un numero di volte non compreso tra due e cinque.

Esempio 3.16. $ca \models a \prec b$, $cb \models a \prec b$ e $\epsilon \models a \prec b$ perché i simboli a e b non compaiono contemporaneamente nella parola, $aacb \models a \prec b$ poiché a non compare mai dopo il simbolo b , mentre $caba \not\models a \prec b$, $cba \not\models a \prec b$ e $cbaa \not\models a \prec b$ perché esiste almeno una occorrenza di a dopo il simbolo b .

Durante la definizione dei vincoli utilizzeremo, al fine di rendere i vincoli più compatti, le seguenti abbreviazioni non presenti nella sintassi data ma esprimibili nel modo seguente:

$$\begin{aligned}
A^+ \Leftrightarrow B^+ &=_{def} A^+ \Rightarrow B^+ \wedge B^+ \Rightarrow A^+ \\
a \prec \succ b &=_{def} (a \prec b) \wedge (b \prec a) \\
A \prec B &=_{def} \bigwedge_{a \in A, b \in B} a \prec b \\
A \prec \succ B &=_{def} \bigwedge_{a \in A, b \in B} a \prec \succ b \\
\mathbf{false} &=_{def} \emptyset^+ \\
A^- &=_{def} A^+ \Rightarrow \emptyset^+
\end{aligned}$$

Definiamo infine alcune proposizioni sulle abbreviazioni presentate.

Proposizione 3.17. $w \models a \prec \succ b \Leftrightarrow a$ e b non compaiono entrambi nella parola w , ovvero se vale $w \models a \prec b \wedge b \prec a$.

Proposizione 3.18. $w \models A \prec \succ B \Leftrightarrow w \not\models A^+ \wedge B^+$.

Proposizione 3.19. $w \models A^- \Leftrightarrow w \not\models A^+$.

3.3 Estrazione dei vincoli

In questa sezione descriveremo come estrarre un insieme di vincoli da un tipo T .

A ogni tipo associamo cinque insiemi di vincoli che ogni parola appartenente al tipo deve soddisfare. Questi vincoli si suddividono in due famiglie più grandi: i vincoli piatti e i vincoli interni.

Vincoli piatti

I vincoli piatti dipendono esclusivamente dalle foglie dell'albero sintattico di T e sono delle seguenti tipologie:

- limite inferiore: se il tipo T non contiene la stringa vuota (non vale il predicato $N(T)$), ogni parola w deve includere almeno un simbolo di $S(T)$ che contiene tutti i simboli presenti in T ;
- cardinalità: ogni simbolo a della parola w deve comparire il giusto numero di occorrenze;
- limite superiore: ogni simbolo a presente nella parola w deve essere contenuto in $S(T)$, quindi non è possibile avere parole con simboli in più rispetto a quelli specificati nel tipo.

Diamo adesso una definizione formale dei vincoli piatti dove con $S^+(T)$ indicheremo $(S(T))^+$.

Definizione 3.20 (Vincoli piatti).

Limite inferiore:

$$SIf(T) =_{def} \begin{cases} S^+(T) & \text{se non vale } N(T) \\ \mathbf{true} & \text{se vale } N(T) \end{cases}$$

Cardinalità:

$$\text{ZeroMinMax}(T) =_{def} \bigwedge_{a[m..n] \in \text{Atoms}(T)} a?[m..n]$$

Limite superiore:

$$\text{upperS}(T) =_{def} \text{upper}(S(T))$$

Vincoli piatti:

$$\mathcal{FC}(T) =_{def} SIf(T) \wedge \text{ZeroMinMax}(T) \wedge \text{upperS}(T)$$

Esempio 3.21. Consideriamo seguente il tipo T :

$$(a [1..2] \&b [0..1]) \cdot ((c [1..1] \cdot d [2..2]) + (e [1..3] \&f [2..3]))$$

Il predicato $N(T)$ è falso quindi abbiamo il vincolo di limite inferiore $S^+(T) = \{a, b, c, d, e, f\}^+$, abbiamo sei vincoli di cardinalità relativi agli atomi $a?[1..2] \wedge b?[1..1] \wedge c?[1..1] \wedge d?[2..2] \wedge e?[1..3] \wedge f?[2..3]$; infine abbiamo il limite superiore espresso con il vincolo $\text{upper}(S(T)) = \text{upper}(\{a, b, c, d, e, f\})$.

L'insieme dei vincoli piatti risulta:

$$\mathcal{FC}(T) = \{a, b, c, d, e, f\}^+ \wedge a?[1..2] \wedge b?[1..1] \wedge c?[1..1] \wedge d?[2..2] \wedge e?[1..3] \wedge f?[2..3] \wedge \text{upper}(\{a, b, c, d, e, f\})$$

Vincoli interni

I vincoli interni dipendono dalla struttura interna del tipo T e sono delle seguenti tipologie:

- **co-occorrenza:** se il tipo T contiene un sottoterminale $T_1 \cdot T_2$ o $T_1 \& T_2$, in ogni parola w se compare un simbolo contenuto in $S(T_1)$ deve comparire almeno un simbolo contenuto in $S(T_2)$, a meno che valga il predicato $N(T_2)$; analogamente, se compare un simbolo contenuto in $S(T_2)$ deve comparire almeno un simbolo contenuto in $S(T_1)$, a meno che valga il predicato $N(T_1)$;
- **ordine:** se il tipo T contiene un sottoterminale $T_1 \cdot T_2$, in ogni parola w non può comparire un simbolo contenuto in $S(T_2)$ seguito da un simbolo contenuto in $S(T_1)$;
- **esclusione:** se il tipo T contiene un sottoterminale $T_1 + T_2$, in ogni parola w non può comparire un simbolo contenuto in $S(T_1)$ insieme a un simbolo contenuto in $S(T_2)$.

Diamo adesso una definizione formale dei vincoli interni, dove i vincoli di esclusione e di ordine sono raggruppati in quanto esprimibili con lo stesso operatore della sintassi vista precedentemente.

Definizione 3.22 (Vincoli interni).

Co-occorrenza:

$$\begin{aligned}
 \mathcal{CC}(T_1 + T_2) &=_{def} \mathcal{CC}(T_1) \wedge \mathcal{CC}(T_2) \\
 \mathcal{CC}(T_1 \cdot T_2) &=_{def} \mathcal{CC}(T_1 \& T_2) =_{def} (S^+(T_1) \Rightarrow SIf(T_2)) \wedge \\
 &\quad (S^+(T_2) \Rightarrow SIf(T_1)) \wedge \\
 &\quad \mathcal{CC}(T_1) \wedge \mathcal{CC}(T_2) \\
 \mathcal{CC}(\epsilon) &=_{def} \mathcal{CC}(a[m..n]) =_{def} \mathbf{true}
 \end{aligned}$$

Ordine e esclusione

$$\begin{aligned}
 \mathcal{OC}(T_1 + T_2) &=_{def} (S(T_1) \prec \succ S(T_2)) \wedge \\
 &\quad \mathcal{OC}(T_1) \wedge \mathcal{OC}(T_2) \\
 \mathcal{OC}(T_1 \cdot T_2) &=_{def} (S(T_1) \prec S(T_2)) \wedge \\
 &\quad \mathcal{OC}(T_1) \wedge \mathcal{OC}(T_2)
 \end{aligned}$$

$$\begin{aligned}\mathcal{OC}(T_1 \& T_2) &=_{def} \mathcal{OC}(T_1) \wedge \mathcal{OC}(T_2) \\ \mathcal{OC}(\epsilon) &=_{def} \mathcal{OC}(a[m..n]) =_{def} \mathbf{true}\end{aligned}$$

Vincoli interni:

$$\mathcal{NC}(T) =_{def} \mathcal{CC}(T) \wedge \mathcal{OC}(T)$$

Esempio 3.23. Consideriamo seguente il tipo T :

$$(a[1..3] \cdot b[0..1]) + (c[2..2] \& d[1..2])$$

I vincoli di co-occorrenza sono:

$$\begin{aligned}\mathcal{CC}(T) &= \mathcal{CC}(a[1..3] \cdot b[0..1]) \wedge \mathcal{CC}(c[2..2] \& d[1..2]) \\ &= (\{b\}^+ \Rightarrow \{a\}^+) \wedge (\{c\}^+ \Rightarrow \{d\}^+) \wedge (\{d\}^+ \Rightarrow \{c\}^+) \wedge \\ &\quad \mathcal{CC}(a[1..3]) \wedge \mathcal{CC}(b[0..1]) \wedge \mathcal{CC}(c[2..2]) \wedge \mathcal{CC}(d[1..2]) \\ &= (\{b\}^+ \Rightarrow \{a\}^+) \wedge (\{c\}^+ \Rightarrow \{d\}^+) \wedge (\{d\}^+ \Rightarrow \{c\}^+)\end{aligned}$$

dove il vincolo **true** viene omissso e il vincolo $\{a\}^+ \Rightarrow \{b\}^+$ non è presente in quanto nel sottoalbero $b[0..1]$ vale il predicato $N(b[0..1])$ e $\{a\}^+ \Rightarrow \mathbf{true}$ è equivalente a **true** che quindi viene omissso.

I vincoli di ordine sono:

$$\begin{aligned}\mathcal{OC}(T) &= (\{a, b\} \prec \succ \{c, d\}) \wedge \mathcal{OC}(a[1..3] \cdot b[0..1]) \wedge \mathcal{OC}(c[2..2] \& d[1..2]) \\ &= (\{a, b\} \prec \succ \{c, d\}) \wedge (\{a\} \prec \{b\}) \wedge \\ &\quad \mathcal{OC}(a[1..3]) \wedge \mathcal{OC}(b[0..1]) \wedge \mathcal{OC}(c[2..2]) \wedge \mathcal{OC}(d[1..2]) \\ &= (\{a, b\} \prec \succ \{c, d\}) \wedge (\{a\} \prec \{b\})\end{aligned}$$

dove il vincolo **true** viene omissso.

L'insieme dei vincoli interni risulta:

$$\begin{aligned}\mathcal{NC}(T) &= (\{b\}^+ \Rightarrow \{a\}^+) \wedge (\{c\}^+ \Rightarrow \{d\}^+) \wedge (\{d\}^+ \Rightarrow \{c\}^+) \wedge \\ &\quad (\{a, b\} \prec \succ \{c, d\}) \wedge (\{a\} \prec \{b\})\end{aligned}$$

3.4 Correttezza e completezza dei vincoli

Dato un tipo definito secondo il linguaggio presentato e il rispettivo insieme di vincoli vale il seguente teorema (per la dimostrazione vedere [GCS07]).

Teorema 3.24. Dato T un qualsiasi tipo conflict-free, vale la seguente proprietà:

$$w \in \llbracket T \rrbracket \quad \Leftrightarrow \quad w \models \mathcal{FC}(T) \wedge \mathcal{NC}(T)$$

Il Teorema 3.24 ci mostra che i vincoli estratti da un tipo conflict-free T sono una rappresentazione corretta e completa; possiamo quindi utilizzarli per definire degli algoritmi efficienti per risolvere i problemi di inclusione e appartenenza che vedremo nei capitoli seguenti.

Capitolo 4

Inclusione

In questo capitolo parleremo dell'approccio teorico utilizzato per risolvere il problema di inclusione tra i tipi T e U e, successivamente, vedremo due diversi algoritmi. Descriveremo, infine, un ulteriore algoritmo, che rappresenta una ottimizzazione, studiata per comportarsi meglio quando i tipi T e U sono strutturalmente simili.

4.1 Sistema di deduzione

In questa sezione parleremo del sistema di deduzione utilizzato per formalizzare il funzionamento dell'algoritmo di inclusione utilizzando i vincoli dei tipi. Il sistema di deduzione è suddiviso in più sistemi distinti, dove ogni sistema opera su una specifica classe di vincoli. Questo sistema è sufficientemente potente per decidere l'inclusione tra due tipi.

Il sistema di deduzione \vdash_x è definito da un insieme di regole della forma $F_1 \wedge \dots \wedge F_n \vdash_x F$, dove la notazione $F_1 \wedge \dots \wedge F_m \vdash_x F'_1 \wedge \dots \wedge F'_n$ indica che da $F_1 \dots F_m$ possiamo dedurre $F'_1 \dots F'_n$ applicando più volte le regole corrispondenti. Per compattare le regole utilizzeremo spesso la notazione A al posto di A^+ .

4.1.1 Vincoli piatti

Introduciamo il sistema per dedurre quando i vincoli piatti di un tipo T_1 implicano tutti i vincoli piatti di un tipo T_2 .

Definizione 4.1 ($T_1 \vdash_{flat} T_2$). Siano T_1 e T_2 due tipi. Diremo che da T_1 si deriva T_2 quando:

$$\begin{aligned} T_1 \vdash_{flat} T_2 &\Leftrightarrow_{\text{def}} \\ (a?[m..n] \in \text{Atoms}(T_1) &\Rightarrow \exists m' \leq m, n' \geq n. a[m'..n'] \in \text{Atoms}(T_2)) \\ \wedge (N(T_1) &\Rightarrow N(T_2)) \end{aligned}$$

Osservazione 4.2. Se vale $T_1 \vdash_{flat} T_2$, allora abbiamo $S(T_1) \subseteq S(T_2)$.

Esempio 4.3. Consideriamo i seguenti tipi $T_1 = b[1..2] \cdot c[1..1]$ e $T_2 = a[1..2] + (b[0..3] \&c[1..1])$; abbiamo $T_1 \vdash_{flat} T_2$ in quanto:

- per l'atomo $b[1..2]$ di T_1 abbiamo il rispettivo atomo $b[1..3]$ in T_2 (ricordiamo che $b[0..3] = b[1..3] + \epsilon$, dove $1 \leq 1$ e $3 \geq 2$;
- per l'atomo $c[1..1]$ di T_1 abbiamo il rispettivo atomo $c[1..1]$ in T_2 , dove $1 \leq 1$ e $1 \geq 1$;
- per il tipo T_1 non vale il predicato $N(T_1)$.

Valgono inoltre i seguenti teoremi dimostrati in [GCS07].

Teorema 4.4 (Correttezza di \vdash_{flat}). Se vale $T_1 \vdash_{flat} T_2$, allora valgono:

1. $T_1 \models \text{SI}(T_2)$;
2. $T_1 \models \text{upperS}(T_2)$;
3. $T_1 \models \text{ZeroMinMax}(T_2)$.

Teorema 4.5 (Completezza di \vdash_{flat}). Se vale $\llbracket T_1 \rrbracket \subseteq \llbracket T_2 \rrbracket$, allora vale $T_1 \vdash_{flat} T_2$.

4.1.2 Vincoli di co-occorrenza

Introduciamo il sistema per dedurre quando i vincoli di co-occorrenza di un tipo T_1 , con l'aggiunta dei suoi vincoli di limite superiore, implicano tutti i vincoli di co-occorrenza di un tipo T_2 .

Definizione 4.6. L'operatore \vdash_{cc} è definito mediante le seguenti regole:

$$\begin{array}{ll}
R : & \vdash_{cc} A \Leftrightarrow AB \\
T : & (A \Leftrightarrow B) \wedge (B \Leftrightarrow C) \vdash_{cc} A \Leftrightarrow C \\
A : & A \Leftrightarrow B \vdash_{cc} AC \Leftrightarrow BC \\
False : & a \notin A : \text{upper}(A) \vdash_{cc} a \Leftrightarrow B
\end{array}$$

dove $AB =_{def} A \cup B$ e $aA =_{def} \{a\} \cup A$.

Le regole R , T e A corrispondono al sistema di Armstrong usato per le deduzioni su dipendenze funzionali dove la parte sinistra e la parte destra sono scambiate. La regola $False$ specifica che, se un vincolo di limite superiore esclude il simbolo a , allora possiamo dedurre qualsiasi B dall'impossibile presenza di a .

La corrispondenza tra le regole RTA e gli assiomi di Armstrong può essere facilmente spiegata: una dipendenza funzionale $X_1, \dots, X_n \Leftrightarrow Y_1, \dots, Y_m$ su una relazione R è una implicazione di congiunzioni $\forall t, u \in R. (P(X_1) \wedge \dots \wedge P(X_n)) \Rightarrow (P(Y_1) \wedge \dots \wedge P(Y_m))$, dove $P(X) =_{def} t.X = u.X$. Un vincolo della forma $\{a_1, \dots, a_n\}^+ \Leftrightarrow \{b_1, \dots, b_m\}^+$ è una implicazione di disgiunzioni $\forall w. (a_1 \in S(w) \vee \dots \vee a_n \in S(w)) \Rightarrow (b_1 \in S(w) \vee \dots \vee b_m \in S(w))$ che, tramite alcune trasformazioni logiche, risulta equivalente a calcolare la chiusura all'indietro di $(Q(b_1) \wedge \dots \wedge Q(b_m)) \Rightarrow (Q(a_1) \wedge \dots \wedge Q(a_n))$, dove $Q(a) =_{def} a \notin S(w)$. Quindi i vincoli di co-occorrenza possono essere manipolati come dipendenze funzionali dopo aver invertito i due membri.

Esempio 4.7. Consideriamo l'insieme di vincoli di co-occorrenza:

$$S = \{\{b\}^+ \Leftrightarrow \{c\}^+, \{c\}^+ \Leftrightarrow \{b\}^+, \{b, c\}^+ \Leftrightarrow \{a\}^+, \{a\}^+ \Leftrightarrow \{b, c\}^+\}$$

La chiusura all'indietro dell'insieme di simboli $\{a\}$ sull'insieme di vincoli S è data dall'insieme $\{a, b, c\}$.

Dalle regole definite nella Definizione 4.6 possiamo derivare altre regole che useremo:

$$\begin{array}{ll}
Down : & A' \subseteq A : A \Leftrightarrow B \vdash_{cc} A' \Leftrightarrow B \\
Up : & B \subseteq B' : A \Leftrightarrow B \vdash_{cc} A \Leftrightarrow B' \\
Union : & (A \Leftrightarrow C) \wedge (B \Leftrightarrow C) \vdash_{cc} AB \Leftrightarrow C \\
Decomp : & AB \Leftrightarrow C \vdash_{cc} A \Leftrightarrow C
\end{array}$$

Analogamente ai vincoli piatti, valgono i seguenti teoremi che sono dimostrati in [GCS07].

Lemma 4.8. Per ogni tipo T e per ogni simbolo $a \in S(T)$, se vale $T \models a \Leftrightarrow B$, allora vale $\mathcal{CC}(T) \vdash_{cc} a \Leftrightarrow B$ utilizzando soltanto le regole R , T e A .

Teorema 4.9 (Correttezza di \vdash_{cc}). Se vale $w \models F$ e vale $F \vdash_{cc} F'$, allora vale $w \models F'$. Se vale $T \models F$ e vale $F \vdash_{cc} F'$, allora vale $T \models F'$.

Teorema 4.10 (Completezza di \vdash_{cc}). Se vale $\llbracket T_1 \rrbracket \subseteq \llbracket T_2 \rrbracket$, allora vale $\text{upperS}(T_1) \wedge \mathcal{CC}(T_1) \vdash_{cc} \mathcal{CC}(T_2)$.

4.1.3 Vincoli di ordine

Descriviamo ora come i vincoli di ordine possono essere dedotti dai vincoli di limite superiore.

Definizione 4.11. L'operatore \vdash_{oc} è definito mediante le seguenti regole:

$$\begin{aligned} \text{FalseL} : b \notin A : \text{upper}(A) \vdash_{oc} b \prec b' \\ \text{FalseR} : b \notin A : \text{upper}(A) \vdash_{oc} b' \prec b \end{aligned}$$

Analogamente ai vincoli piatti e di co-occorrenza, valgono i seguenti teoremi dimostrati in [GCS07].

Teorema 4.12 (Correttezza di \vdash_{oc}). Se vale $w \models F$ e vale $F \vdash_{oc} F'$, allora vale $w \models F'$. Se vale $T \models F$ e vale $F \vdash_{oc} F'$, allora vale $T \models F'$.

Lemma 4.13 (Completezza di \vdash_{oc}). Se vale $\{a, b\} \subseteq S(T)$, dove $a \neq b$, e vale $T \models a \prec b$, allora vale $\mathcal{OC}(T) \vdash_{oc} a \prec b$.

Teorema 4.14 (Completezza di \vdash_{oc} per sottotipi). Se vale $\llbracket T_1 \rrbracket \subseteq \llbracket T_2 \rrbracket$, allora vale $\text{upperS}(T_1) \wedge \mathcal{OC}(T_1) \vdash_{oc} \mathcal{OC}(T_2)$.

4.1.4 Correttezza e completezza

Teorema 4.15 (Correttezza e completezza). Siano T_1 e T_2 due tipi. Vale la seguente proprietà:

$$\begin{aligned} \llbracket T_1 \rrbracket \subseteq \llbracket T_2 \rrbracket \Leftrightarrow & (\text{upperS}(T_1) \wedge \mathcal{CC}(T_1) \vdash_{cc} \mathcal{CC}(T_2)) \wedge \\ & (\text{upperS}(T_1) \wedge \mathcal{OC}(T_1) \vdash_{oc} \mathcal{OC}(T_2)) \wedge \\ & (T_1 \vdash_{flat} T_2) \end{aligned}$$

Il Teorema 4.15, dimostrato in [GCS07], ci dice che il sistema di deduzione è corretto e completo rispetto all'inclusione di due tipi utilizzando i sistemi \vdash_{flat} , \vdash_{cc} e \vdash_{oc} che abbiamo visto.

4.2 Algoritmo base

Il Teorema 4.15 prova che l'inclusione tra tipi conflict-free può essere decisa tramite il sistema di deduzione presentato. Da questo teorema ricaviamo un algoritmo per verificare quando il tipo T è sottotipo di U ovvero quando $T \subseteq U$.

L'algoritmo di inclusione, descritto in Figura 4.1, verifica $T \vdash_{flat} U$ in tempo lineare nella dimensione di T e U . Successivamente l'algoritmo verifica i vincoli di co-occorrenza mediante una semplice estensione all'algoritmo di Beeri-Bernstein per le dipendenze funzionali; infine l'algoritmo verifica che ogni vincolo di $\mathcal{OC}(U)$ sia anche in $\mathcal{OC}(T)$ oppure riguardi un simbolo che non è presente in $S(T)$.

SUB(T, U)

1 **return** FLATIMPLIES(T, U) \wedge COIMPLIES(T, U) \wedge ORDERIMPLIES(T, U)

Figura 4.1: *Algoritmo di inclusione.*

Nell'algoritmo in Figura 4.1 le operazioni in and sono *non strette*: a partire da sinistra verso destra, se un sistema di deduzione restituisce il valore falso il risultato finale è falso e non c'è bisogno di valutare i successivi.

4.2.1 Vincoli piatti

Come abbiamo visto nella Definizione 4.1 l'implicazione dei vincoli piatti viene verificata direttamente. Costruiamo una rappresentazione dei valori m e n relativi al tipo U mediante due tabelle hash Min_U e Max_U , indirizzate con il simbolo a , tale che per ogni $a[m..n] \in Atoms(U)$ abbiamo $Min_U[a] = m$ e $Max_U[a] = n$. Per verificare i vincoli, quindi, analizziamo tutti gli atomi presenti in T e per ogni atomo $a[m..n]$ verifichiamo che $Min_U[a] \leq m$ e $Max_U[a] \geq n$. Infine, verifichiamo che, se vale il predicato $N(T)$ allora valga anche il predicato $N(U)$ (Figura 4.2).

```

FLATIMPLIES( $T, U$ )
1  ( $Min_U[], Max_U[]$ ) := BUILDMINMAXARRAYS( $U$ )
2   $flat$  := (every  $a?[m..n] \in Atoms(T)$ 
3      satisfies ( $Min_U[a] \leq m$ )  $\wedge$  ( $Max_U[a] \geq n$ ))
4       $\wedge$  ( $\neg N(T) \vee N(U)$ );
5  return  $flat$ 

```

Figura 4.2: Algoritmo per la verifica dei vincoli piatti.

Esempio 4.16. Consideriamo i seguenti tipi, rappresentati in Figura 4.3:

$$T = a[1..1] \cdot b[2..2]$$

$$U = (a[1..2] \& b[2..2]) + ((c[1..1] + \epsilon) \cdot d[1..1])$$

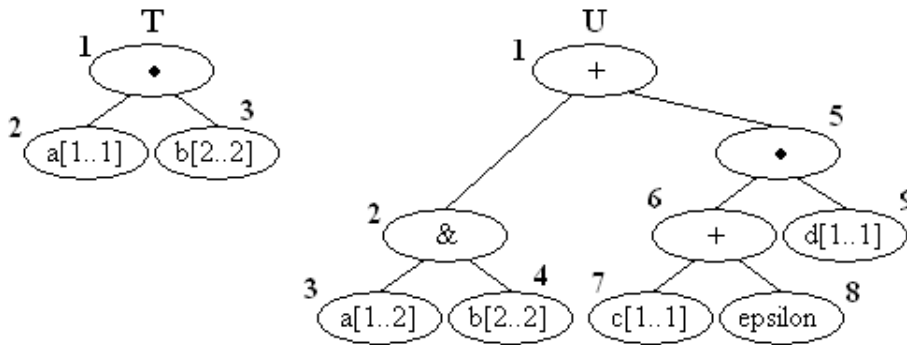


Figura 4.3: Rappresentazione grafica dei tipi dell'Esempio 4.16.

Le tabelle hash Min_U e Max_U , che costruiamo, risultano:

atomo	Min_U	Max_U
$a[1..2]$	1	2
$b[2..2]$	2	2
$c[1..1]$	1	1
$d[1..1]$	1	1

Osserviamo che gli atomi di T , ovvero $a[1..1]$ e $b[2..2]$, hanno valori di n e m compresi nelle tabelle Min_U e Max_U , infine calcoliamo $N(T) = \mathbf{false}$, quindi ϵ

non è compreso tra le parole accettate dal tipo T e non è necessario verificare che sia presente o meno in U .

Il costo dell'algoritmo dipende sia dal numero di atomi di U , per costruire le tabelle hash, sia dal numero di atomi di T per fare il confronto. L'algoritmo risulta quindi lineare con la dimensione di T e U , ovvero con costo $O(n + m)$.

4.2.2 Vincoli di co-occorrenza

Presentiamo, adesso, un algoritmo per verificare quando $\text{upperS}(T_1) \wedge \mathcal{CC}(T_1) \vdash_{cc} \mathcal{CC}(T_2)$. In questo algoritmo utilizzeremo la chiusura all'indietro di un insieme U su un tipo T , definita come il massimo insieme $R \in S(T)$ tale che $\mathcal{CC}(T) \vdash_{cc} R \Rightarrow U$.

L'algoritmo per calcolare la chiusura all'indietro è ottenuto mediante una modifica all'algoritmo di Beeri-Bernstein ed è corretto e completo per le regole R , T , A e $False$.

Dati due tipi T e U con U_i e U_j gli argomenti di un operatore di concatenazione o di interleaving dentro U , utilizzando il Lemma 4.8 e le regole di *Union* e *Decomp*, possiamo dire: $\text{upperS}(T) \wedge \mathcal{CC}(T) \vdash_{cc} S^+(U_j) \Rightarrow S^+(U_i)$ se e solo se $(S(U_j) \cap S(T)) \subseteq \text{TBACKWARDCLOSE}(S(U_i))$, dove *TBACKWARDCLOSE* è l'algoritmo di chiusura all'indietro sul tipo T .

L'algoritmo *TBACKWARDCLOSE*, illustrato in Figura 4.4, richiede che tutti gli n vincoli del sottotipo T , della forma $F = L_i \Rightarrow R_i$, siano codificati in tre strutture dati secondo lo schema seguente:

- un array *SizeOfRight*[] di n interi inizializzato con $\text{SizeOfRight}[i] = k$, se R_i contiene k simboli;
- una tabella hash *IsIn*[], indicizzata sui simboli, che contiene un insieme di interi tale che $\text{IsIn}[a] = \{i \mid a \in R_i\}$;
- un array *Left*[] composto da n insiemi di simboli dove $\text{Left}[i] = L_i$.

L'algoritmo si appoggia su due ulteriori strutture: un array *Found*[], che per ogni vincolo i , indica quanti simboli di R_i sono stati incontrati, inizialmente inizializzato a 0, e un insieme *ToDo* che indica i simboli, presenti nella chiusura, non ancora analizzati.

```

TBACKWARDCLOSE( $A$ )
1  global const  $SizeOfRight[]$ ,  $IsIn[]$ ,  $Left[]$ ;
2   $Result = A$ ;
3   $ToDo = A$ ;
4  while  $ToDo \neq \emptyset$ 
5  do estrai a da ToDo;
6     $ToDo = ToDo - \{a\}$ ;
7    for  $i \in IsIn[a]$ 
8    do  $Found[i] = Found[i] + 1$ 
9      if  $(SizeOfRight[i] = Found[i]) \wedge (Left[i] - Result \neq \emptyset)$ 
10     then  $New = Left[i] - Result$ ;
11            $Result = Result + New$ ;
12            $ToDo = ToDo + New$ ;
13 return  $Result$ 

```

Figura 4.4: Algoritmo per la chiusura all'indietro.

L'algoritmo procede estraendo simboli dall'insieme $ToDo$. L'array $Found[i]$ conta il numero di simboli $a \in R_i$ tali che $F \vdash_{cc} a \Rightarrow A$.

Quando $Found[i] = SizeOfRight[i]$, sappiamo che $F \vdash_{cc} R_i \Rightarrow A$, dato che $L_i \Rightarrow R_i$, allora $F \vdash_{cc} L_i \Rightarrow R$. A questo punto i simboli di L_i che non sono ancora presenti nella chiusura ne entrano a far parte e vengono inseriti nell'insieme $ToDo$.

Esempio 4.17. Consideriamo il tipo U dell'Esempio 4.16 e calcoliamo la chiusura all'indietro dell'insieme di simboli $\{b\}$. L'insieme F è costituito da $\{a \Rightarrow b, b \Rightarrow a, c \Rightarrow d\}$, le strutture dati che l'algoritmo costruisce risultano:

i	$SizeOfRight[]$	$Left[]$	$IsIn[]$	
1	1	$\{a\}$	a	$\{2\}$
2	1	$\{b\}$	b	$\{1\}$
3	1	$\{c\}$	c	$\{\}$
			d	$\{3\}$

Alla prima iterazione, l'algoritmo procede estraendo il simbolo b da $ToDo$ e,

```

TIMPLIES( $U$ )
1  switch ( $U$ )
2    case ( $\epsilon$ ) or ( $a[m..n]$ ) : return true;
3    case ( $U_1 + U_2$ ) : return (TIMPLIES( $U_1$ )  $\wedge$  TIMPLIES( $U_2$ ));
4    case ( $U_1 \cdot U_2$ ) or ( $U_1 \& U_2$ ) :
5      return (TIMPLIES( $U_1$ )  $\wedge$  TIMPLIES( $U_2$ ))
6           $\wedge$  ( $N(U_2) \vee (S(U_1) \cap S(T)) \subseteq \text{TBACKWARDCLOSE}(S(U_2))$ )
7           $\wedge$  ( $N(U_1) \vee (S(U_2) \cap S(T)) \subseteq \text{TBACKWARDCLOSE}(S(U_1))$ )

COIMPLIES( $T, U$ )
1   $SizeOfRight[], IsIn[], Left[] = \text{ENCODECC}(T)$ ;
2  return TIMPLIES( $U$ );

```

Figura 4.5: Algoritmo per la verifica dei vincoli di co-occorrenza.

essendo $IsIn[b] = \{1\}$, incrementando di uno $Found[1]$. Successivamente verifica che $SizeOfRight[i] = Found[i]$ ($1 = 1$) e che $Left[i] - Result = \{a\}$, quindi aggiunge il simbolo a a $ToDo$ e $Result$, che serve per contenere il risultato finale.

Alla seconda iterazione, l'algoritmo estrae il simbolo a , incrementa di uno $Found[2]$ e successivamente verifica che $Left[i] - Result = \emptyset$, quindi non esegue ulteriori operazioni. Infine, l'insieme $ToDo$ è vuoto e l'algoritmo termina restituendo l'insieme di simboli contenuti in $Result$.

Di seguito possiamo osservare l'evoluzione delle strutture dati durante i vari cicli n dell'algoritmo.

n	$Result$	$ToDo$	$Found[]$		
0	$\{b\}$	$\{b\}$	0	0	0
1	$\{b, a\}$	$\{a\}$	1	0	0
2	$\{b, a\}$	$\{a\}$	1	1	0

Nell'algoritmo di chiusura all'indietro, mostrato in Figura 4.4, un simbolo non può comparire in più di $2 * d_T$ vincoli di co-occorrenza, dove d_T è il numero di con-

catenazioni o interleaving interni al tipo T , quindi ogni invocazione dell'algoritmo di chiusura impiega tempo $O(n * d_T)$.

L'algoritmo **TIMPLIES** procede ricorsivamente per verificare che i vincoli di co-occorrenza di U siano soddisfatti dal tipo T . L'algoritmo, quando analizza un sottoalbero di U , procede valutando tre possibili casi:

- se la radice del sottoalbero è ϵ o $a [m..n]$ significa che non ci sono vincoli da verificare, quindi l'algoritmo restituisce il valore **true**;
- se la radice del sottoalbero è $U_1 + U_2$ significa che non ci sono vincoli da verificare, ma l'algoritmo deve verificare vincoli interni rispettivamente a U_1 e U_2 , quindi si richiama ricorsivamente su questi due sottoalberi;
- se la radice del sottoalbero è $U_1 \cdot U_2$ o $U_1 \& U_2$ significa che ci sono vincoli da verificare, inoltre l'algoritmo deve verificare vincoli interni dei sottoalberi U_1 e U_2 , quindi si richiama ricorsivamente su questi due sottoalberi.

Esempio 4.18. Consideriamo il tipo T e U dell'Esempio 4.16. L'algoritmo per verificare i vincoli di co-occorrenza inizia analizzando il nodo 1 dell'albero U in Figura 4.3, riconosce un operatore di unione, quindi procede nel richiamarsi ricorsivamente sui nodi 2 e 5.

Analizzando il nodo 2 l'algoritmo riconosce un operatore di interleaving e si richiama ricorsivamente sui nodi 3 e 4, ritornando immediatamente **true** in quanto atomi. Successivamente verifica che entrambi i figli del nodo 2 hanno il predicato $N(i) = \mathbf{false}$, quindi calcola la chiusura all'indietro di $\{a\}$ e di $\{b\}$, che valgono entrambe $\{a, b\}$, che contiene sia i simboli del figlio destro che del figlio sinistro, quindi le condizioni richieste sono verificate.

Analizzando il nodo 5 l'algoritmo riconosce un operatore di concatenazione e si richiama ricorsivamente sui nodi 6 e 9. Il nodo 6 è composto da una unione, quindi l'algoritmo si richiama sui suoi figli, ma ritorna immediatamente **true** in quanto atomi. Inoltre, il nodo 9 è a sua volta un atomo, quindi l'algoritmo restituisce, anche in questo caso, il valore **true**. Successivamente, l'algoritmo verifica che $N(5) = \mathbf{true}$ e $N(9) = \mathbf{false}$ e calcola la chiusura soltanto per il nodo 9, ovvero dell'insieme $\{d\}$, che vale $\{c, d\}$ e che contiene tutti i simboli del nodo 5. Nel caso in cui la chiusura non avesse contenuto il simbolo c , e quindi i simboli del

nodo 5, l'algoritmo avrebbe comunque restituito il valore **true** in quanto $S(T)$ non contiene tale simbolo.

L'algoritmo termina restituendo un valore **true** dopo aver analizzato tutti i nodi del tipo U .

Nel caso pessimo TIMPLIES (Figura 4.5) invoca la chiusura per ogni operatore del tipo U , quindi impiega tempo $O(n * n * d_T)$, ovvero $O(n^3)$.

4.2.3 Vincoli di ordine

I vincoli di ordine corrispondono alle operazioni di concatenazione e unione all'interno del tipo.

Per ogni coppia di atomi $a[m..n]$ e $b[m'..n']$ nell'albero sintattico di T , sia $LCA_T(a, b)$ il loro minimo antenato comune. Per ogni a e b in $S(T)$, $a \prec b$ se e solo se $LCA_T(a, b)$ è etichettato con l'operatore $+$. Analogamente, $a \prec b \in \mathcal{OC}(T)$ se e solo se $LCA_T(a, b)$ è etichettato con l'operatore $+$, oppure è etichettato con l'operatore \cdot e a precede b in T .

Riassumendo, $\text{upper}S(T) \wedge \mathcal{OC}(T) \vdash_{oc} \mathcal{OC}(U)$ se e solo se, per ogni a e b dove a precede b in $S(U)$, abbiamo:

- se $LCA_U(a, b) = +$, allora $a \notin S(T)$ oppure $b \notin S(T)$ oppure $LCA_T(a, b) = +$;
- se $LCA_U(a, b) = \cdot$, allora $a \notin S(T)$ oppure $b \notin S(T)$ oppure $LCA_T(a, b) = +$ oppure $(LCA_T(a, b) = \cdot$ e a precede b in T).

L'algoritmo, illustrato in Figura 4.6, esegue dapprima le fasi di preprocessing nel quale costruisce LCA_T e LCA_U . Successivamente l'algoritmo procede nel verificare le due precedenti proprietà per tutte le coppie di simboli a e b dove a precede b in $S(U)$, che può essere fatto in tempo $O(n^2)$.

Esempio 4.19. Consideriamo il tipo T e U dell'Esempio 4.16. L'algoritmo per valutare i vincoli di ordine costruisce le strutture dati necessarie per calcolare LCA_T e LCA_U viste nel Capitolo 2. Successivamente, l'algoritmo considera tutte le coppie dell'insieme $S(U) = \{a, b, c, d\}$ calcolando, per ogni coppia, i valori della seguente tabella:

a	b	$LCA_U(a, b)$	$a \in S(T)$	$b \in S(T)$	$LCA_T(a, b)$
a	b	$\&$	true	true	\cdot
a	c	$+$	true	false	
a	d	$+$	false	false	
b	c	$+$	false	false	
b	d	$+$	false	false	
c	d	\cdot	false	false	

L'algoritmo verifica che soltanto i simboli della coppia (a, b) appartengono a $S(T)$, ciò significa che per tutte le altre coppie non vale la condizione $b \in S(U)$, quindi non ci sono vincoli di ordine da soddisfare.

Infine, l'algoritmo verifica che $LCA_U(a, b) = \&$, ovvero il nodo non ha vincoli di ordine, quindi termina e restituisce il valore **true**.

```

ORDERIMPLIES(Type T, Type U)
1  build  $LCA_T$  and  $LCA_U$ 
2  for each leaf  $a$  in  $U$ , leaf  $b$  following  $a$  in  $U$ 
3  do if  $LCA_U(a, b) = + \wedge a \in S(T) \wedge b \in S(T) \wedge LCA_T(a, b) \neq +$ 
4     then return false
5     if  $LCA_U(a, b) = \cdot \wedge a \in S(T) \wedge b \in S(T) \wedge LCA_T(a, b) \neq + \wedge$ 
6        $(LCA_T(a, b) \neq \cdot \vee b$  precedes  $a$  in  $T)$ 
7     then return false
8  return true

```

Figura 4.6: Algoritmo per la verifica dei vincoli di ordine.

Il costo complessivo dell'algoritmo è dato da $O(n^2)$ in quanto, sia usando l'algoritmo LCA che abbiamo proposto con costo $\langle O(n^2), O(1) \rangle$ sia utilizzando quello con costo $\langle O(n), O(1) \rangle$, il costo dell'algoritmo resta quadrato per il controllo di tutte le coppie di simboli a e b in $S(U)$.

4.3 Algoritmo quadratico

In questa sezione presentiamo una variante dell'algoritmo di chiusura all'indietro, avente complessità lineare. Questa variante, denominata *chiusura veloce*, richiede che il tipo di input sia ϵ -normalizzato, ovvero che i sottotipi della forma $T \cdot \epsilon$, $T \& \epsilon$, $\epsilon \cdot T$ e $\epsilon \& T$ siano riscritti con T e tutti i sottotipi della forma $\epsilon + \epsilon$ siano riscritti con ϵ . La ϵ -normalizzazione richiede un tempo lineare e assumiamo sia stata fatta prima di invocare la chiusura veloce.

L'algoritmo procede visitando soltanto i sottoalberi di T che sono contenuti in A_T^\dagger , definito come segue.

Definizione 4.20. Sia $T = C[T']$, T' sottoalbero di T , e si indichi con \otimes gli operatori di concatenazione e interleaving; definiamo A_T^\dagger come:

$$\begin{array}{ll} \forall A, T & \epsilon \in A_T^\dagger \\ T = C[a [m..n]], a \in A & \Rightarrow a [m..n] \in A_T^\dagger \\ T = C[T_1 + T_2], T_1 \in A_T^\dagger, T_2 \in A_T^\dagger & \Rightarrow T_1 + T_2 \in A_T^\dagger \\ T = C[T_1 \otimes T_2], T_1 \in A_T^\dagger, T_2 \in A_T^\dagger & \Rightarrow T_1 \otimes T_2 \in A_T^\dagger \\ T = C[T_1 \otimes T_2], \text{ or } C[T_2 \otimes T_1], \neg N(T_1), T_1 \in A_T^\dagger & \Rightarrow T_1 \otimes T_2 \in A_T^\dagger \end{array}$$

Osservazione 4.21. Per ogni $C[T]$, se $T' \in A_T^\dagger$ allora $T' \in A_{C[T]}^\dagger$.

Lemma 4.22. Per ogni tipo T , dove con A indichiamo in insieme di simboli, vale la seguente proprietà:

$$T \models A^+ \Rightarrow T \in A_T^\dagger$$

Dimostrazione Dimostriamo per induzione strutturale:

$$T \models A^+ \Rightarrow T \in A_T^\dagger$$

$T = \epsilon$:

Ovvio per definizione di A_T^\dagger .

$T = b [m..n]$:

Da $T \models A$ otteniamo $b \in A$, quindi $b [m..n] \in A_{b[m..n]}^\dagger$ per definizione di A_T^\dagger .

$T = T_1 + T_2$:

Da $T \models A$ otteniamo $T_1 \models A$ e $T_2 \models A$ che, per induzione, implica $T_1 \in A_{T_1}^\dagger$ e $T_2 \in A_{T_2}^\dagger$, quindi $T_1 + T_2 \in A_{T_1+T_2}^\dagger$.

$T = T_1 \otimes T_2$:

Definiamo $A_1 = (A \cap S(T_1))$ e $A_2 = (A \cap S(T_2))$. Vale $T_1 \models A_1$ o $T_2 \models A_2$, altrimenti esisterebbe $w_1 \in \llbracket T_1 \rrbracket$ e $w_2 \in \llbracket T_2 \rrbracket$ dove $w_1 \not\models A_1$ e $w_2 \not\models A_2$ quindi $w_1 \cdot w_2 \not\models A_1 \cup A_2$. Assumiamo senza perdita di generalità che $T_1 \models A_1$, di conseguenza $\epsilon \notin \llbracket T_1 \rrbracket$, quindi $\neg N(T_1)$. Per induzione abbiamo $T_1 \in A_{T_1}^\uparrow$, quindi $T_1 \in A_{T_1 \otimes T_2}^\uparrow$ quindi $T_1 \otimes T_2 \in A_{T_1 \otimes T_2}^\uparrow$. ■

Teorema 4.23. Per ogni tipo T e per ogni simbolo $a \in S(T)$, dove con A indichiamo in insieme di simboli e con $S(A_T^\uparrow)$ indichiamo $\bigcup_{U \in A_T^\uparrow} S(y)$, vale la seguente proprietà:

$$T \models a \Leftrightarrow A \Leftrightarrow a \in S(A_T^\uparrow)$$

Dimostrazione Dimostriamo per induzione strutturale:

$$T \models a \Leftrightarrow A \Rightarrow a \in S(A_T^\uparrow)$$

$T = \epsilon$:

Banale in quanto non esiste nessun a con $a \in S(T)$.

$T = b[m..n]$:

Da $a \in S(T)$ otteniamo $a = b$. Da $T \models b \Leftrightarrow A$ otteniamo $b_1 \cdot \dots \cdot b_m \models A$, quindi $b \in A$, quindi $b[m..n] \in A_{b[m..n]}^\uparrow$.

$T = T_1 + T_2$:

Senza perdita di generalità assumiamo $a \in S(T_1)$.

Da $T \models a \Leftrightarrow A$ otteniamo $T_1 \models a \Leftrightarrow A$ che, per induzione, implica $a \in S(A_{T_1}^\uparrow)$, quindi $a \in S(A_T^\uparrow)$.

$T = T_1 \otimes T_2$:

Senza perdita di generalità assumiamo $a \in S(T_1)$.

Distinguiamo due casi: se $T_2 \not\models A$, ovvero $\exists w_2 \in \llbracket T_2 \rrbracket$ tale che $w_2 \not\models A$, quindi w_2 non contiene nessun simbolo di A . Per ogni stringa $w_1 \in \llbracket T_1 \rrbracket$, $w_1 \cdot w_2 \in \llbracket T_1 \otimes T_2 \rrbracket$, quindi per ipotesi abbiamo $w_1 \cdot w_2 \models a \Leftrightarrow A$. Da $w_2 \not\models A$ deriviamo $w_1 \not\models a \Leftrightarrow A$, quindi $T_1 \models a \Leftrightarrow A$ che, per induzione, implica $a \in S(A_{T_1}^\uparrow)$, quindi $a \in S(A_T^\uparrow)$.

Altrimenti, $T_2 \models A$, ovvero $\forall w_2 \in \llbracket T_2 \rrbracket$ abbiamo $w_2 \models A$. Da $T_2 \models A$ ricaviamo $\epsilon \notin \llbracket A \rrbracket$, quindi $\neg N(T_2)$. Utilizzando il Lemma 4.22 e $T_2 \models A$ abbiamo $T_2 \in A_{T_2}^\uparrow$, quindi $T_2 \in A_{T_1 \otimes T_2}^\uparrow$, quindi $T_1 \otimes T_2 \in A_{T_1 \otimes T_2}^\uparrow$. ■

Dimostrazione Dimostriamo:

$$a \in S(A_T^\dagger) \Rightarrow T \models a \Leftrightarrow A$$

$T = \epsilon$:

Banale in quanto $S(T) = \emptyset$.

$T = b[m..n]$:

Da $a \in S(T)$ otteniamo $a = b$. Da $a \in S(A_T^\dagger)$ otteniamo $a \in A$ quindi per la proprietà riflessiva degli assiomi di Armstrong vale $T \models a \Leftrightarrow A$.

$T = T_1 + T_2$:

Senza perdita di generalità assumiamo $a \in S(T_1)$. Essendo $a \notin S(T_2)$ vale $T_2 \models a \Leftrightarrow A$. Da $a \in S(T_1)$ abbiamo $a \in S(A_{T_1}^\dagger)$, quindi per induzione vale $T_1 \models a \Leftrightarrow A$, quindi vale $T \models a \Leftrightarrow A$.

$T = T_1 \otimes T_2$:

Senza perdita di generalità assumiamo $a \in S(T_1)$.

Distinguiamo due casi: se $a \in S(A_{T_1}^\dagger)$ allora vale lo stesso ragionamento di $T_1 + T_2$.

Altrimenti, $a \notin S(A_{T_1}^\dagger)$, abbiamo inoltre $a \notin S(A_{T_2}^\dagger)$, in quanto $a \notin S(T_2)$. Come ipotesi induttiva abbiamo che $T_2 \in S(A_{T_2}^\dagger)$ e $\neg N(T_2)$. Da $\neg N(T_2)$ abbiamo che vale il vincolo $S(T_1) \Leftrightarrow S(T_2)$, da cui, per la completezza dei vincoli di co-occorrenza abbiamo $T \models a \Leftrightarrow S(T_2)$. Da $T_2 \in S(A_{T_2}^\dagger)$ abbiamo $S(T_2) = S(A_{T_2}^\dagger)$ e, per l'ipotesi induttiva, abbiamo $T_2 \models S(T_2) \Leftrightarrow A^+$. Concludendo, abbiamo $T \models a \Leftrightarrow S(T_2) = S(A_{T_2}^\dagger) \Leftrightarrow A^+$ quindi, per transitività, abbiamo $T \models a \Leftrightarrow A^+$.

■

Il Teorema 4.23 ci dice che il risultato della chiusura di A sull'albero T è dato da $S(A_T^\dagger)$. Per calcolarlo ci avvaliamo di un array *Marked*[] che permette di analizzare ogni nodo una sola volta e di ricostruire A_T^\dagger (Figura 4.7).

Il tipo T viene codificato nelle seguenti strutture dati:

- una tabella hash *NodeOfSymbol*[], indicizzata sui simboli, dove, per ogni simbolo a , *NodeOfSymbol*[a] rappresenta il nodo n dell'atomo a [$m..n$] oppure null se il simbolo non esiste;

```

MARK( $d$ )
1  global  $NodeOfSymbol[]$ ,  $Parent[]$ ,  $Oper[]$ ,  $LeftChild[]$ ,  $RightChild[]$ 
2  if  $\neg Marked[d]$ 
3    then  $Marked[d] := true$ 
4      if  $Oper[d] \notin \{a[m..n], \epsilon\}$ 
5        then MARK( $LeftChild[d]$ )
6          MARK( $RightChild[d]$ )

```

Figura 4.7: Procedure di supporto all'algoritmo della Figura 4.8.

- un array $Parent[]$ dove, per ogni nodo d , $Parent[d]$ è null, se d è la radice, oppure una coppia $(d_p, direction)$, dove d_p è il nodo del padre e $direction$ la direzione di d ovvero se d è figlio destro o sinistro di d_p ;
- un array $Oper[]$ dove per ogni nodo d , $Oper[d]$ associa:
 - $true$ se d è un nodo che non da origine a nessun vincolo di co-occorrenza, ovvero se d è un nodo con operatore $+$, oppure un operatore \cdot o $\&$ dove per entrambi i figli T vale il predicato $N(T)$; indicheremo come pedice il numero di figli da analizzare, ovvero:
 - * $true_2$ se d è un nodo con operatore $+$ dove nessun figlio è ϵ oppure se d è un nodo con operatore \cdot o $\&$ dove per entrambi i figli T vale il predicato $N(T)$;
 - * $true_1$ se d è un nodo con operatore $+$ dove un solo figlio è ϵ ;
 - * $true_0$ se d è un atomo $a[m..n]$ oppure ϵ ;
 - \Leftrightarrow se d è un nodo con operatore \cdot o $\&$ dove per nessun figlio T vale il predicato $N(T)$;
 - \Leftarrow se d è un nodo con operatore \cdot o $\&$ dove solo per il figlio di destra T vale il predicato $N(T)$;
 - \Rightarrow se d è un nodo con operatore \cdot o $\&$ dove solo per il figlio di sinistra T vale il predicato $N(T)$;

```

TBACKWARDCLOSE( $A$ )
1  global  $NodeOfSymbol[]$ ,  $Parent[]$ ,  $Oper[]$ ,  $LeftChild[]$ ,  $RightChild[]$ 
2  ( $ToDo$ ,  $Marked[]$ ) := INITLOCAL( $A$ )
3  while  $ToDo \neq \emptyset$ 
4  do pick and remove  $d$  from  $ToDo$ ;
5     ( $parent$ ,  $pos$ ) :=  $Parent[d]$ 
6     if  $Marked[d]$ 
7         then continue
8     else  $Marked[d] := true$ 
9     if  $parent = null$ 
10        then continue
11    switch ( $Oper[parent]$ ,  $pos$ )
12        case ( $true_2$ ,  $-$ ) :  $Oper[parent] := true_1$ 
13        case ( $true_1$ ,  $-$ ) :  $ToDo := ToDo ++ parent$ 
14        case ( $\Leftarrow$ ,  $left$ ) or ( $\Leftrightarrow$ ,  $left$ ) : MARK( $RightChild[parent]$ )
15                                      $ToDo := ToDo ++ parent$ 
16        case ( $\Rightarrow$ ,  $right$ ) or ( $\Leftrightarrow$ ,  $right$ ) : MARK( $LeftChild[parent]$ )
17                                      $ToDo := ToDo ++ parent$ 
18  return  $\{a \mid \exists d Oper[d] = a[m..n] \wedge Marked[d]\}$ 

```

Figura 4.8: Algoritmo per la chiusura all'indietro veloce.

- due array $LeftChild[]$ e $RightChild[]$ che, per ogni nodo d indicano rispettivamente il nodo del figlio sinistro e del figlio destro, oppure null se il nodo d non ha figli.

L'algoritmo, mostrato in Figura 4.8, utilizza una lista di nodi $ToDo$, inizializzata con tutti i nodi dei simboli presenti nell'insieme A del quale vogliamo calcolare la chiusura: un nodo d entra a far parte della lista solo se $T \models S(T[d]) \Leftrightarrow A$, dove $T[d]$ è il sottoalbero di T radicato in d .

Esempio 4.24. Consideriamo il tipo e l'insieme da chiudere dell'Esempio 4.17.

Le strutture dati che l'algoritmo costruisce risultano:

i	$Parent[]$	$Oper[]$	$LeftChild[]$	$RightChild[]$	$NodeOfSymbol[]$	
1	$null$	$true_2$	2	5	a	3
2	$(1, left)$	\Leftrightarrow	3	4	b	4
3	$(2, left)$	$true_0$	$null$	$null$	c	7
4	$(2, right)$	$true_0$	$null$	$null$	d	9
5	$(1, right)$	\Rightarrow	6	9		
6	$(5, left)$	$true_1$	7	8		
7	$(6, left)$	$true_0$	$null$	$null$		
8	$(6, right)$	$true_0$	$null$	$null$		
9	$(5, right)$	$true_0$	$null$	$null$		

Alla prima iterazione, l'algoritmo estrae il nodo 4 dalla lista $ToDo$ e imposta $Marked[4] = true$. Il padre del nodo 4 è rappresentato dalla coppia $(2, right)$ e, essendo $(Oper[parent], pos) = (\Leftrightarrow, right)$, applica la procedura MARK al sottoalbero sinistro radiato nel nodo 3 e aggiunge il nodo 2 alla lista $ToDo$.

Alla seconda iterazione, l'algoritmo estrae il nodo 2 dalla lista $ToDo$ e imposta $Marked[2] = true$. Il padre del nodo 2 è rappresentato dalla coppia $(1, left)$ e, essendo $(Oper[parent], pos) = (true_2, left)$, imposta $Oper[1] = true_1$. Infine, la lista $ToDo$ è vuota, quindi l'algoritmo termina restituendo l'insieme $\{a, b\}$.

Grazie alla costruzione di A_T^\dagger , l'algoritmo analizza ogni nodo al più una volta, quindi richiede tempo lineare con il numero di nodi del tipo, ovvero tempo $O(n)$. L'algoritmo di inclusione effettua n chiamate all'algoritmo di chiusura veloce, il che significa che il costo totale dell'algoritmo è $O(n^2)$.

4.4 Algoritmo strutturale

In questa sezione mostreremo un algoritmo che verifica l'inclusione di due tipi, strutturalmente simili tra loro, eseguendo dei confronti tra la struttura di T e di U senza fare uso di backtraking. Tale algoritmo rappresenta una ottimizzazione agli algoritmi che abbiamo già discusso e richiede un tempo lineare e non quadratico o cubico come i precedenti.

L'algoritmo utilizza la nozione di \subseteq_k , che introduciamo formalmente mediante la seguente definizione.

Definizione 4.25 ($T \subseteq_k U$). Siano T e U due tipi. Diciamo che T è incluso in U , a meno di ϵ , se:

$$T \subseteq_k U \quad =_{def} \quad \llbracket T \rrbracket \setminus \{\epsilon\} \subseteq \llbracket U \rrbracket \setminus \{\epsilon\}$$

Osservazione 4.26. $T \subseteq U \Leftrightarrow (T \subseteq_k U \wedge (N(T) \Rightarrow N(U)))$.

Vediamo adesso una serie di regole che utilizzeremo per semplificare sia il tipo T che il tipo U . Tali regole trasformano il problema dell'inclusione in al più due inclusioni più semplici sul quale è possibile applicare l'algoritmo strutturale ricorsivamente.

Teorema 4.27 (decomposizione). Possiamo decomporre l'inclusione tra i tipi T e U mediante le seguenti regole, dove con i nomi indichiamo gli operatori associati rispettivamente a T e a U :

(divide++)

$$\begin{aligned} S(T_1) \subseteq S(U_1) \wedge S(T_2) \subseteq S(U_2) \\ \Rightarrow T_1 + T_2 \subseteq_k U_1 + U_2 &\Leftrightarrow T_1 \subseteq_k U_1 \wedge T_2 \subseteq_k U_2 \\ T_1 + T_2 \leq U_1 + U_2 &\Leftrightarrow T_1 \subseteq_k U_1 \wedge T_2 \subseteq_k U_2 \\ &\quad \wedge (N(T_1 + T_2) \Rightarrow N(U_1 + U_2)) \end{aligned}$$

(divide&&):

$$\begin{aligned} S(T_1) \subseteq S(U_1) \wedge S(T_2) \subseteq S(U_2) \\ \Rightarrow T_1 \&T_2 \subseteq_k U_1 \&U_2 &\Leftrightarrow T_1 \subseteq_k U_1 \wedge T_2 \subseteq_k U_2 \\ &\quad \wedge (N(T_1) \wedge (S(T_2) \neq \emptyset) \Rightarrow N(U_1)) \\ &\quad \wedge (N(T_2) \wedge (S(T_1) \neq \emptyset) \Rightarrow N(U_2)) \\ T_1 \&T_2 \leq U_1 \&U_2 &\Leftrightarrow T_1 \leq U_1 \wedge T_2 \leq U_2 \end{aligned}$$

(divide $\cdot\cdot$):

$$\begin{aligned} S(T_1) \subseteq S(U_1) \wedge S(T_2) \subseteq S(U_2) \\ \Rightarrow T_1 \cdot T_2 \subseteq_k U_1 \cdot U_2 &\Leftrightarrow T_1 \subseteq_k U_1 \wedge T_2 \subseteq_k U_2 \\ &\quad \wedge (N(T_1) \wedge (S(T_2) \neq \emptyset) \Rightarrow N(U_1)) \\ &\quad \wedge (N(T_2) \wedge (S(T_1) \neq \emptyset) \Rightarrow N(U_2)) \end{aligned}$$

ogni nodo dell'albero una stringa della forma x_1, x_2, \dots, x_k , dove k rappresenta la profondità del nodo e i vari x_i sono numeri. La stringa è ricavata ricorsivamente nel seguente modo:

- al nodo radice assegnamo la stringa 0;
- se il nodo è un figlio sinistro concateniamo alla codifica del padre la stringa .0;
- se il nodo è un figlio destro concateniamo alla codifica del padre la stringa .1.

Esempio 4.28. Consideriamo il tipo $U = (a[2..3] \cdot b[3..3]) + c[2..4]$, la radice ha la stringa 0 per definizione, il nodo $a[2..3] \cdot b[3..3]$ ha la stringa 0.0 mentre $c[2..4]$ ha 0.1, infine $a[2..3]$ e $b[3..3]$ hanno rispettivamente 0.0.0 e 0.0.1.

Il tipo T viene codificato in modo analogo ma iniziando dagli atomi: se $a \in S(T)$, allora all'atomo che contiene il simbolo a in T associamo la stessa codifica dell'atomo che contiene il simbolo a in U . Ai nodi con un operatore associamo come codifica la più lunga parte iniziale comune della codifica dei suoi figli.

Esempio 4.29. Consideriamo il tipo $T = c[2..4] \&a[2..3]$ e il tipo U dell'Esempio 4.28, gli atomi $c[2..4]$ e $a[2..3]$ hanno codifica rispettivamente 0.1 e 0.0.0, mentre il nodo $c[2..4] \&a[2..3]$ ha codifica 0 in quanto è la più lunga parte comune tra i suoi figli.

Proprietà 4.30. Se $C_{T,U}(n) = \alpha$, con n un nodo di T , allora α è il nodo di U più lontano dalla radice tale che $S(n) \subseteq S(\alpha)$.

Corollario 4.31. Se vale $C_{T,U}(n) = \alpha$, allora vale $\alpha' < \alpha \Leftrightarrow S(n) \subseteq S(\alpha')$, dove con $\alpha' < \alpha$ indichiamo α' ha profondità minore di α .

Verifichiamo $S(T) \subseteq S(U)$, dove T e U sono due nodi, semplicemente verificando se la codifica di T inizia con la codifica di U .

Esempio 4.32. Consideriamo il tipo T dell'Esempio 4.29 e il tipo U dell'Esempio 4.28: $S(T) \subseteq S(U)$ in quanto entrambi hanno come codifica 0, mentre $S(T) \not\subseteq S(a[2..3] \cdot b[3..3])$ in quanto la codifica di T è 0 e quindi non inizia con la codifica di $a[2..3] \cdot b[3..3]$ che è 0.0.

L'algoritmo raffigurato in Figura 4.10 calcola \subseteq_k e procede con il verificare la possibilità di applicare una regola *focus*; in questo caso si focalizza su uno dei due figli del tipo U . Se non riesce ad applicare una regola *focus*, verifica la possibilità di applicare una regola *divide*; in caso di successo richiama ricorsivamente l'algoritmo sulle due coppie generate. Infine, l'algoritmo verifica, come per le versioni precedenti, i vincoli piatti con lo stesso meccanismo.

Se l'algoritmo non riesce ad applicare nessuna regola *focus* o *divide*, invoca un algoritmo di inclusione, a meno di ϵ , tra quelli che abbiamo precedentemente presentato, su tutti i sottotermini al quale ci siamo ridotti tramite l'applicazione di queste regole. Tali regole, infatti, ci garantiscono che, applicando un normale algoritmo di inclusione ai sottotermini il risultato complessivo è corretto, quindi non è necessario eseguire nessun tipo di backtraking quando l'algoritmo non riesce ad applicare nessuna regola *focus* o *divide*.

FOCUS(T, U)

```

1  if ( $S(T) \subseteq S(\text{FST}(U))$ )  $\wedge$  ( $(U = +) \vee (U \in \{., \&\} \wedge N(\text{SND}(U)))$ )
2    then return FST( $U$ )
3    else return nil

```

DIVIDE(T, U)

```

1  if ( $(S(T) = \emptyset) \vee (S(T) = \{a\})$ )
2    then return nil
3  if ( $(S(\text{FST}(T)) \subseteq S(\text{FST}(U))) \wedge (S(\text{SND}(T)) \subseteq S(\text{SND}(U)))$ )
4     $\wedge$  ( $(T = +) \wedge (U = +)$ )
5     $\vee$  ( $(T = +) \wedge (U \in \{., \&\}) \wedge N(\text{FST}(U)) \wedge N(\text{SND}(U))$ )
6     $\vee$  ( $((T = \cdot) \wedge (U \in \{., \&\})) \vee ((T = \&) \wedge (U = \&))$ )
7       $\wedge$  ( $N(\text{FST}(T)) \wedge (S(\text{SND}(T)) \neq \emptyset) \Rightarrow N(\text{FST}(U))$ )
8       $\wedge$  ( $N(\text{SND}(T)) \wedge (S(\text{FST}(T)) \neq \emptyset) \Rightarrow N(\text{SND}(U))$ ))
9    then return ( $(\text{FST}(T), \text{FST}(U)), (\text{SND}(T), \text{SND}(U))$ )
10   else return nil

```

Figura 4.9: Procedure di supporto per l'algoritmo in Figura 4.10.

```

DECSUB( $T, U$ )
1  if ( $(S(U) = \{a\})$ )
2    then return true
3  switch (FOCUS( $T, U$ ))
4    case  $U_1$  : return DECSUB( $T, U_1$ );
5    case nil :
6      switch (DIVIDE)
7        case ( $(T_1, U_1), (T_2, U_2)$ ) :
8          return (DECSUB( $T_1, U_1$ )  $\wedge$  DECSUB( $T_2, U_2$ ));
9        case nil : return SUB( $T, U$ );

```

Figura 4.10: Algoritmo per l'inclusione di tipi simili.

Esempio 4.33. Consideriamo il tipo T e U dell'Esempio 4.16. L'algoritmo strutturale costruisce le codifiche dei tipi T e U che risultano:

n	$dewey(U)$	n	$dewey(T)$
1	0	1	0.0
2	0.0	2	0.0.0
3	0.0.0	3	0.0.1
4	0.0.1		
5	0.1		
6	0.1.0		
7	0.1.0.0		
8	0.1.0.1		
9	0.1.1		

Dopo aver costruito la codifica, abbiamo $U = U_1 + U_2$ e $S(T) \subseteq S(U)$, poiché 0.0, che rappresenta la codifica di T , inizia con 0, che rappresenta la codifica di U , quindi l'algoritmo applica la regola *focus* sul sottoalbero U_1 che corrisponde al nodo 2 di U .

Alla seconda chiamata abbiamo $T = a [1..1] \cdot b [2..2]$ e $U = a [1..2] \&b [2..2]$, inoltre abbiamo $S(a [1..1]) \subseteq S(a [1..2])$ e $S(b [2..2]) \subseteq S(b [2..2])$, quindi l'algoritmo

applica la regola *divide* chiamando prima $\text{DECSub}(a[1..1], a[1..2])$, che restituisce il valore **true** e successivamente $\text{DECSub}(b[2..2], b[2..2])$, che restituisce **true**. Quindi, il risultato dell'algoritmo è **true**.

Verificare se $S(T) \subseteq S(U)$ in questo modo richiede un tempo pari alla profondità degli alberi T e U . Applicare ricorsivamente le regole *focus* e *divide* richiede un numero lineare di ricorsione, quindi il costo complessivo dell'algoritmo è $O(n * \text{depth})$ che rappresenta un miglioramento rispetto all'algoritmo quadratico, nel caso quest'ultimo non sia mai richiamato. Nel capitolo dove parleremo della sperimentazione di questi tre algoritmi vedremo che questa ottimizzazione si comporta bene anche quando viene richiamato l'algoritmo quadratico.

Capitolo 5

Validazione

In questo capitolo parleremo degli algoritmi di validazione. Inizieremo vedendo una prima versione per stringhe e espressioni regolari, che successivamente ottimizzeremo per ottenere un algoritmo lineare. Successivamente, vedremo alcuni algoritmi che lavorano su dati XML veri e propri e che al loro interno utilizzano gli algoritmi per le stringhe.

5.1 Algoritmo base

Il primo algoritmo decide l'appartenenza di una parola w a un tipo T in tempo $O(|W| * depth(T))$, dove con $depth(T)$ indichiamo:

$$\begin{aligned} depth(\epsilon) &=_{def} 0 \\ depth(a [m..n]) &=_{def} 1 \\ depth(T_1 + T_2) &=_{def} 1 + depth(T_1) + depth(T_2) \\ depth(T_1 \cdot T_2) &=_{def} 1 + depth(T_1) + depth(T_2) \\ depth(T_1 \& T_2) &=_{def} 1 + depth(T_1) + depth(T_2) \end{aligned}$$

L'algoritmo esegue una scansione lineare della parola w verificando che tutti i vincoli del tipo T siano soddisfatti. Il ragionamento base dell'algoritmo prevede che, a ogni simbolo a letto dalla parola $a \cdot w_1$, l'insieme dei vincoli F venga trasformato nel suo *residuo* F' in accordo alla Tabella 5.1. L'insieme F' rappresenta l'insieme dei vincoli che la parola w_1 deve soddisfare. Indicheremo con $F \xrightarrow{a} F'$ la trasformazione dell'insieme di vincoli F nell'insieme F' leggendo il simbolo a .

Vincoli di co-occorrenza					
Condizioni	$a \in A$	$a \in B$	$a \in A$	$a \in B$	$a \in A$
Vincoli	$A^+ \Rightarrow B^+$	$A^+ \Rightarrow B^+$	$A^+ \Leftrightarrow B^+$	$A^+ \Leftrightarrow B^+$	A^+
Residuo	B^+	true	B^+	A^+	true
Vincoli di ordine					
Condizioni	$a \in A$	$a \in B$	$a \in A$	$a \in B$	$a \in A$
Vincoli	$A \prec B$	$A \prec B$	$A \prec \succ B$	$A \prec \succ B$	A^-
Residuo	$A \prec B$	A^-	B^-	A^-	false

Figura 5.1: Tabella di trasformazione dei vincoli interni in residuo.

La residuazione viene applicata soltanto ai vincoli interni, in quanto i vincoli piatti possono essere controllati contando il numero di occorrenze di ogni simbolo a , memorizzandoli in $Count[a]$, e successivamente verificandoli in tempo lineare tramite la procedura **CARDINALITYOK** che restituisce un valore di accettazione se $\forall a [m..n] \in Atoms(T)$ vale $m \leq Count[a] \leq n$.

Definizione 5.1 (\rightarrow^+). Siano F, F', F'' insiemi di vincoli interni, a un simbolo e w una parola. Definiamo la funzione \rightarrow^+ come:

$$\begin{aligned}
 F &\xrightarrow{a} F' && \Rightarrow && F \xrightarrow{a^+} F' \\
 w \neq \epsilon : F &\xrightarrow{a} F' \wedge F' \xrightarrow{w^+} F'' && \Rightarrow && F \xrightarrow{a^+ w^+} F''
 \end{aligned}$$

Definizione 5.2 (\rightarrow^ϵ). Siano A e B insiemi di simboli. Definiamo la funzione \rightarrow^ϵ come:

$$\begin{array}{llll}
 A^+ \Rightarrow B^+ & \rightarrow^\epsilon \mathbf{true} & A^+ \Leftrightarrow B^+ & \rightarrow^\epsilon \mathbf{true} \\
 A^+ & \rightarrow^\epsilon \mathbf{false} & \mathbf{false} & \rightarrow^\epsilon \mathbf{false} \\
 A \prec B & \rightarrow^\epsilon \mathbf{true} & A \prec \succ B & \rightarrow^\epsilon \mathbf{true} \\
 A^- & \rightarrow^\epsilon \mathbf{true} & \mathbf{true} & \rightarrow^\epsilon \mathbf{true}
 \end{array}$$

Definizione 5.3 (\rightarrow^*). Siano F, F' insiemi di vincoli interni, $b \in \{\mathbf{true}, \mathbf{false}\}$ e w una parola. Definiamo la funzione \rightarrow^* come:

$$\begin{aligned}
 F &\rightarrow^\epsilon b && \Rightarrow && F \xrightarrow{\epsilon^*} b \\
 F &\xrightarrow{w^+} F' \wedge F' \rightarrow^\epsilon b && \Rightarrow && F \xrightarrow{w^*} b
 \end{aligned}$$

Teorema 5.4 (Residuazione). Data un insieme di vincoli interni F e una parola w , vale:

$$w \models F \Leftrightarrow F \xrightarrow{w}^* \mathbf{true}$$

Il Teorema 5.4 ci dice che, per verificare tutti i vincoli interni di un tipo T in una parola w , è sufficiente calcolare e aggiornare l'insieme di vincoli tramite il residuo per tutti i simboli di w e, infine, verificare se l'insieme di vincoli finale ammette la stringa vuota, ovvero se non sono presenti vincoli della forma A^+ oppure **false**.

Il costo dell'algoritmo basato sul Teorema 5.4 è $O(|w| * |\mathcal{NC}|)$, in quanto per ogni simbolo applichiamo una trasformazione su tutti i vincoli, e per calcolare l'appartenenza è necessario leggere tutti i simboli. Questo costo è comunque troppo elevato: infatti l'insieme di simboli è spesso più grande del tipo stesso che, a sua volta, può essere molto più grande della parola sul quale vogliamo decidere l'appartenenza (si consideri, ad esempio, un tipo molto grande composto in maggior parte da operatori +).

Per ovviare al problema dell'alto costo computazionale, l'algoritmo che presentiamo nel dettaglio non estrae direttamente tutti i vincoli dal tipo T , ma li codifica nel seguente insieme di strutture dati:

- una tabella hash $NodeOfSymbol[]$, indicizzata sui simboli, dove, per ogni simbolo a , $NodeOfSymbol[a]$ rappresenta il nodo n dell'atomo $a [m..n]$ oppure null se il simbolo non esiste;
- un array $Parent[]$ dove, per ogni nodo d , $Parent[d]$ è null, se d è la radice, oppure una coppia $(d_p, direction)$, dove d_p è il nodo del padre e $direction$ la direzione di d ovvero se d è figlio destro o sinistro di d_p ;
- un array $CC[]$ dove, per ogni nodo d , con A_1 e A_2 rispettivamente i simboli a destra e sinistra di d , $CC[d]$ è un simbolo di $\{\Leftrightarrow, \Rightarrow, \Leftarrow, L^+, R^+, \mathbf{true}\}$ con il seguente significato:
 - \Leftrightarrow codifica il vincolo $A_1^+ \Leftrightarrow A_2^+$;
 - \Rightarrow codifica il vincolo $A_1^+ \Rightarrow A_2^+$;
 - \Leftarrow codifica il vincolo $A_1^+ \Leftarrow A_2^+$;

- L^+ codifica il vincolo A_1^+ ;
- R^+ codifica il vincolo A_2^+ ;
- **true** codifica il vincolo **true**.

T	$N(T_1)$	$N(T_2)$	$CC[n]$	$OC[n]$
$T_1 \cdot T_2$	true	true	true	\prec
$T_1 \cdot T_2$	true	false	\Rightarrow	\prec
$T_1 \cdot T_2$	false	true	\Leftarrow	\prec
$T_1 \cdot T_2$	false	false	\Leftrightarrow	\prec
$T_1 \& T_2$	true	true	true	true
$T_1 \& T_2$	true	false	\Rightarrow	true
$T_1 \& T_2$	false	true	\Leftarrow	true
$T_1 \& T_2$	false	false	\Leftrightarrow	true
$T_1 + T_2$	-	-	true	$\prec \succ$

Figura 5.2: Inizializzazione delle strutture per codificare i vincoli interni.

- un array $OC[]$ dove, per ogni nodo d , con A_1 e A_2 rispettivamente i simboli a destra e sinistra di d , $OC[d]$ è un simbolo di $\{\prec, \prec \succ, L^+, R^+, \mathbf{true}\}$ e ha il seguente significato:
 - \prec codifica il vincolo $A_1 \prec A_2$;
 - $\prec \succ$ codifica il vincolo $A_1 \prec \succ A_2$;
 - L^- codifica il vincolo A_1^- ;
 - R^- codifica il vincolo A_2^- ;
 - **true** codifica il vincolo **true**.
- due array $Min[]$ e $Max[]$ dove, per ogni nodo d , se d è un atomo, $Min[d]$ e $Max[d]$ codificano il vincolo $a?[Min[a]..Max[a]]$, altrimenti contengono null;
- una variabile *Nullable* che contiene il valore del predicato $N(T)$.

```

ANCESTORS( $d$ )
1  if  $Parent[d]$  is null
2    then return (emptylist);
3    else return ( $Parent[d] + +ANCESTORS(Parent[d])$ );

CHECKCLOSE()
1  if exists  $d$  in  $ToCheck$  with  $CC[d] \neq \mathbf{true}$ 
2    then return false;
3  if  $\neg$ CARDINALITYOK( $Count[]$ ,  $Min[]$ ,  $Max[]$ )
4    then return false;
5  return true;

```

Figura 5.3: Procedure di supporto all'algoritmo della Figura 5.5.

In Tabella 5.2 troviamo con che simbolo inizializzare gli array $CC[d]$ e $OC[d]$ per un nodo d del tipo T .

L'algoritmo utilizza due ulteriori strutture locali:

- una tabella hash $Count[]$ indirizzata sui simboli, dove per ogni simbolo a di una parola w , $Count[a]$ contiene il numero di occorrenze del simbolo a in w ;
- una lista $ToCheck$ contenente i nodi n tali che, al termine dell'algoritmo, deve valere $\forall n \in ToCheck, CC[n] = \mathbf{true}$.

L'algoritmo procede costruendo la codifica delle strutture appena descritte, inizializzando $Count[]$ a zero e $ToCheck$ con la lista vuota, successivamente per ogni carattere a della parola w , calcola il residuo per tutti gli antenati del nodo $a[m..n]$ in T , tenendo traccia dei vincoli A^+ generati inserendoli nella lista $ToCheck$, e incrementa di uno il contatore $Count[a]$ corrispondente al simbolo letto. Quando la parola w è terminata, l'algoritmo verifica che tutti i vincoli A^+ siano stati trasformati in **true** e che il numero di occorrenze dei simboli sia corretto rispetto ai vincoli di cardinalità (Figura 5.5).

Esempio 5.5. Consideriamo il seguente tipo (Figura 5.4):

$$T = (a[1..*] + \epsilon) \& (b[1..2] + (c[2..3] + d[1..1]))$$

Applichiamo l'algoritmo di validazione alla parola *abba*.

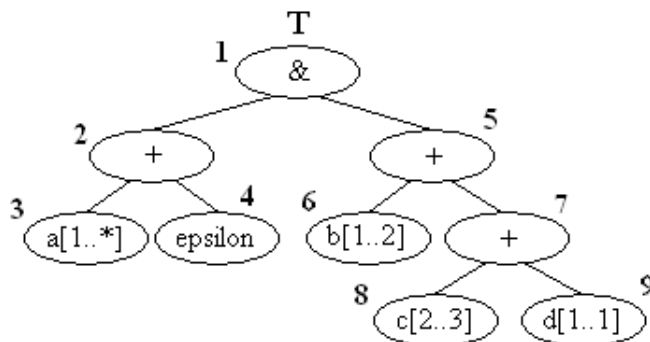


Figura 5.4: Rappresentazione grafica del tipo dell'Esempio 5.5.

Le strutture dati iniziali risultano le seguenti:

n	$Parent[]$	$CC[]$	$OC[]$	$Min[]$	$Max[]$	$NodeOfSymbol[]$	$Nullable$
1	<i>null</i>	\Rightarrow	true			a 3	<i>false</i>
2	(1, <i>left</i>)	true	$\langle \rangle$			b 6	
3	(2, <i>left</i>)	true	true	1	*	c 8	
4	(2, <i>right</i>)	true	true			d 9	
5	(1, <i>right</i>)	true	$\langle \rangle$				
6	(5, <i>left</i>)	true	true	1	2		
7	(5, <i>right</i>)	true	$\langle \rangle$				
8	(7, <i>left</i>)	true	true	2	3		
9	(7, <i>right</i>)	true	true	1	1		

Alla prima iterazione, l'algoritmo legge il simbolo a , incrementa $Count[a]$, che adesso vale 1, e analizza tutti i nodi antenati del nodo 3, ovvero l'insieme di nodi $\{3, 2, 1\}$. Tra i vincoli di co-occorrenza soltanto quello del nodo 1 si trasforma in R^+ , mentre tra i vincoli di ordine solo il nodo 2 diventa R^- . Inoltre, il nodo 1 viene aggiunto alla lista *ToCheck*.

Alla seconda iterazione, l'algoritmo legge il simbolo b , incrementa $Count[b]$, che adesso vale 1, e analizza tutti i nodi antenati del nodo 6, ovvero l'insieme di

```

MEMBER( $w, T$ )
1  if (ISEMPTY( $w$ )  $\wedge$   $\neg$ Nullable)
2    then return false;
3  for  $a$  in  $w$ 
4  do if (NodeOfSymbol[ $a$ ] is null)
5    then return false;
6    Count[ $a$ ] := Count[ $a$ ] + 1;
7    for ( $d_p, direction$ ) in ANCESTORS(NodeOfSymbol[ $a$ ])
8    do switch (CC[ $d_p$ ], direction)
9      case ( $\Rightarrow$  or  $\Leftrightarrow$ , left) : CC[ $d_p$ ] :=  $R^+$ ;
10     push( $d_p$ , ToCheck);
11     case ( $\Leftarrow$  or  $\Leftrightarrow$ , right) : CC[ $d_p$ ] :=  $L^+$ ;
12     push( $d_p$ , ToCheck);
13     case ( $\Leftarrow$  or  $L^+$ , left) or ( $\Rightarrow$  or  $R^+$ , right) : CC[ $d_p$ ] := true;
14   switch (OC[ $d_p$ ], direction)
15     case ( $\prec$  or  $\succ$ , right) : OC[ $d_p$ ] :=  $L^-$ ;
16     case ( $\prec$  or  $\succ$ , left) : OC[ $d_p$ ] :=  $R^-$ ;
17     case ( $L^-$ , left) or ( $R^-$ , right) : return false;
18 return CHECKCLOSE();

```

Figura 5.5: Algoritmo per l'appartenenza con costo $O(|T| + |w| * depth(T))$.

nodi $\{6, 5, 1\}$. Tra i vincoli di co-occorrenza soltanto quello del nodo 1, che valeva R^+ , si trasforma in **true**, mentre tra i vincoli di ordine solo il nodo 5 diventa R^- .

Alla terza iterazione, l'algoritmo legge il simbolo b , incrementa $Count[b]$, che adesso vale 2, e analizza tutti i nodi antenati del nodo 6. I vincoli non subiscono nessuna modifica.

Alla quarta iterazione, l'algoritmo legge il simbolo a , incrementa $Count[a]$, che adesso vale 2, e analizza tutti i nodi antenati del nodo 3. I vincoli non subiscono nessuna modifica.

L'algoritmo ha letto tutta la stringa, essendo $Min[a] \leq Count[a] \leq Max[a]$

e $Min[b] \leq Count[b] \leq Max[b]$, i vincoli piatti sono soddisfatti. Infine, essendo presente il nodo 1 nella lista *ToCheck*, controlla che $CC[1] = \mathbf{true}$, che è vero, quindi la stringa viene accettata dall'algoritmo.

Analizzando il costo computazionale di questo algoritmo, osserviamo che la codifica può essere eseguita in tempo $O(|T|)$, dove $|T|$ rappresenta il numero di nodi del tipo T . Successivamente l'algoritmo analizza tutti gli antenati del nodo $a[m..n]$, che corrispondono alla profondità del tipo T , per ogni simbolo a letto. Infine verifica che ogni simbolo abbia la corretta cardinalità in tempo lineare. Il costo complessivo dell'algoritmo risulta quindi $O(|T| + |w| * depth(T))$.

Infine vale il seguente teorema, dimostrato in [GCS08].

Teorema 5.6 (Correttezza e completezza). Sia w una parola e T un tipo. Vale la seguente proprietà:

$$Member(w, T) = true \Leftrightarrow w \in \llbracket T \rrbracket$$

5.2 Algoritmo con vincoli appiattiti

Gli algoritmi che abbiamo esposto fino ad ora utilizzavano tutti operatori binari per rappresentare il tipo T ; questo comporta un notevole aumento della profondità dell'albero che rappresenta il tipo, con un conseguente aumento del costo dell'algoritmo di validazione. In questa sezione presentiamo un miglioramento all'algoritmo trasformando, in fase di codifica, gli operatori da binari a n-ari.

Definizione 5.7. Sia T un tipo con operatori n-ari, dove T_1, \dots, T_n sono i suoi n figli, $\otimes \in \{., \&\}$ e $SIf(T_1, \dots, T_n) = \{S^+(T_i) \mid \neg N(T_i)\}$. I vincoli interni del tipo T sono definiti come:

$$\begin{aligned} CC(\epsilon) = CC(a[m..n]) &= \mathbf{true} \\ CC(T_1 + \dots + T_n) &= \mathbf{true} \\ CC(T_1 \otimes \dots \otimes T_n) &= (\cup_{i \in 1..n} S(T_i))^+ \Leftrightarrow SIf(T_1, \dots, T_n) \\ &\quad \wedge_{1 \leq i \leq n} CC(T_i) \end{aligned}$$

$$\begin{aligned}
\mathcal{OC}(\epsilon) = \mathcal{OC}(a[m..n]) &= \mathbf{true} \\
\mathcal{OC}(T_1 \& \dots \& T_n) &= \mathbf{true} \\
\mathcal{OC}(T_1 \cdot \dots \cdot T_n) &= S(T_1) \prec \dots \prec S(T_n) \bigwedge_{1 \leq i \leq n} \mathcal{OC}(T_i) \\
\mathcal{OC}(T_1 + \dots + T_n) &= \prec \succ (S(T_1), \dots, S(T_n)) \bigwedge_{1 \leq i \leq n} \mathcal{OC}(T_i)
\end{aligned}$$

Definizione 5.8. Siano $(A_1 \cup \dots \cup A_n) \subseteq A_0$ insiemi di simboli, $i \geq 0$ e $n \geq 0$. La sintassi del linguaggio dei vincoli estesa agli operatori n-ari è data da:

$$\begin{aligned}
F ::= & A_0^+ \Rightarrow \{A_1^+, \dots, A_n^+\} \\
& | A_1^+, \dots, A_n^+ \\
& | \prec \succ (A_1^+, \dots, A_n^+) \\
& | A_1^-, \dots, A_i^-, A_{i+1}^+, \dots, A_n^+
\end{aligned}$$

Nella Definizione 5.8 i vincoli A_1^-, \dots, A_i^- e A_1^+, \dots, A_n^+ sono due casi speciali del vincolo $A_1^-, \dots, A_i^-, A_{i+1}^+, \dots, A_n^+$, dove $i = n$ e $i = 0$ rispettivamente. Inoltre, quanto vale $n = 0$, il vincolo è abbreviato con **true**, mentre con **false** abbreviamo il vincolo \emptyset^+ .

La semantica di questi vincoli è espressa in Tabella 5.6 tramite gli operatori per calcolare il residuo. Inoltre, se in un insieme di vincoli F è presente un vincolo della forma A_1^+, \dots, A_n^+ , vale $F \rightarrow^\epsilon \mathbf{false}$ altrimenti vale $F \rightarrow^\epsilon \mathbf{true}$.

Condizioni	Vincolo	Residuo dopo a
$a \in A_i \subseteq A$	$A^+ \Rightarrow \{A_1^+, \dots, A_n^+\}$	$A_1^-, \dots, A_i^-, A_{i+1}^+, \dots, A_n^+$
$a \in A$	A^+	true
$a \in A_i, n > 1$	A_1^+, \dots, A_n^+	$A_1^+, \dots, A_{i-1}^+, A_{i+1}^+, \dots, A_n^+$
$a \in A_i$	$\prec \succ (A_1, \dots, A_n)$	$A_1^-, \dots, A_{i-1}^-, A_{i+1}^-, \dots, A_n^-$
$a \in A_i, i \leq j$	$A_1^-, \dots, A_j^-, A_{j+1}^+, \dots, A_n^+$	false
$a \in A_i, i > j$	$A_1^-, \dots, A_j^-, A_{j+1}^+, \dots, A_n^+$	$A_1^-, \dots, A_i^-, A_{i+1}^+, \dots, A_n^+$

Figura 5.6: Calcolo del residuo per operatori n-ari.

Analogamente all'algoritmo base, i vincoli vengono codificati in opportune strutture dati senza bisogno di estrarli direttamente. Le strutture dati vengono codificate in modo diverso dall'algoritmo base, e sono:

- un array $Parent[]$, dove, per ogni nodo d , $Parent[d]$ è null, se d è la radice, oppure una coppia $(d_p, position)$, dove d_p è il nodo del padre e $position$ la posizione tra i figli di d ovvero d è il figlio in posizione $position$ a partire da sinistra di d_p ;
- un array $CC[]$ composto da record aventi i campi $kind$, $neededCount$ e $needed[]$, dove, per ogni nodo d , con A_1 e A_n indichiamo i simboli del rispettivo n-esimo figlio di d , con $CC[d].neededCount$ indichiamo il numero di figli T_i dove non vale il predicato $N(T_i)$ e, per ogni figlio di d , $CC[d].needed[i] = \neg N(T_i)$. Il significato di questo array per il nodo d dipende dal valore contenuto in $CC[d].kind$:

- se $CC[d].kind = \Rightarrow$, $CC[d]$ rappresenta il vincolo:

$$(\cup_{i \in 1..n} S(T_i))^+ \Rightarrow \{S^+(T_{k(1)}), \dots, S^+(T_{k(j)})\}$$

dove j è $CC[d].neededCount$ e $k(1), \dots, k(j)$ sono gli indici i , dove $CC[d].needed[i] = true$;

- se $CC[d].kind = A^+$, $CC[d]$ rappresenta il vincolo:

$$\{S^+(T_{k(1)}), \dots, S^+(T_{k(j)})\}$$

dove j è $CC[d].neededCount$ e $k(1), \dots, k(j)$ sono gli indici i , dove $CC[d].needed[i] = true$.

- un array $OC[]$ composto da record aventi i campi $kind$ e $allowed$, dove per ogni nodo d , con A_1 e A_n indichiamo i simboli del rispettivo n-esimo figlio di d . Il significato di questo array per il nodo d dipende dal valore contenuto in $OC[d].kind$:

- se $OC[d].kind = \prec$, $OC[d]$ rappresenta il vincolo:

$$A_1^-, \dots, A_{i-1}^-, A_i^+, \dots, A_n^+$$

dove $OC[n].allowed = i$;

- se $OC[d].kind = \prec \succ$, $OC[d]$ rappresenta il vincolo:

$$\prec \succ (A_1, \dots, A_n)$$

dove $OC[n].allowed$ ha null in quanto valore non significativo;

– se $OC[d].kind = A^-$, $OC[d]$ rappresenta il vincolo:

$$A_1^-, \dots, A_{i-1}^-, A_{i+1}^-, \dots, A_n^-$$

dove $OC[n].allowed = i$;

```

MEMBERFLAT( $w, T$ )
1  for  $a$  in  $w$ 
2  do if ( $NodeOfSymbol[]$  is null)
3      then return false;
4       $Count[a] := Count[a] + 1$ ;
5      for ( $d_p, position$ ) in ANCESTORS( $NodeOfSymbol[a]$ )
6      do switch  $CC[d_p].kind$ 
7          case ( $\Rightarrow$ ) :  $CC[d_p] := A^+$ ;
8                       $push(d_p, ToCheck)$ ;
9                      RESIDUATEPLUS( $CC[d_p], position$ );
10         case ( $A^+$ ) : RESIDUATEPLUS( $CC[d_p], position$ );
11     switch  $OC[d_p].kind$ 
12         case ( $\prec$ ) : if ( $OC[d_p].allowed \leq position$ )
13             then  $OC[d_p].allowed := position$ ;
14             else return false;
15         case ( $\prec \succ$ ) :  $OC[d_p].kind := A^-$ ;
16                      $OC[d_p].allowed := position$ ;
17         case ( $A^-$ ) : if ( $OC[d_p].allowed \neq position$ )
18             then return false;
19 return CHECKCLOSE();

```

Figura 5.7: Algoritmo per l'appartenenza con costo $O(|T| + |w| * flatdepth(T))$.

L'algoritmo (Figura 5.7) procede leggendo i simboli della parola w , come per la versione base, e analizzando gli antenati dell'atomo $a[m..n]$ corrispondente.

Per ogni antenato, quando incontra un nodo i figlio di un nodo d per la prima volta, aggiorna $CC[d].kind$, ovvero se vale $CC[d].kind = \Rightarrow$ l'algoritmo lo trasforma in $CC[d].kind = A^+$, e imposta $CC[d].needed[i] = false$ decrementando

$CC[d].neededCount$ di uno. Un vincolo $CC[d].kind = A^+$ si trasforma in **true** quando vale $CC[d].neededCount = 0$.

Analogamente ai vincoli di co-occorrenza, se l'algoritmo incontra un nodo d con $OC[d].kind = \prec$, verifica se la posizione $position$ di d rispetto a suo padre è nei vincoli della forma A_1^-, \dots, A_i^- , e in quel caso l'algoritmo termina restituendo false, oppure vincoli della forma $A_{i+1}^- \prec A_n$, e in quel caso aggiorna $OC[d].allowed = position$. Se $OC[d].kind = \prec \succ$, l'algoritmo calcola il residuo del vincolo, ovvero imposta $OC[d].kind = A^-$ e $OC[d].allowed = position$. Infine, se vale $OC[d].kind = A^-$ l'algoritmo controlla che la posizione del nodo d rispetto a suo padre sia quella accettata, ovvero se vale $OC[d].allowed = position$, in caso contrario termina restituendo false. Quando la parola w è terminata, l'algoritmo esegue i controlli sui vincoli di co-occorrenza e di cardinalità visti per l'algoritmo precedente.

Esempio 5.9. Consideriamo il tipo T dell'Esempio 5.5, la cui versione appiattita è rappresentata in Figura 5.8. Applichiamo l'algoritmo di validazione alla parola *abba*.

Le strutture dati iniziali risultano le seguenti:

n	$Parent[]$	$CC[]$			$OC[]$	
		$kind$	$neededCount$	$needed[]$	$kind$	$allowed$
1	<i>null</i>	\Rightarrow	1	{false, true}	true	
2	(1, 1)	true			$\prec \succ$	<i>null</i>
3	(2, 1)	true			true	
4	(2, 2)	true			true	
5	(1, 2)	true			$\prec \succ$	<i>null</i>
6	(5, 1)	true			true	
7	(5, 2)	true			true	
8	(5, 3)	true			true	

Alla prima iterazione, l'algoritmo legge il simbolo a , incrementa $Count[a]$, che adesso vale 1, e analizza tutti i nodi antenati del nodo 3, ovvero l'insieme di nodi $\{2, 1\}$. Tra i vincoli di co-occorrenza $CC[1].kind$ si trasforma in A^+ e il nodo 1 viene aggiunto alla lista *ToCheck*. Tra i vincoli di ordine $OC[2].kind$ diventa A^- e $OC[2].allowed$ viene assegnato con 1.

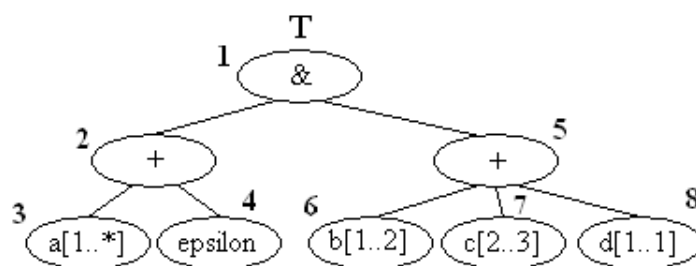


Figura 5.8: Rappresentazione grafica del tipo dell'Esempio 5.9.

```

RESIDUATEPLUS(ccd, childPos)
1  if (ccd.needed[childPos])
2    then ccd.needed[childPos] := false;
3       ccd.neededCount := ccd.neededCount - 1;
4       if (ccd.neededCount = 0)
5         then ccd.kind := true;

```

Figura 5.9: Procedure di supporto all'algoritmo della Figura 5.7.

Alla seconda iterazione, l'algoritmo legge il simbolo b , incrementa $Count[b]$, che adesso vale 1, e analizza tutti i nodi antenati del nodo 6, ovvero l'insieme di nodi $\{5, 1\}$. Tra i vincoli di co-occorrenza $CC[1].needed[2]$ diventa *false* e $CC[1].neededCount$ viene decrementato di uno. Essendo $CC[1].neededCount = 0$, l'algoritmo trasforma $CC[1].kind$ nel vincolo **true**. Tra i vincoli di ordine $OC[5].kind$ diventa A^- e $OC[5].allowed$ viene assegnato con 1.

Alla terza iterazione, l'algoritmo legge il simbolo b , incrementa $Count[b]$, che adesso vale 2, e analizza tutti i nodi antenati del nodo 6. I vincoli non subiscono nessuna modifica.

Alla quarta iterazione, l'algoritmo legge il simbolo a , incrementa $Count[a]$, che adesso vale 2, e analizza tutti i nodi antenati del nodo 3. I vincoli non subiscono nessuna modifica.

L'algoritmo ha letto tutta la stringa, essendo $Min[a] \leq Count[a] \leq Max[a]$ e $Min[b] \leq Count[b] \leq Max[b]$, i vincoli piatti sono soddisfatti. Infine, essendo

presente il nodo 1 nella lista *ToCheck*, controlla che $CC[1] = \mathbf{true}$, che è vero, quindi la stringa viene accettata dall'algoritmo.

Il costo computazionale di questo algoritmo è dato da $O(|T| + |w| * flatdepth(T))$ dove $flatdepth(T)$ è la profondità del tipo dopo che tutti gli operatori sono stati appiattiti.

Il seguente teorema, dimostrato in [GCS08] enuncia la correttezza dell'algoritmo appena illustrato.

Teorema 5.10 (Correttezza e completezza). Sia w una parola e T un tipo. Vale la seguente proprietà:

$$MemberFlat(w, T) = true \Leftrightarrow w \in \llbracket T \rrbracket$$

5.3 Algoritmo lineare

In questa sezione presenteremo una seconda ottimizzazione all'algoritmo di validazione che lo rende lineare.

L'algoritmo base visita tutti gli antenati dell'atomo $a[m..n]$ ogni volta che incontra un simbolo a nella parola w : questo è ridondante. Consideriamo un nodo d dove A_1 e A_2 sono rispettivamente i figli del nodo sinistro e del nodo destro; quando il vincolo di d è stato residuo perché abbiamo letto un simbolo a di A_1 non ci sono motivi per visitare nuovamente il nodo d . L'unica eccezione a questa regola è data dal vincolo $A_1 \prec A_2$ dove, dopo aver letto un simbolo di A_1 , la lettura di un simbolo di A_2 introduce il vincolo A_1^- e, quindi, ulteriori simboli di A_1 non possono essere ignorati in quanto abbiamo:

$$a \in A_1, b \in A_2, c \in A_1 : (A_1 \prec A_2) \xrightarrow{ab^+} A_1^- \xrightarrow{c} \mathbf{false}$$

L'algoritmo lineare si basa sulle seguenti osservazioni:

- quando viene attraversato un nodo d , $Parent[d]$, che contiene la coppia $(d_p, position)$, viene impostato a null e in un $Deleted[d_p, position]$ viene memorizzato d ; in questo modo il nodo d non verrà restituito dalla funzione ANCESTORS;

```

MEMBERFLATLIN( $w, T$ )
1  for  $a$  in  $w$ 
2  do ...
3    for  $(d, (d_p, position))$  in UNVISITEDANCESTORS( $NodeOfSymbol[a]$ )
4    do  $Parent[d] := null$ ;
5        $Deleted[d_p, position] := d$ ;
6    ...
7    switch  $OC[d_p].kind$ 
8      case  $(\prec)$  : if  $(OC[d_p].allowed < position)$ 
9                then REACTIVATE( $d_p, OC[d_p].allowed$ )
10                $OC[d_p].allowed := position$ ;
11      else if  $(OC[d_p].allowed > position)$ 
12            then return false;
13      case  $(\prec\succ)$  :  $OC[d_p].kind := A^-$ ;
14                 $OC[d_p].allowed := position$ ;
15      case  $(A^-)$  : if  $(OC[d_p].allowed \neq position)$ 
16            then return false;

```

Figura 5.10: Algoritmo per l'appartenenza con costo $O(|T| + |w|)$.

- nel caso in cui raggiungiamo un nodo d tale che $Parent[d] = (d_p, position)$, $OC[d_p].kind = \prec$ e $OC[d_p].allowed < position$ significa che, per calcolare il residuo, $OC[d_p].allowed$ viene incrementato introducendo un nuovo vincolo A^- ; dobbiamo, quindi, ripristinare il sottoalbero radicato nel figlio $OC[d_p].allowed$ di d_p utilizzando l'array $Deleted[]$. I sottoalberi precedenti a $OC[d_p].allowed$ non necessitano di essere riattivati, in quanto sono già stati riattivati precedentemente.

Nessun nodo può essere eliminato e ripristinato per due volte. Supponiamo di avere un nodo d , i -esimo figlio di d_p , che è stato eliminato e ripristinato una volta: significa che d_p ha un vincolo della forma $A_1^-, \dots, A_{j-1}^-, A_j^+, \dots, A_n^+$, dove $i < j$, in quanto è già stato eliminato e, quindi, ha già prodotto A_i^- . Se d

```

REACTIVATE( $d_p, position$ )
1  if Deleted[ $d_p, position$ ] is not null
2    then  $d := Deleted[d_p, position]$ ;
3        Deleted[ $d_p, position$ ] := null;
4        Parent[ $d$ ] := ( $d_p, position$ );
5        for  $pos$  in 1..KidNum[ $d$ ]
6        do REACTIVATE( $d, pos$ );

```

Figura 5.11: Procedure di supporto all'algoritmo della Figura 5.10.

viene eliminato una seconda volta, l'algoritmo restituisce il valore false in quanto è stato letto un simbolo $a \in A_i$.

L'algoritmo ha costo iniziale $O(|T|)$ per la fase di codifica e costo $O(|w|)$ per leggere ogni simbolo della parola w . Il costo complessivo risulta quindi $O(|T|+|w|)$.

Il seguente teorema, dimostrato in [GCS08] enuncia la correttezza dell'algoritmo appena illustrato.

Teorema 5.11 (Correttezza e completezza). Sia w una parola e T un tipo. Vale la seguente proprietà:

$$MemberFlatLin(w, T) = true \Leftrightarrow w \in \llbracket T \rrbracket$$

5.4 Validare sequenze di stringhe

Prima di passare dalle stringhe agli alberi XML, analizziamo il problema di verificare una sequenza di parole, ovvero il caso dove il tipo T viene utilizzato per validare le parole w_1, \dots, w_m . Applicare ripetutamente l'algoritmo MEMBERFLATLIN senza nessuna modifica ha costo $O(m * (|T| + |w|))$, dove con $|w|$ indichiamo la lunghezza media delle parole.

Il costo $O(m * (|T| + |w|))$, in generale, non è lineare con la dimensione dei dati di ingresso, che è $|T| + m * |w|$. Se abbiamo $|T| < |w|$, possiamo limitare superiormente ($m * (|T| + |w|)$ con $O(2m * |w|)$, quindi in questo caso è lineare.

In generale, $|T|$ può essere molto più grande di $|w|$ quindi la soluzione migliore consiste nell'evitare di ricostruire la codifica di T ogni volta.

Per ricostruire la codifica iniziale di $Parent[]$, $CC[]$ e $OC[]$, introduciamo le strutture $Update[]$, $CCSave[]$ e $OCSave[]$. Con $CCSave[]$ e $OCSave[]$ memorizziamo lo stato iniziale di $CC[]$ e $OC[]$, mentre in $Update[]$ inseriamo tutti i nodi che sono stati eliminati: ogni volta che nell'algoritmo di Figura 5.10 incontriamo $Deleted[d_p, position] := d$, memorizziamo $Update[d_p, position] := d$; in questo modo $Update[]$ contiene tutti i nodi per ripristinare $Parent[]$.

```

RESTOREROOT()
1  CC[root] := CCSave[root];
2  OC[root] := OCSave[root];
3  for pos in KidNum[root]
4  do RESTORECHILD(root, pos);
5  CLEANSTACK(ToCheck);

RESTORECHILD(d_p, position)
1  if Update[d_p, position] is not null
2  then d := Update[d_p, position];
3      Update[d_p, position] := null;
4      Deleted[d_p, position] := null;
5      Parent[d] = (d_p, position);
6      CC[d] := CCSave[d];
7      OC[d] := OCSave[d];
8      if SymbolOfNode[d] is not null
9      then Count[SymbolOfNode[d]] := 0;

```

Figura 5.12: Procedure il ripristino delle strutture dati.

Dopo aver applicato l'algoritmo di validazione a una parola w , chiamiamo l'algoritmo in Figura 5.12 che, a partire dalla radice, ripristina i vincoli di co-occorrenza e di ordine copiandoli da $CCSave[]$ e $OCSave[]$. Successivamente, per ogni figlio d richiama una funzione ricorsiva che controlla se il nodo è stato

modificato, ovvero se $Update[d_p, position] = d$ con $d \neq null$. Se è stato modificato, la funzione ripristina il valore di $Parent[d]$, ricopia i vincoli di co-occorrenza e di ordine del nodo e, se d è un atomo, resetta il valore $Count[d]$. Per verificare se d è un atomo utilizziamo l'array $SymbolOfNode[]$ che contiene la codifica inversa di $NodeOfSymbol[]$, ovvero se d è un atomo $a[m..n]$, $SymbolOfNode[d]$ restituisce il simbolo corrispondente a , altrimenti restituisce null. Infine, la funzione si richiama ricorsivamente su tutti i figli di d .

La fase di ripristino non visita tutto il tipo T , ma soltanto la parte che è stata modificata, quindi ha costo $O(\min(|T|, |w| * flatdepth(T)))$. Il costo complessivo per l'algoritmo di validazione su più parole diventa $O(|T| + m * \min(|T|, |w| * flatdepth(T)))$.

Nel caso $|T| < |w|$, possiamo maggiorare $|T|$ con $|w|$ ottenendo un costo pari a $O(|w| + m * |w|)$ ovvero una complessità lineare. Nel caso $|T| > |w|$, il costo per validare m parole diventa $O(|T| + m * |w| * flatdepth(T))$. Assumendo k come limite superiore alla profondità di T abbiamo $O(|T| + m * |w| * k)$, quindi lineare nella dimensione dei dati di ingresso.

5.5 Validazione per XML Schema

In questa sezione passeremo dagli algoritmi che lavorano sull'espressioni regolari e le parole ad algoritmi che lavorano su XML Schema e alberi XML, dove un XML Schema è formalizzato con una EDTDst e i content model sono espressi con il nostro linguaggio dei tipi. Ricordiamo che, in una EDTDst, la funzione μ è iniettiva quando ci stiamo riferendo a uno specifico content model T e indicheremo il suo inverso come μ_T^{-1} .

Consideriamo un albero XML radicato nel simbolo a e con n figli a_1, \dots, a_n , che a loro volta hanno il rispettivo albero t_1, \dots, t_n , ovvero $t = a(a_1(t_1) \dots a_n(t_n))$. Per validare t su una EDTDst, recuperiamo il tipo T del simbolo a tramite la funzione τ , ovvero $T = \tau(a)$; successivamente preleviamo il tipo T_i di ogni figlio a_i calcolando il tipo dell'elemento tramite l'inverso della funzione μ nel content model T , ovvero $T_i = \tau(\mu_T^{-1}(a_i))$. Dopo aver recuperato tutti i content model T_i , eseguiamo la validazione di ogni sottoalbero t_i rispetto al proprio content model,

infine costruiamo la parola $w = \mu_T^{-1}(a_1) \cdot \dots \cdot \mu_T^{-1}(a_n)$ e la validiamo su T tramite l'algoritmo $\text{MEMBERFLATLIN}(w, T)$.

L'approccio appena descritto prevede di utilizzare le seguenti strutture dati che codificano una EDTDst della forma $(\Sigma, \Delta, d, s_d, \mu)$:

- una variabile *RootName* che contiene il nome $s_d \in \Delta$ dell'elemento radice;
- una tabella hash *Tau*[], dove, per ogni nome $a \in \Delta$, *Tau*[a] contiene il content model del nome a , ovvero contiene $\tau(a)$;
- una tabella hash *MuInverse*[], dove, per ogni etichetta $e \in \Sigma$ e content model T , *MuInverse*[e, T] contiene il nome $a \in \Sigma$ dell'etichetta e all'interno del content model T .

Presenteremo adesso due algoritmi che si differenziano su quando viene invocato l'algoritmo di validazione su stringhe. Questi algoritmi utilizzano entrambi un parser SAX, quindi, per applicare il ragionamento che abbiamo appena visto, descriveremo i nostri algoritmi mediante la definizione degli eventi di: inizio del documento, apertura di un tag, chiusura di un tag e fine del documento.

5.5.1 Validazione lazy

Questo algoritmo prende la denominazione di *lazy* in quanto, durante l'apertura dei tag, si limita a costruire la parola e la validazione viene effettuata alla chiusura di un tag. L'algoritmo procede seguendo il seguente meccanismo:

- all'inizio del documento l'algoritmo codifica tutti i tipi del content model della EDTDst e crea due pile:
 - una pila *TypeStack* che contiene i nomi dei content model aperti (per i quali abbiamo incontrato il tag di apertura ma non di chiusura), inizialmente vuota, e, come elemento in testa alla pila $\text{top}(\textit{TypeStack})$, contiene l'ultimo nome che abbiamo aperto;
 - una pila *WordStack* che contiene tutte le parole che devono essere ancora validate, inizialmente inizializzata con una parola vuota, e, come elemento in testa alla pila $\text{top}(\textit{WordStack})$, contiene la parola che deve essere validata dal content model di $\text{top}(\textit{TypeStack})$.

- all'apertura di un elemento e , se *TypeStack* non è vuota, l'algoritmo calcola il nome $\mu_{top(TypeStack)}^{-1}(e)$ e lo aggiunge in testa a *TypeStack*, altrimenti inserisce il simbolo iniziale s_d ; successivamente inserisce su *WordStack* una parola vuota;
- alla chiusura di un elemento e l'algoritmo prende e rimuove gli elementi in testa alle due pile $T = \tau(top(TypeStack))$ e $w = top(WordStack)$; successivamente applica l'algoritmo di validazione a w e T , in caso di fallimento l'algoritmo termina rifiutando l'albero XML, altrimenti ripristina la codifica di T tramite le funzioni presentate nella sezione precedente e, dopo aver eliminato w e T dalle pile, aggiunge il simbolo $\mu_{top(TypeStack)}^{-1}(e)$ in fondo alla parola in testa alla pila *WordStack*;
- alla fine del documento l'algoritmo verifica che la pila *TypeStack* sia vuota e la pila *WordStack* contenga una sola parola che a sua volta contiene solo il simbolo iniziale s_d .

5.5.2 Validazione eager

Questa versione dell'algoritmo prevede che, a ogni apertura di un tag, venga subito richiamato l'algoritmo di validazione sul simbolo appena aperto, calcolando il residuo prima di aggiungere elementi alle pile, mentre alla chiusura del tag viene eseguito soltanto il controllo sulla cardinalità e di presenza di ϵ nel residuo generato dalla lettura di tutti i simboli.

Questo comporta la necessità di dividere l'algoritmo in tre parti per poter calcolare e memorizzare il residuo parziale ogni volta che viene letta l'apertura di un tag:

- la prima parte prevede l'inizializzazione, o il ripristino, di tutte le strutture dati necessarie;
- la seconda parte prevede il calcolo del residuo aggiornando tutte le strutture dati come abbiamo visto, successivamente le strutture dati vengono salvate in attesa della lettura di un altro simbolo;

- la terza parte prevede l'esecuzione di tutte le operazioni che negli algoritmi presentati erano alla fine della lettura di tutti i simboli.

La suddivisione dell'algoritmo in queste tre parti ci obbliga a memorizzare lo stato dell'algoritmo ogni volta che incontriamo un'apertura di un tag, anche se si tratta di un tag che avevamo precedentemente aperto (si pensi a una ricorsione). L'algoritmo di validazione su dati XML procede seguendo il seguente meccanismo:

- all'inizio del documento l'algoritmo codifica tutti i tipi del content model della EDTDst e crea due pile:
 - una pila *TypeStack* che contiene i nomi dei content model aperti (per i quali abbiamo incontrato il tag di apertura ma non di chiusura), inizialmente vuota, e come elemento in testa alla pila $top(TypeStack)$ contiene l'ultimo nome che abbiamo aperto;
 - una pila *StateStack* che contiene tutti gli stati dei simboli aperti che non sono stati chiusi, ovvero che sono sempre nella seconda fase in attesa di simboli da leggere, inizialmente comprende lo stato iniziale del tipo corrispondente al simbolo iniziale, e come elemento in testa alla pila $top(StateStack)$ contiene lo stato che analizzerà il prossimo simbolo.
- all'apertura di un elemento e , l'algoritmo recupera $top(StateStack)$ e calcola il residuo per la lettura del simbolo $a = \mu_{top(TypeStack)}^{-1}(e)$; se il calcolo restituisce un valore di non accettazione, l'algoritmo termina non accettando l'albero XML. Se il residuo viene calcolato accettando il simbolo, viene creata una copia dello stato iniziale del tipo corrispondente al simbolo a , ovvero $\tau(a)$, che viene inserita in testa a *StateStack*;
- alla chiusura di un elemento e l'algoritmo preleva e rimuove gli elementi in testa alle due pile $T = top(TypeStack)$ e $S = top(StateStack)$, successivamente applica l'algoritmo di chiusura per lo stato S , se il risultato non è di accettazione l'algoritmo termina non accettando l'albero XML, altrimenti prosegue;

- alla fine del documento l'algoritmo verifica che la pila *TypeStack* sia vuota e la pila *TypeStack* contenga un solo stato S relativo al tipo del simbolo iniziale sul quale applichiamo l'algoritmo di chiusura e, se il risultato non è di accettazione, l'algoritmo rifiuta l'albero XML, altrimenti lo accetta.

5.5.3 Differenze tra gli algoritmi lazy e eager

Gli algoritmi che abbiamo presentato hanno una struttura comune molto simile differenziandosi, di fatto, su quando viene richiamato l'algoritmo di validazione per le parole.

Dal punto di vista computazionale il costo in tempo della versione lazy è dato da una fase iniziale per codificare tutti i tipi del content model, che indichiamo con $O(|\tau|)$, più la validazione di m stringhe che vengono costruite prima di applicare l'algoritmo nella sua interezza, per poi ripristinare lo stato iniziale. Il difetto di questo algoritmo è dato dallo spazio occupato: l'algoritmo ricostruisce per intero tutta la parola prima di applicare un algoritmo di validazione sulle parole, quindi il costo in spazio è lineare con la dimensione delle parole presenti nell'albero XML.

Il costo in tempo della versione eager si differenzia dal costo lazy in quanto gli stati devono essere copiati ogni volta che si incontra l'apertura di un elemento, quindi non utilizziamo il costo della procedura di ripristino che lavora soltanto sui nodi visitati, ma dobbiamo copiare l'intera codifica. Considerando che il costo computazionale della copia del tipo T sia $O(T)$, dove T è il tipo più grande, il costo dell'algoritmo eager risulta $O(|\tau| + m * (|T| + |w|))$. Dal punto di vista dello spazio questo algoritmo non memorizza tutta la parola, ma soltanto gli stati dei residui che crescono al crescere della profondità dell'albero XML e non dell'ampiezza, quindi il costo in spazio è lineare con la profondità degli alberi XML.

Nel prossimo capitolo vedremo che, in fase sperimentale, questi due algoritmi si comportano allo stesso modo e le differenze che abbiamo presentato sono marginali nella pratica.

Capitolo 6

Implementazione e sperimentazione

In questo capitolo parleremo degli aspetti implementativi più rilevanti al fine di realizzare gli algoritmi di inclusione e validazione proposti nei capitoli precedenti; successivamente parleremo delle sperimentazioni eseguite su tali algoritmi descrivendone le metodologie e i risultati ottenuti e confrontandoci con alcuni strumenti simili.

6.1 Aspetti implementativi

In questa sezione mostreremo come sono rappresentati i tipi e gli algoritmi, e come la rappresentazione dei tipi in memoria venga costituita partendo da espressioni che rispettano la sintassi che abbiamo dato. Infine, parleremo dei test di regressione.

Gli algoritmi sono stati implementati in Java 6, utilizzando Eclipse come strumento di sviluppo.

6.1.1 Tipi e algoritmi

Per rappresentare la struttura di un tipo T abbiamo inizialmente introdotto una generalizzazione di operatore mediante un'interfaccia; successivamente abbiamo creato una classe per ogni operatore presente nella grammatica per rappresentare

i tipi in modo da poter costruire l'albero sintattico dove l'operatore di un nodo è dato dal nome della classe (Figura 6.1).

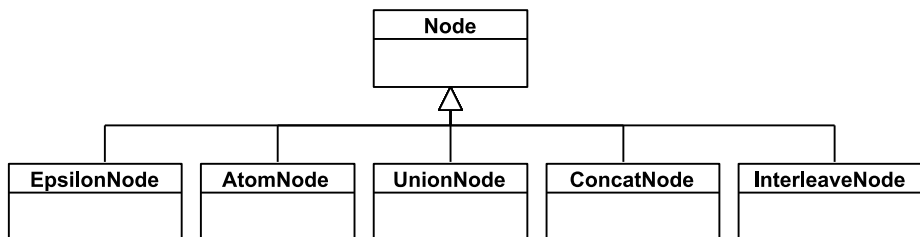


Figura 6.1: *Struttura delle classi per rappresentare un tipo.*

I tipi che rappresentiamo con questa struttura sono operatori binari e non n-ari; questo implica che le classi UNIONNODE, INTERLEAVENODE e CONCATNODE hanno esattamente due figli, un figlio destro e un figlio sinistro. In queste classi abbiamo inserito dei metodi ricorsivi che calcolano le funzioni che abbiamo definito sui tipi, ovvero il predicato $N()$, la funzione $S()$ e la funzione $Atoms()$ che abbiamo visto nel Capitolo 3.

Analogamente ai tipi, gli algoritmi sono generalizzati mediante due interfacce, una per gli algoritmi di inclusione e una per quelli di validazione. Gli algoritmi di inclusione hanno un metodo che, come parametri, prende due tipi e restituisce un valore booleano, mentre gli algoritmi di validazione ne hanno uno che prende una parola, un tipo, e restituisce un valore booleano. La realizzazione di un algoritmo prevede l'implementazione della rispettiva interfaccia scrivendone il metodo relativo.

6.1.2 Il parser

Il parser serve per costruire la rappresentazione in memoria, mediante le classi appena viste, di un tipo, partendo dalla grammatica vista nel Capitolo 3. Questo parser è stato costruito automaticamente mediante una libreria di generatori di parser e apportando alcune modifiche alla grammatica che abbiamo precedentemente visto.

Prendendo una espressione qualunque, non completamente parentesizzata, è possibile costruire più alberi binari per rappresentarla in memoria, a seconda del-

l'ordine di operatori che consideriamo. Inoltre, la semantica di tali alberi è, in generale, diversa. Quindi, è necessario che le espressioni che vogliamo rappresentare siano *termini*, ovvero espressioni completamente parentesizzate, per poter essere descritte da un unico albero. Nel caso che un'espressione non lo sia, stabiliamo una precedenza tra gli operatori in modo da indicare un unico termine, con una semantica ben precisa, tra tutti i possibili termini che l'espressione rappresenta.

La precedenza che abbiamo scelto è: $\cdot > \& > +$; quindi, la grammatica per rappresentare i tipi viene riscritta come:

$$\begin{aligned} T &::= A \mid A + T \\ A &::= B \mid B \& A \\ B &::= C \mid C \cdot B \\ C &::= \epsilon \mid a[m..n] \mid (T) \end{aligned}$$

Esempio 6.1. Consideriamo il tipo $T = a[1..1] \cdot b[1..1] + c[1..1]$. Introducendo le parentesi, possiamo scrivere il tipo T come $(a[1..1] \cdot b[1..1]) + c[1..1]$ oppure come $a[1..1] \cdot (b[1..1] + c[1..1])$, la cui semantica è rispettivamente $\{ab, c\}$ e $\{ab, ac\}$.

Considerando la precedenza di operatori che abbiamo introdotto, il tipo T rappresenta soltanto il termine $(a[1..1] \cdot b[1..1]) + c[1..1]$, mentre se vogliamo scrivere $a[1..1] \cdot (b[1..1] + c[1..1])$ dobbiamo esplicitare le parentesi.

6.1.3 Test di regressione

Il codice prodotto durante la realizzazione degli algoritmi è accompagnato da un'ampia schiera di test di regressione. Per ogni classe Java realizzata è presente una classe con lo stesso nome, con l'aggiunta della parola *Test* alla fine, che prevede una serie di prove al fine di valutare la robustezza dell'algoritmo. Ad esempio, per la classe chiamata *InclusionAlgorithm*, i test sono contenuti nella classe *InclusionAlgorithmTest*.

L'insieme delle prove di una classe contiene un numero sufficiente di casi per far eseguire tutti i possibili rami dei condizionali presenti nel codice della classe, compresi i controlli sugli errori. Inoltre, i test sui vari algoritmi prevedono prove sia positive, ovvero dove l'algoritmo deve ritornare un valore di accettazione, sia negative, ovvero dove l'algoritmo deve ritornare un valore di rifiuto.

Ogni test è stato scritto durante l'implementazione della rispettiva classe in modo da verificare immediatamente eventuali errori di implementazione. Inoltre, tutti i test così prodotti sono stati aggiunti a quelli precedenti, in modo da formare una batteria di test. Tale batteria viene verificata dopo ogni modifica all'implementazione degli algoritmi per evitare che, se si apportano alcune modifiche al codice, tali modifiche vadano a danneggiare il comportamento di quelle parti che non sono state modificate. Eclipse, mediante l'integrazione in questo ambiente di una libreria specializzata denominata *JUnit*, ci permette di scrivere i test con il linguaggio Java e di eseguirli tutti, o in parte, semplicemente premendo un tasto dell'interfaccia grafica.

6.2 Sperimentazione

In questa sezione parleremo della generazione dei dati utilizzati per fare i test e delle modalità dei test stessi, sia per quanto riguarda gli algoritmi di inclusione sia per gli algoritmi di validazione. Inoltre, confronteremo i risultati degli algoritmi di validazione con i rispettivi algoritmi dei sistemi Harmony e Xerces.

La sperimentazione di tutti gli algoritmi che abbiamo descritto è stata eseguita su un Pentium(R) D con processore dual core a 3GHz e 2 GB di memoria RAM. Il sistema operativo utilizzato sia per i nostri algoritmi che per i sistemi con il quale ci confronteremo è stato Linux.

I grafici che mostreremo da qui in avanti visualizzeranno i tempi ottenuti dopo aver eseguito per dieci volte l'algoritmo, aver calcolato i tempi medi delle esecuzioni, togliendo sia il tempo migliore che quello peggiore. Inoltre, i valori presentati nei grafici sono stati generati selezionando degli intervalli per l'asse delle ascisse e calcolando la media di tutti gli esperimenti che cadevano in quel dato intervallo. Tale semplificazione si è resa necessaria per avere dei grafici che facciano capire il costo medio dei vari algoritmi.

6.2.1 Inclusione

Nei capitoli precedenti abbiamo visto tre differenti algoritmi per verificare l'inclusione tra due tipi:

- l'algoritmo base, che ha un costo cubico;
- l'algoritmo che utilizza la chiusura veloce, che ha un costo quadratico;
- l'algoritmo strutturale, che ha un costo lineare se non viene mai richiamato uno degli altri due algoritmi.

Per questi tre algoritmi genereremo due insiemi di tipi, con diverse caratteristiche, sul quale verificare l'inclusione e confrontarne i tempi.

Generazione dei tipi

La costruzione del primo insieme di dati per la sperimentazione prevede la suddivisione in due fasi: durante la prima fase generiamo una vasta quantità di tipi casuali, mentre nella seconda fase costruiamo delle coppie sottotipo candidato/-tipo candidato tramite il prodotto cartesiano di tutti i tipi generati, generando, così, da n tipi, n^2 coppie di tipi.

L'algoritmo di generazione dei tipi casuali (Figura 6.2) costruisce gli alberi che rappresentano i tipi come se gli operatori fossero n -ari, e non binari, per generare tipi più realistici. L'algoritmo è configurabile mediante le seguenti strutture:

- una variabile *expectedDepth* contenente la profondità media dell'albero che vogliamo generare;
- una variabile *expectedChild* contenente il numero medio di figli che viene generato per ogni operatore;
- una variabile *epsilonProbability* contenente la probabilità di avere atomi $a[m..n]$ dove $m = 0$;
- una tabella hash *probabilities*[], indicizzata sui simboli degli operatori, dove, per ogni operatore $\otimes \in \{+, \&, \cdot\}$, *probabilities*[\otimes] contiene la probabilità che sia generato un operatore \otimes .

L'algoritmo (Figura 6.2), in fase di generazione, prevede una prima distinzione tra la creazione un nodo foglia, ovvero un atomo $a[m..n]$, dove $n \in \{0, 1\}$ e $m = 1$, oppure la generazione di un nodo intermedio, ovvero un nodo $T_1 \otimes \dots \otimes T_n$, dove

$\otimes \in \{+, \cdot, \&\}$, tramite la funzione `GENLEAF`. Tale funzione, data una profondità di input *depth*, restituisce *true* se tale profondità non è superiore alla profondità calcolata da una variabile aleatoria di Poisson con media *expectedDepth*.

Quando l'algoritmo genera un nodo foglia, procede nell'invocare la funzione `SELECTSYMBOL`, che genera un simbolo *a* sempre differente ad ogni chiamata. Successivamente, l'algoritmo distingue se $m = 1$ oppure se $m = 0$; nel primo caso genera $a[1..1]$ mentre nel secondo caso genera $a[1..1] + \epsilon$. Questa seconda distinzione viene fatta utilizzando la funzione `GENEPS` che modella una variabile aleatoria Bernoulliana di parametro *epsilonProbability*.

Quando l'algoritmo genera un nodo con un operatore, procede nell'invocare la funzione `SELECTOPERATOR`, che genera casualmente un operatore \otimes rispettando le probabilità di *probabilities[]*. Successivamente, l'algoritmo chiama `SELECTCHILDNUMBER`, che, modellando una variabile aleatoria di Poisson con media *expectedChild*, calcola un numero casuale *n* di figli dell'operatore \otimes . Infine, la funzione si richiama ricorsivamente *n* volte per generare i figli dell'operatore, e restituisce l'albero risultante.

```

GENTREE(depth)
1  if GENLEAF(depth)
2    then a = SELECTSYMBOL();
3      if GENEPS()
4        then return a[1..1];
5        else return a[1..1] + ε;
6  else ⊗ = SELECTOPERATOR();
7      n = SELECTCHILDNUMBER();
8      for i = 1; i <= n; i = i + 1
9        do child[i] = GENTREE(depth + 1);
10     return child[1] ⊗ ... ⊗ child[n];

```

Figura 6.2: Algoritmo per la generazione di tipi casuali.

Utilizzando questo algoritmo, abbiamo generato cinquecento tipi differenti con una profondità media pari a 2 e un numero di figli medio pari a 30, mentre la

selezione degli operatori è stata equiprobabile. Abbiamo fatto questa scelta per avere dei tipi poco profondi ma molto grandi, dato l'alto numero di figli che ogni operatore possiede, per avere dei tipi casuali che avessero la stessa struttura della maggior parte dei tipi XML usati realmente.

Il primo insieme lo abbiamo generato utilizzando i cinquecento tipi prodotti tramite un prodotto cartesiano tra gli stessi: per ogni tipo abbiamo confrontato se era un sottotipo di tutti gli altri, e abbiamo suddiviso le coppie di tipi che verificavano l'inclusione da quelle che non la verificavano generando circa ventimila coppie (U, T) tali che $U \subseteq T$.

Questo insieme è caratterizzato da tipi U molto più piccoli dei tipi T in quanto risulta poco probabile che, generando due tipi molto grandi in modo casuale, essi soddisfino l'inclusione.

La costruzione del secondo insieme utilizza anch'essa una suddivisione in due fasi: durante la prima fase generiamo i tipi casuali, mentre nella seconda fase, per ogni tipo, costruiamo un tipo, strutturalmente simile al primo, ma con un numero k di piccole differenze.

Le differenze tra i tipi possono essere:

- cambio di un operatore per un nodo interno, lasciando inalterato il numero e l'ordine dei figli;
- aggiunta di un figlio, in posizione casuale tra i precedenti figli del nodo, per un nodo interno;
- rimozione di un figlio casuale per un nodo interno;
- inversione della posizione di due figli per un nodo interno;
- modifica dei vincoli di conteggio per un atomo;
- aggiunta di ϵ per un atomo che ne risulta sprovvisto;

Tali differenze sono generate casualmente e, analogamente all'algoritmo precedente, configurabili tramite le seguenti strutture dati:

- una tabella hash *probabilities*[], dove, per ogni possibile modifica, contiene la probabilità che tale modifica sia generata;

- una matrice *change*, che indica le probabilità, nel caso la modifica sia il cambio di un operatore, di trasformare un operatore in uno degli altri due.

L'algoritmo per generare i tipi modificati prende il tipo di ingresso e calcola casualmente, rispettando la tabella delle probabilità, quale modifica effettuare e a quale nodo, successivamente la applica e ripete il ciclo fino a quando non ha applicato tutte e k le modifiche.

Questo secondo insieme è caratterizzato da tipi U e T molto simili, sia strutturalmente che di grandezza, ed è stato ideato principalmente per valutare l'algoritmo strutturale, ma lo utilizzeremo per tutti e tre gli algoritmi che abbiamo proposto.

Il secondo insieme è stato generato utilizzando questo meccanismo, per questo lo chiameremo *insieme modificato*, e sono stati generate circa ventimila coppie di tipi con $k = 1$, quindi con una sola differenza tra sottotipo e tipo. Insieme con $k > 1$ sono stati generati ma, non essendoci differenze sostanziali in termini di tempi rispetto all'insieme con $k = 1$, sono stati accantonati e utilizzeremo soltanto quello con una differenza.

Entrambi gli insiemi di coppie che abbiamo costruito sono stati ulteriormente suddivisi in due insiemi: l'insieme delle coppie che soddisfano l'inclusione e l'insieme delle coppie che non la soddisfano. Nelle sperimentazioni che presenteremo abbiamo considerato soltanto l'insieme delle coppie che soddisfano l'inclusione in quanto gli algoritmi devono verificare tutti i vincoli, mentre nel caso in cui la coppia non soddisfi l'inclusione, l'algoritmo si interrompe al primo vincolo violato, impiegando un tempo dipendente da dove sta la violazione piuttosto che dalla grandezza dei due tipi. Le sperimentazioni sulle coppie false è stata comunque fatta ma non verrà riportata in quanto poco significativa.

Insieme casuale

I primi esperimenti sono stati effettuati sull'insieme creato casualmente che abbiamo visto in precedenza. In Figura 6.3 osserviamo il grafico dei tempi dei tre algoritmi e notiamo subito una sovrapposizione quasi perfetta tra l'algoritmo quadratico e quello cubico; tale sovrapposizione sembra avere un andamento quadratico. Inoltre, i tempi dell'algoritmo strutturale sono nettamente inferiori

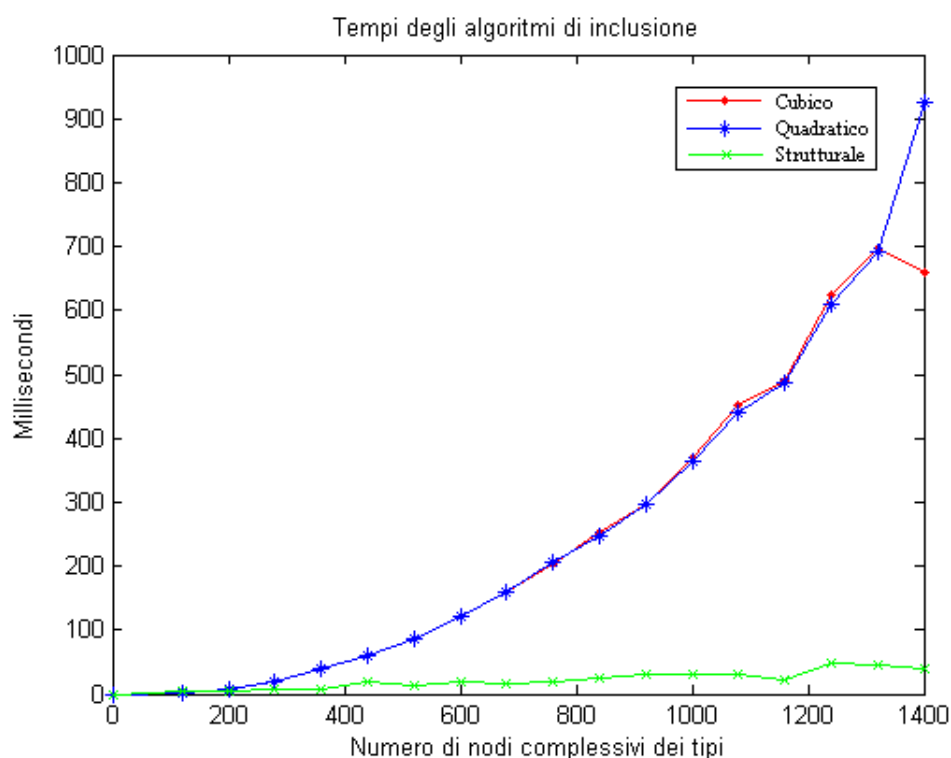


Figura 6.3: Grafico dei tempi relativi ai diversi algoritmi di inclusione misurati sull'insieme casuale.

rispetto agli altri, oltre ad avere un andamento lineare. Questo è dovuto al fatto che l'algoritmo strutturale riesce ad applicare alcune volte le regole *focus* e *divide* riducendo notevolmente la dimensione dei due tipi prima di richiamare l'algoritmo cubico.

Per capire meglio la sovrapposizione tra l'algoritmo quadratico e quello cubico, ne andiamo a studiare i tempi delle singole parti dell'algoritmo, ovvero suddividiamo i tempi nel calcolo dei: vincoli piatti, vincoli di co-occorrenza, vincoli di ordine e costruzione delle strutture per l'algoritmo LCA. In Figura 6.4 possiamo vedere i tempi suddivisi, dove mostriamo l'algoritmo quadratico soltanto nel calcolo dei vincoli di co-occorrenza, in quanto unica differenza tra questi due algoritmi.

Come possiamo vedere nella Figura 6.4, il costo dell'algoritmo è pesantemente influenzato dal calcolo dei vincoli di ordine. Questo è dovuto al fatto che il costo quadratico del calcolo dei vincoli di ordine è quadratico sul numero di simboli del

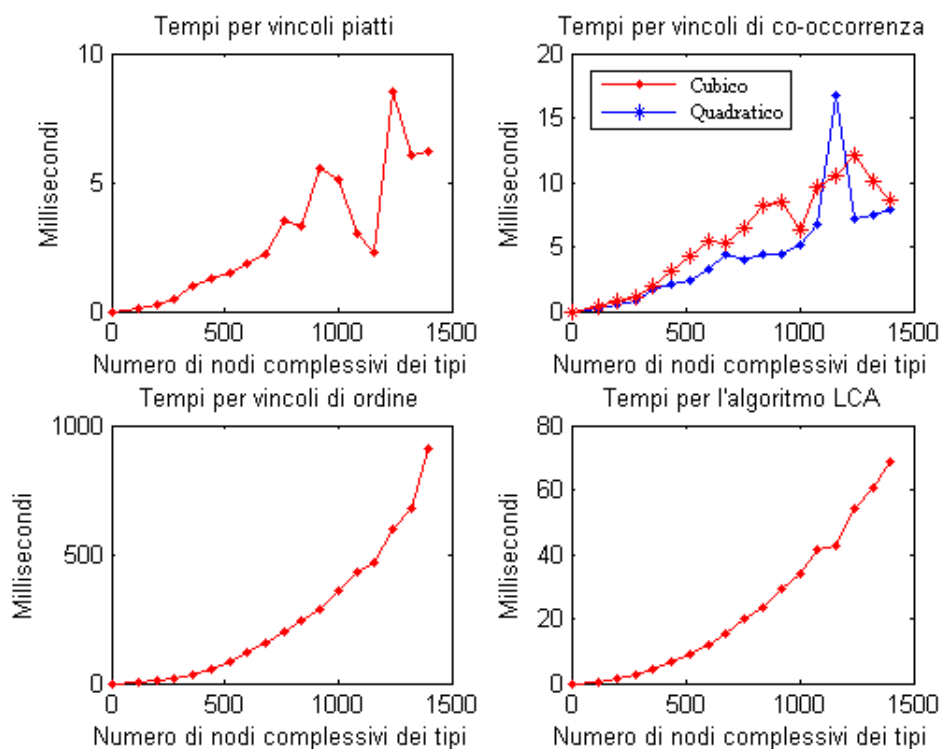


Figura 6.4: *Grafici dei tempi parziali relativi agli algoritmi di inclusione misurati sull'insieme casuale.*

tipo U , che nel nostro insieme è generalmente molto più grande del sottotipo T ; inoltre, tale costo rappresenta un limite inferiore, ciò vuol dire che il tempo è sempre quadratico.

I tempi relativi ai vincoli di co-occorrenza sono molto simili sia per la variante cubica che quadratica, nonostante l'algoritmo quadratico sia più veloce. Questo è dovuto al fatto che il costo computazionale del calcolo dei vincoli di co-occorrenza dipende molto dalla dimensione del sottotipo T . Inoltre, il costo della verifica dei vincoli di co-occorrenza è un limite superiore, ciò significa che l'algoritmo è cubico, o quadratico, nel caso pessimo, ma in pratica può comportarsi molto meglio.

Dai risultati su questo insieme risulta che la parte più pesante, sulla quale studiare soluzioni più efficienti, è la verifica dei vincoli di ordine, mentre la sostituzione dell'algoritmo di Beeri-Bernstein con l'algoritmo per la chiusura veloce rappresenta un miglioramento marginale, ma in realtà, nel prossimo esperimento,

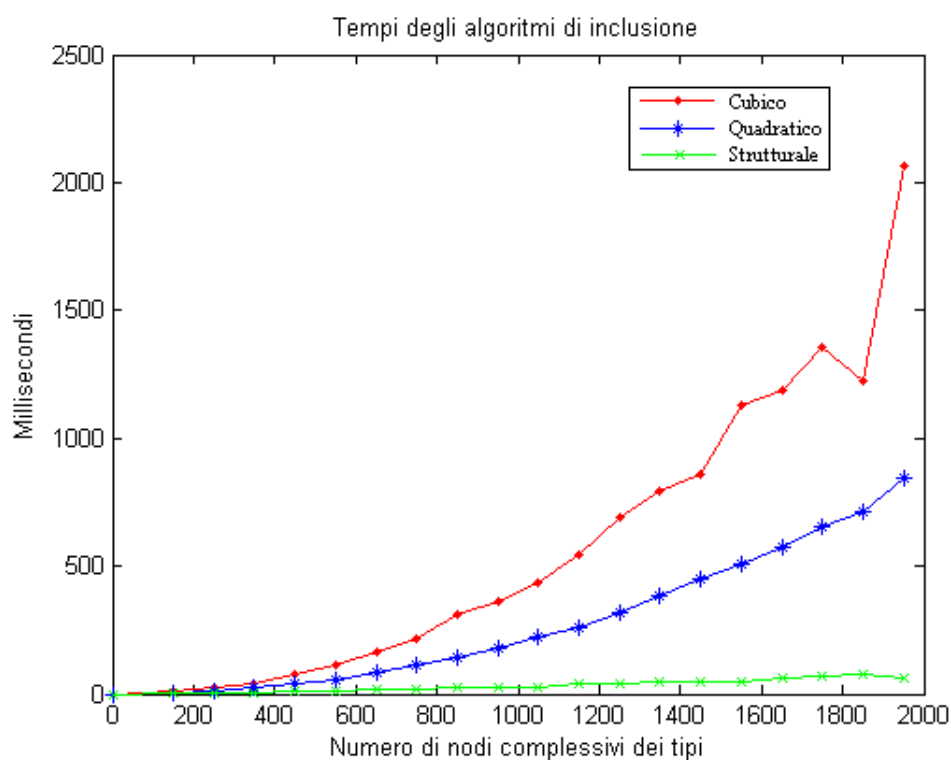


Figura 6.5: *Grafico dei tempi relativi ai diversi algoritmi di inclusione misurati sull'insieme modificato.*

vedremo che ciò non è sempre vero.

Insieme modificato

Un secondo gruppo di esperimenti è stato effettuato sull'insieme dei tipi costruiti tramite modifiche, che abbiamo descritto in precedenza.

In Figura 6.5 osserviamo il grafico dei tempi dei tre algoritmi e notiamo subito che la sovrapposizione che avevamo tra l'algoritmo quadratico e cubico è scomparsa; l'algoritmo con la chiusura di Beer-Bernstein ha un andamento cubico, mentre quello con la chiusura veloce ha un andamento quadratico.

I tempi dell'algoritmo strutturale restano nettamente inferiori rispetto agli altri e di andamento lineare. Questo è dovuto al fatto che l'algoritmo strutturale lavora molto bene con coppie di tipi che hanno una sola differenza, riuscendo

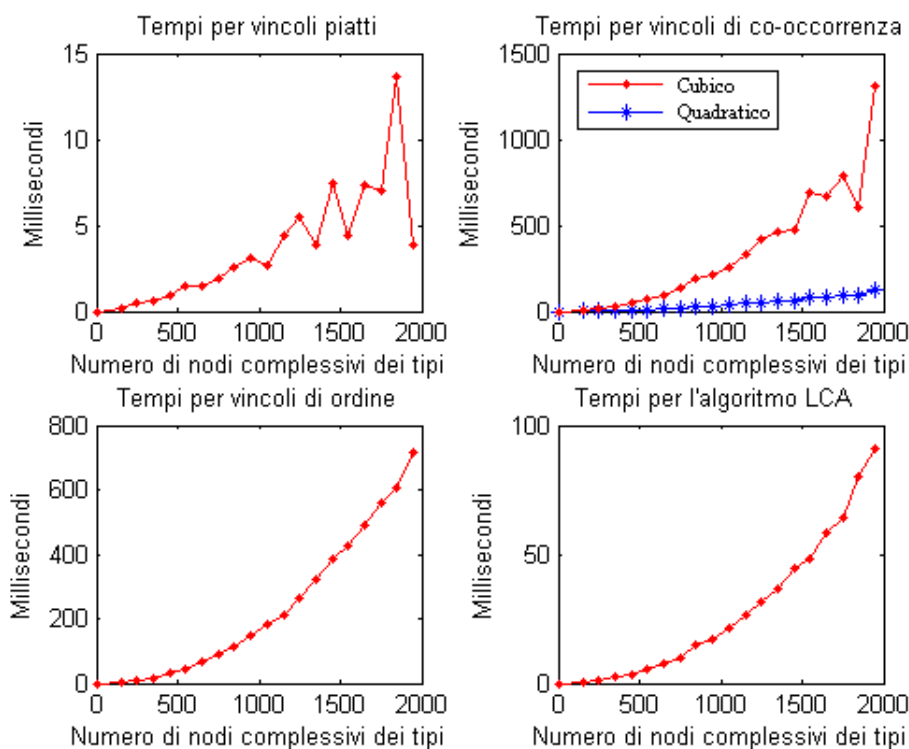


Figura 6.6: Grafici dei tempi parziali relativi agli algoritmi di inclusione misurati sull'insieme modificato.

ad applicare molte volte le regole *focus* e *divide* che riducono notevolmente la dimensione dei tipi prima di richiamare l'algoritmo quadratico di inclusione.

Osservando la Figura 6.6, vediamo che il costo dell'algoritmo cubico è, in questo caso, influenzato dal calcolo dei vincoli di co-occorrenza, e non dal calcolo dei vincoli di ordine, come nel precedente insieme. Questo è dovuto al fatto che il sottotipo T , che incide notevolmente sul costo nel calcolare la chiusura all'indietro, è di dimensioni molto simili al tipo U , quindi l'andamento cubico dell'algoritmo prende il sopravvento sulla verifica dei vincoli di ordine.

Da questi risultati possiamo dedurre che, quando la dimensione di T e di U sono paragonabili, la sostituzione dell'algoritmo di Beeri-Bernstein con l'algoritmo per la chiusura veloce è un miglioramento concreto dell'algoritmo e non soltanto teorico, migliorando notevolmente, in fase sperimentale, l'efficienza dell'algoritmo di inclusione.

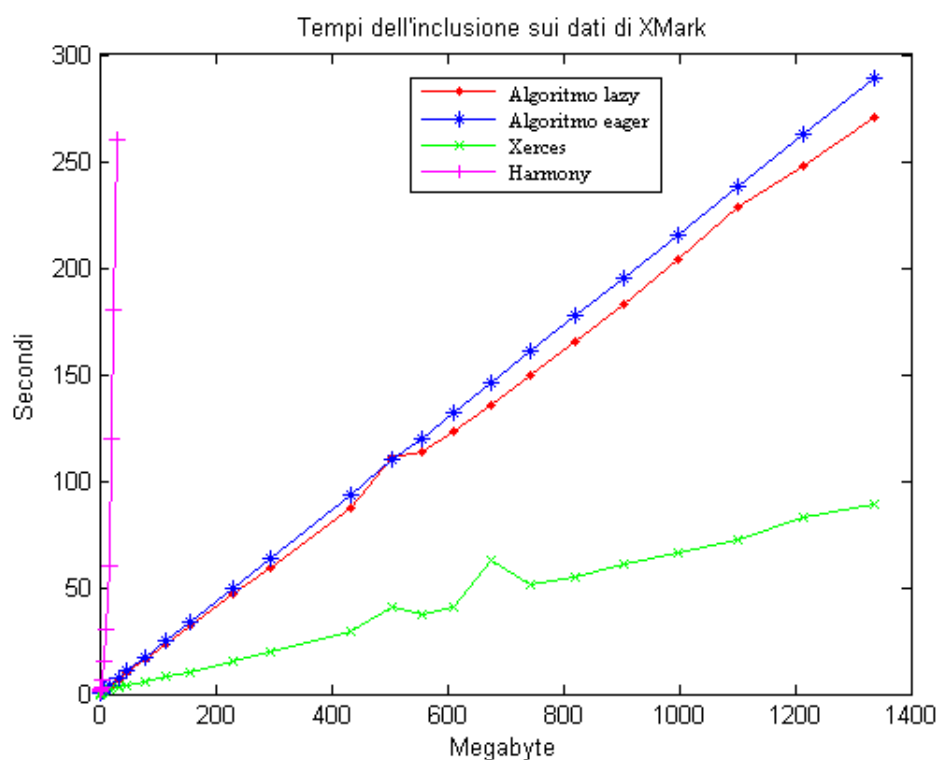


Figura 6.7: Grafico dei tempi relativi agli algoritmi di validazione misurati sull'insieme generato da XMark.

6.2.2 Validazione

Nei capitoli precedenti abbiamo visto due differenti algoritmi per la validazione di un documento XML:

- l'algoritmo lazy, che ha un costo in tempo di $O(|\tau| + m * |w|)$ e costo in spazio lineare con la dimensione delle parole;
- l'algoritmo eager, che ha un costo in tempo di $O(|\tau| + m * (|T| + |w|))$ e costo in spazio lineare con la profondità dell'albero.

Per questi algoritmi prenderemo come riferimento schemi XML realmente esistenti e ne genereremo molte istanze; successivamente misureremo i tempi dei nostri algoritmi. Infine, confronteremo i tempi dei nostri due algoritmi con i sistemi Harmony e Xerces, di cui abbiamo già parlato.

Generazione dei dati

I dati XML sono stati generati utilizzando due diversi sistemi: XMark ([SWK⁺01]) e Taxi ([BGMP07a]). XMark è stato impiegato in quanto rappresenta uno strumento molto conosciuto, e utilizzato, da chi lavora con dati e documenti XML, mentre Taxi è uno strumento meno conosciuto, essendo ancora in sviluppo, ma sicuramente più versatile, e lo abbiamo utilizzato per misurare i nostri algoritmi su schemi diversi da quello proposto da XMark.

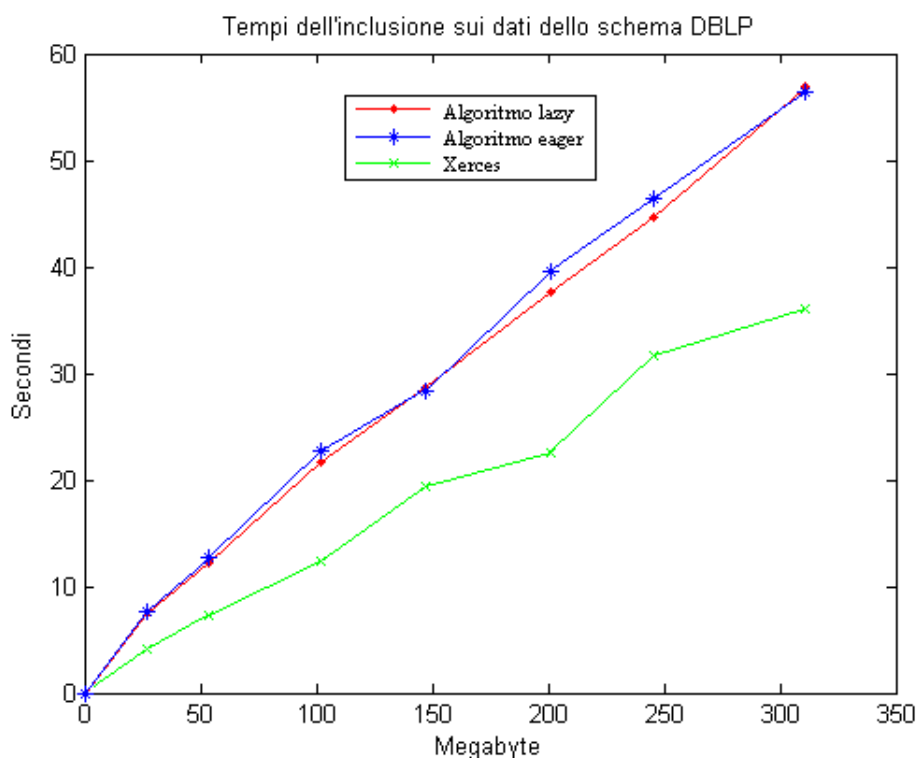


Figura 6.8: *Grafico dei tempi relativi agli algoritmi di validazione misurati sull'insieme DBLP.*

Lo schema XML di XMark contiene tutti gli operatori che abbiamo presentato, ovvero è presente sia la concatenazione che l'unione. Inoltre, nello schema originale di XMark, è presente un content model nella forma $(a_1 + \dots + a_n)^*$ che noi abbiamo riscritto come $a_1 [0..1] \& \dots \& a_n [0..1]$, quindi è presente anche l'operatore di interleaving, nonostante in XML non sia presente. Utilizzando XMark

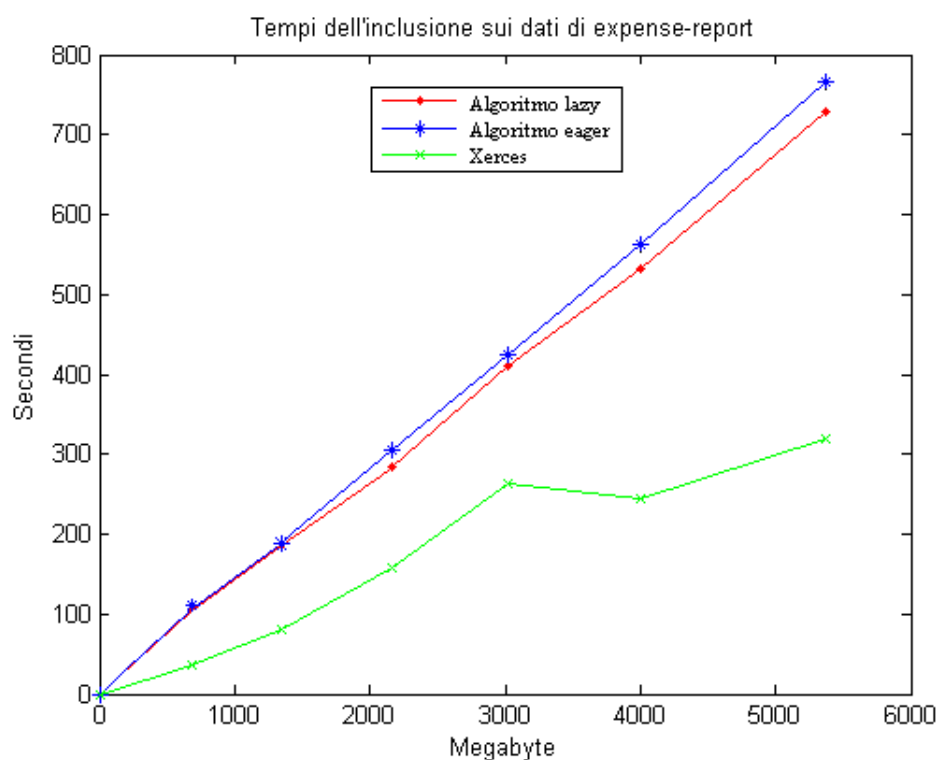


Figura 6.9: Grafico dei tempi relativi agli algoritmi di validazione misurati sull'insieme *expense-report*.

abbiamo costruito l'insieme di test generando circa trecento istanze XML, le quali variano in dimensione da circa 27 KB a circa 1,5 GB.

In Figura 6.7 possiamo vedere i tempi degli algoritmi lazy e eager confrontati con i tempo del sistema Xerces e Harmony. I dati relativi a Harmony arrivano a istanze di soli 30 MB, in quanto la memoria non era sufficiente per istanze più grandi. Come possiamo osservare gli algoritmi che abbiamo proposto sono circa il triplo più lenti del validatore di Xerces, ma ricordiamoci che in questo sistema non è presente l'operatore di interleaving che noi utilizziamo.

Il secondo e il terzo schema che abbiamo utilizzato si chiamano rispettivamente *DBLP* e *expense-report* che possiamo vedere nell'Allegato A. Abbiamo scelto questi due schemi per due motivi: il primo motivo è che questi schemi sono utilizzati in pratica, mentre lo schema di XMark è soltanto creato per fare benchmark. Il secondo motivo è che gli schemi si differenziano tra di loro negli operatori uti-

lizzati: DBLP utilizza fortemente l'operatore di concatenazione, determinando, quindi, istanze con parole molto lunghe, mentre expense-report utilizza molto l'operatore di unione, determinando, quindi, alberi XML più profondi ma con parole più corte.

Le istanze XML di questi due schemi sono state generate utilizzando Taxi; abbiamo generato circa mille istanze per ogni schema di dimensione variabile da 1 KB a 300 MB per DBLP, e da 1 KB a 5 GB per expense-report.

In Figura 6.8 e in Figura 6.9 troviamo rispettivamente i dati relativi alle misurazioni utilizzando l'insieme DBLP e expense-report. Su questi due insiemi non ci siamo confrontati anche con Harmony, in quanto risultava estremamente complicato riscrivere gli schemi dei tipi con il formalismo richiesto da questo sistema. I risultati sono del tutto simili a quelli ottenuti con l'insieme di XMark.

Concludendo, possiamo affermare che l'algoritmo che abbiamo costruito si comporta molto bene, e risulta soltanto tre volte più lento di un sistema come Xerces. Inoltre, abbiamo realizzato che il comportamento dell'algoritmo è realmente lineare nonostante la presenza dell'interleaving.

Capitolo 7

Conclusioni

In questa tesi abbiamo descritto una classe di tipi XML che contiene l'operatore di conteggio e l'operatore di interleaving definendone sintassi e semantica. Successivamente, abbiamo introdotto i vincoli e abbiamo descritto come si estrae un insieme di vincoli da un tipo XML. Infine, abbiamo utilizzato i vincoli per disegnare algoritmi polinomiali per i problemi relativi all'inclusione e alla validazione.

Il problema dell'inclusione è stato risolto costruendo un sistema di deduzione, corretto e completo, basato sui vincoli. Successivamente abbiamo visto tre differenti algoritmi:

- l'algoritmo cubico, che utilizza il sistema di deduzione presentato e l'algoritmo di Beeri-Bernstein esteso per fare la chiusura all'indietro;
- l'algoritmo quadratico, che utilizza il sistema di deduzione presentato e l'algoritmo di chiusura veloce che abbiamo presentato;
- l'algoritmo lineare, che applica delle semplificazioni basandosi sulla struttura simile dei tipi e senza eseguire backtracking.

Il problema della validazione è stato risolto tramite la tecnica di residuazione, che modifica i vincoli dopo la lettura di ogni simbolo della parola. Successivamente, abbiamo presentato l'algoritmo di validazione su stringhe mostrando, nel dettaglio, alcune migliorie che ne abbassano la complessità fino a renderlo lineare. Infine, abbiamo visto due algoritmi, entrambi lineari, per la validazione su documenti XML, che utilizzano l'algoritmo precedente.

La tesi si conclude illustrando le sperimentazioni svolte sia per gli algoritmi di inclusione che per quelli di validazione mostrando le differenze tra di loro e confrontando gli algoritmi di validazione con strumenti già esistenti. Dalle sperimentazioni abbiamo confermato ciò che affermavamo in teoria, ovvero che tutti gli algoritmi proposti sono realmente efficienti.

Sviluppi futuri Gli algoritmi sono stati implementati in Java così come sono stati descritti in questa tesi e utilizzando le strutture dati che messe a disposizione dal linguaggio. L'ottimizzazione di tali algoritmi, utilizzando le ben note tecniche di programmazione dinamica, rappresenta, quindi, il prossimo passo del lavoro che verrà fatto nei prossimi mesi.

Un secondo sviluppo futuro consiste nell'aggiunta di un nuovo operatore alla classe dei tipi: l'intersezione. Lo studio relativo a questo nuovo operatore non è ancora concluso per entrambi i problemi. Allo stato attuale, soltanto l'algoritmo di validazione è stato leggermente modificato per supportare questo nuovo operatore, mentre l'algoritmo di inclusione necessita ancora di ulteriori studi.

Infine, un ulteriore sviluppo potrebbe rappresentare nella costruzione di un'applicazione, o integrazione in un'applicazione già esistente, degli algoritmi che sono stati implementati durante il periodo di tesi.

Appendice A

Schemi XML utilizzati

Di seguito riportiamo gli schemi utilizzati per la sperimentazione degli algoritmi di validazione.

A.1 Schema DBLP

```
<?xml version="1.0" encoding="UTF-8"?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema"
elementFormDefault="qualified">
  <xs:element name="author" type="xs:string"/>
  <xs:element name="cdrom" type="xs:string"/>
  <xs:element name="cite">
    <xs:complexType>
      <xs:simpleContent>
        <xs:extension base="xs:string">
          <xs:attribute name="label" type="xs:string"/>
        </xs:extension>
      </xs:simpleContent>
    </xs:complexType>
  </xs:element>
  <xs:element name="dblp">
    <xs:annotation>
      <xs:documentation>root</xs:documentation>
    </xs:annotation>
  </xs:element>
</xs:schema>
```

```
</xs:annotation>
<xs:complexType>
  <xs:sequence>
    <xs:element name="proceedings" minOccurs="0"
      maxOccurs="unbounded">
      <xs:complexType>
        <xs:sequence>
          <xs:element ref="editor" minOccurs="0"
            maxOccurs="unbounded"/>
          <xs:element ref="title"/>
          <xs:element ref="publisher"/>
          <xs:element ref="year"/>
          <xs:element ref="isbn" minOccurs="0"/>
          <xs:element ref="url" minOccurs="0"/>
        </xs:sequence>
        <xs:attribute name="key" type="xs:string"
          use="required"/>
      </xs:complexType>
    </xs:element>
    <xs:element name="inproceedings" minOccurs="0"
      maxOccurs="unbounded">
      <xs:complexType>
        <xs:sequence>
          <xs:element ref="author" minOccurs="0"
            maxOccurs="unbounded"/>
          <xs:element ref="title"/>
          <xs:element ref="pages"/>
          <xs:element name="booktitle"
            type="xs:string"/>
          <xs:element ref="url" minOccurs="0"/>
          <xs:element ref="cdrom" minOccurs="0"/>
          <xs:element ref="month"/>
          <xs:element ref="year"/>
        </xs:sequence>
      </xs:complexType>
    </xs:element>
  </xs:sequence>
</xs:complexType>
</xs:element>
```

```
        <xs:attribute name="key" type="xs:string"
            use="required"/>
    </xs:complexType>
</xs:element>
<xs:element name="article" minOccurs="0"
maxOccurs="unbounded">
    <xs:complexType>
        <xs:sequence>
            <xs:element ref="author" minOccurs="0"
                maxOccurs="unbounded"/>
            <xs:element ref="title"/>
            <xs:element ref="pages"/>
            <xs:element ref="cdrom" minOccurs="0"/>
            <xs:element ref="month" minOccurs="0"/>
            <xs:element ref="year"/>
            <xs:element name="volume"
                type="xs:string"/>
            <xs:element name="journal"
                type="xs:string"/>
            <xs:element ref="number"/>
            <xs:element ref="url" minOccurs="0"/>
        </xs:sequence>
        <xs:attribute name="key" type="xs:string"
            use="required"/>
    </xs:complexType>
</xs:element>
<xs:element name="book" minOccurs="0"
maxOccurs="unbounded">
    <xs:complexType>
        <xs:sequence>
            <xs:element ref="author" minOccurs="0"
                maxOccurs="unbounded"/>
            <xs:element ref="editor" minOccurs="0"
                maxOccurs="unbounded"/>
            <xs:element ref="title"/>
        </xs:sequence>
    </xs:complexType>
</xs:element>
```

```
<xs:element ref="publisher"/>
<xs:element ref="year"/>
<xs:element ref="isbn" minOccurs="0"/>
<xs:element ref="cdrom" minOccurs="0"/>
<xs:element ref="cite" minOccurs="0"
maxOccurs="unbounded"/>
<xs:element ref="url" minOccurs="0"/>
</xs:sequence>
<xs:attribute name="key" type="xs:string"
use="required"/>
</xs:complexType>
</xs:element>
<xs:element name="mastersthesis" minOccurs="0"
maxOccurs="unbounded">
  <xs:complexType>
    <xs:sequence>
      <xs:element ref="author"/>
      <xs:element ref="title"/>
      <xs:element ref="year"/>
      <xs:element ref="school"/>
    </xs:sequence>
    <xs:attribute name="key" type="xs:string"
use="required"/>
  </xs:complexType>
</xs:element>
<xs:element name="phdthesis" minOccurs="0"
maxOccurs="unbounded">
  <xs:complexType>
    <xs:sequence>
      <xs:element ref="author"/>
      <xs:element ref="title"/>
      <xs:element ref="year"/>
      <xs:element name="series"
type="xs:string" minOccurs="0"/>
      <xs:element ref="number" minOccurs="0"/>
    </xs:sequence>
  </xs:complexType>
</xs:element>
```

```
        <xs:element ref="month" minOccurs="0"/>
        <xs:element ref="school"/>
        <xs:element ref="publisher"
minOccurs="0"/>
        <xs:element ref="isbn" minOccurs="0"/>
    </xs:sequence>
    <xs:attribute name="key" type="xs:string"
use="required"/>
</xs:complexType>
</xs:element>
<xs:element name="www" minOccurs="0"
maxOccurs="unbounded">
    <xs:complexType>
        <xs:sequence>
            <xs:element ref="author" minOccurs="0"
maxOccurs="unbounded"/>
            <xs:element ref="title"/>
            <xs:element ref="year" minOccurs="0"/>
            <xs:element ref="url"/>
        </xs:sequence>
        <xs:attribute name="key" type="xs:string"
use="required"/>
    </xs:complexType>
</xs:element>
</xs:sequence>
</xs:complexType>
</xs:element>
<xs:element name="editor" type="xs:string"/>
<xs:element name="ee" type="xs:string"/>
<xs:element name="isbn" type="xs:string"/>
<xs:element name="month">
    <xs:simpleType>
        <xs:restriction base="xs:string">
            <xs:enumeration value="April"/>
            <xs:enumeration value="August"/>
```

```
<xs:enumeration value="December"/>
<xs:enumeration value="January"/>
<xs:enumeration value="July"/>
<xs:enumeration value="June"/>
<xs:enumeration value="March"/>
<xs:enumeration value="May"/>
<xs:enumeration value="November"/>
<xs:enumeration value="September"/>
</xs:restriction>
</xs:simpleType>
</xs:element>
<xs:element name="number">
  <xs:simpleType>
    <xs:restriction base="xs:string">
      <xs:enumeration value="1"/>
      <xs:enumeration value="11045"/>
      <xs:enumeration value="2"/>
      <xs:enumeration value="3"/>
      <xs:enumeration value="4"/>
      <xs:enumeration value="DS-96-3"/>
      <xs:enumeration value="LBL-32883"/>
    </xs:restriction>
  </xs:simpleType>
</xs:element>
<xs:element name="pages" type="xs:string"/>
<xs:element name="publisher">
  <xs:complexType>
    <xs:simpleContent>
      <xs:extension base="xs:string">
        <xs:attribute name="href" type="xs:string"/>
      </xs:extension>
    </xs:simpleContent>
  </xs:complexType>
</xs:element>
<xs:element name="school" type="xs:string"/>
```

```
<xs:element name="title" type="xs:string"/>
<xs:element name="url" type="xs:string"/>
<xs:element name="year" type="xs:string"/>
</xs:schema>
```

A.2 Schema expense-report

```
<?xml version="1.0" encoding="UTF-8"?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema"
  elementFormDefault="qualified"
  attributeFormDefault="unqualified">
  <xs:element name="expense-report">
    <xs:annotation>
      <xs:documentation>expense-report is the root
        element.</xs:documentation>
    </xs:annotation>
    <xs:complexType>
      <xs:sequence>
        <xs:element name="Person">
          <xs:complexType>
            <xs:sequence>
              <xs:element name="First" maxOccurs="3">
                <xs:simpleType>
                  <xs:restriction base="xs:string" >
                    <xs:minLength value="1"/>
                  </xs:restriction>
                </xs:simpleType>
              </xs:element>
              <xs:element name="Last" minOccurs="0">
                <xs:simpleType>
                  <xs:restriction base="xs:string">
                    <xs:minLength value="1"/>
                  </xs:restriction>
                </xs:simpleType>
              </xs:element>
            </xs:sequence>
          </xs:complexType>
        </xs:element>
      </xs:sequence>
    </xs:complexType>
  </xs:element>
</xs:schema>
```

```
<xs:element name="Title"
type="xs:string" maxOccurs="5"/>
<xs:element name="Phone">
  <xs:simpleType>
    <xs:restriction base="xs:string">
      <xs:pattern value="[0-9 \-]*"/>
    </xs:restriction>
  </xs:simpleType>
</xs:element>
<xs:element name="Email"
type="emailType" maxOccurs="unbounded"/>
</xs:sequence>
</xs:complexType>
</xs:element>
<xs:element ref="expense-item"
maxOccurs="unbounded"/>
</xs:sequence>
<xs:attribute name="detailed" type="xs:boolean"
use="required"/>
<xs:attribute name="currency" use="required">
  <xs:simpleType>
    <xs:restriction base="xs:string">
      <xs:enumeration value="USD"/>
      <xs:enumeration value="Euro"/>
      <xs:enumeration value="JPY"/>
    </xs:restriction>
  </xs:simpleType>
</xs:attribute>
<xs:attribute name="total-sum" type="xs:decimal"
use="required"/>
</xs:complexType>
</xs:element>
<xs:element name="expense" type="xs:decimal"/>
<xs:complexType name="TextType" mixed="true">
  <xs:choice minOccurs="0" maxOccurs="unbounded">
```

```
        <xs:element name="test" type="xs:string"/>
    </xs:choice>
</xs:complexType>
<xs:element name="description" type="TextType"/>
<xs:element name="Travel">
    <xs:complexType>
        <xs:sequence>
            <xs:element name="Destination"
                type="xs:string"/>
            <xs:element name="Mileage" type="xs:decimal"
                minOccurs="0"/>
        </xs:sequence>
        <xs:attribute name="means">
            <xs:simpleType>
                <xs:restriction base="xs:string">
                    <xs:enumeration value="Taxi"/>
                    <xs:enumeration value="CharterAir"/>
                    <xs:enumeration value="Airline"/>
                    <xs:enumeration value="Limo"/>
                    <xs:enumeration value="CharterSea"/>
                    <xs:enumeration value="Rail"/>
                    <xs:enumeration value="CharterLand"/>
                    <xs:enumeration value="Bus"/>
                </xs:restriction>
            </xs:simpleType>
        </xs:attribute>
    </xs:complexType>
</xs:element>
<xs:element name="Meal">
    <xs:complexType>
        <xs:sequence>
            <xs:element name="Name" type="xs:string"
                minOccurs="0"/>
            <xs:element ref="Location"/>
        </xs:sequence>
    </xs:complexType>
</xs:element>
```

```

    <xs:attribute name="mealtype">
      <xs:simpleType>
        <xs:restriction base="xs:string">
          <xs:enumeration value="dinner"/>
          <xs:enumeration value="breakfast"/>
          <xs:enumeration value="lunch"/>
          <xs:enumeration value="other"/>
        </xs:restriction>
      </xs:simpleType>
    </xs:attribute>
  </xs:complexType>
</xs:element>
<xs:element name="Parking">
  <xs:complexType>
    <xs:sequence>
      <xs:element ref="Location"/>
    </xs:sequence>
  </xs:complexType>
</xs:element>
<xs:element name="Date" type="xs:date"/>
<xs:simpleType name="emailType">
  <xs:restriction base="xs:string">
    <xs:pattern value="[\p{L}_-]+(\.[\p{L}_-]+)*
      @[\p{L}_-]+(\.[\p{L}_-]+)"/>
  </xs:restriction>
</xs:simpleType>
<xs:element name="expense-item">
  <xs:complexType>
    <xs:sequence>
      <xs:element ref="Date"/>
      <xs:element ref="expense"/>
      <xs:choice minOccurs="0" maxOccurs="unbounded">
        <xs:element ref="Meal"/>
        <xs:element name="Lodging">
          <xs:complexType>

```

```
<xs:sequence>
  <xs:element name="Name"
    type="xs:string" minOccurs="0"/>
  <xs:element ref="Location"/>
</xs:sequence>
</xs:complexType>
</xs:element>
<xs:element ref="Travel"/>
<xs:element ref="Parking"/>
<xs:element name="Entertainment">
  <xs:complexType>
    <xs:sequence>
      <xs:element name="Client-name"
        type="xs:string" minOccurs="0"
        maxOccurs="unbounded"/>
    </xs:sequence>
  </xs:complexType>
</xs:element>
<xs:element name="Misc">
  <xs:complexType>
    <xs:attribute name="misctype">
      <xs:simpleType>
        <xs:restriction base="xs:string">
          <xs:enumeration
            value="TeamBuilding"/>
          <xs:enumeration value="Tips"/>
          <xs:enumeration value="Fines"/>
          <xs:enumeration value="Rental"/>
          <xs:enumeration
            value="EverythingElse"/>
          <xs:enumeration value="Tolls"/>
          <xs:enumeration value="Telephone"/>
        </xs:restriction>
      </xs:simpleType>
    </xs:attribute>
```

```
        </xs:complexType>
    </xs:element>
</xs:choice>
</xs:sequence>
<xs:attribute name="type" use="required">
    <xs:simpleType>
        <xs:restriction base="xs:string">
            <xs:enumeration value="Meal"/>
            <xs:enumeration value="Lodging"/>
            <xs:enumeration value="Travel"/>
            <xs:enumeration value="Parking"/>
            <xs:enumeration value="Entertainment"/>
            <xs:enumeration value="Misc"/>
        </xs:restriction>
    </xs:simpleType>
</xs:attribute>
<xs:attribute name="expto" use="required">
    <xs:simpleType>
        <xs:restriction base="xs:string">
            <xs:enumeration value="Development"/>
            <xs:enumeration value="Marketing"/>
            <xs:enumeration value="Accounting"/>
            <xs:enumeration value="Sales"/>
            <xs:enumeration value="Operations"/>
            <xs:enumeration value="Support"/>
            <xs:enumeration value="IT"/>
        </xs:restriction>
    </xs:simpleType>
</xs:attribute>
</xs:complexType>
</xs:element>
<xs:element name="Location" type="xs:string"/>
<xs:element name="strong" type="TextType"/>
<xs:element name="italic" type="TextType"/>
</xs:schema>
```

Bibliografia

- [AM00] Bender Michael A. and Farach-Colton Martin. The LCA Problem Revisited. In *Latin American Theoretical Informatics*, pages 88–94, 2000.
- [AYCLS01] Sihem Amer-Yahia, SungRan Cho, Laks V. S. Lakshmanan, and Divesh Srivastava. Minimization of tree pattern queries. In *SIGMOD Conference*, pages 497–508, 2001.
- [AYG04] Sihem Amer-Yahia and Luis Gravano, editors. *Proceedings of the Seventh International Workshop on the Web and Databases, WebDB 2004, June 17-18, 2004, Maison de la Chimie, Paris, France, Colocated with ACM SIGMOD/PODS 2004*, 2004.
- [BB79] Catriel Beeri and Philip A. Bernstein. Computational problems related to the design of normal form relational schemas. *ACM Trans. Database Syst.*, 4(1):30–59, 1979.
- [BGMP07a] Antonia Bertolino, Jinghua Gao, Eda Marchetti, and Andrea Polini. Systematic Generation of XML Instances to Test Complex Software Applications. 2007.
- [BGMP07b] Antonia Bertolino, Jinghua Gao, Eda Marchetti, and Andrea Polini. TAXI - A Tool for XML-Based Testing. 2007.
- [BLS06] Denilson Barbosa, Gregory Leighton, and Andrew Smith. Efficient incremental validation of XML documents after composite updates. In Sihem Amer-Yahia, Zohra Bellahsene, Ela Hunt, Rainer Unland,

- and Jeffrey Xu Yu, editors, *XSym*, volume 4156 of *Lecture Notes in Computer Science*, pages 107–121. Springer, 2006.
- [BM04] Paul V. Biron and Ashok Malhotra. XML Schema Part 2: Datatypes Second Edition. Technical report, World Wide Web Consortium, Oct 2004. W3C Recommendation.
- [BML⁺04] Denilson Barbosa, Alberto O. Mendelzon, Leonid Libkin, Laurent Mignet, and Marcelo Arenas. Efficient incremental validation of XML documents. In *ICDE [DBL04]*, pages 671–682.
- [BNdB04] Geert Jan Bex, Frank Neven, and Jan Van den Bussche. DTDs versus XML Schema: A Practical Study, 2004.
- [BNST06] Geert Jan Bex, Frank Neven, Thomas Schwentick, and Karl Tuyls. Inference of concise DTDs from XML data. In Dayal et al. [DWL⁺06], pages 115–126.
- [BNV07] Geert Jan Bex, Frank Neven, and Stijn Vansummeren. Inferring XML schema definitions from XML data. In *VLDB*, pages 998–1009, 2007.
- [Brz64] J.A. Brzozowski. Derivates of regular expression. *Journal of the ACM*, 11:481–494, 1964.
- [Cho02] Byron Choi. What are real DTDs like? In *WebDB*, pages 43–48, 2002.
- [CLM02] Diego Calvanese, Maurizio Lenzerini, and Rajeev Motwani, editors. *Database Theory - ICDT 2003, 9th International Conference, Siena, Italy, January 8-10, 2003, Proceedings*, volume 2572 of *Lecture Notes in Computer Science*. Springer, 2002.
- [CS07] Dario Colazzo and Carlo Sartiani. Efficient subtyping for unordered XML types. Technical report, Dipartimento di Informatica - Università di Pisa, 2007.

- [DBL04] *Proceedings of the 20th International Conference on Data Engineering, ICDE 2004, 30 March - 2 April 2004, Boston, MA, USA*. IEEE Computer Society, 2004.
- [DWL⁺06] Umeshwar Dayal, Kyu-Young Whang, David B. Lomet, Gustavo Alonso, Guy M. Lohman, Martin L. Kersten, Sang Kyun Cha, and Young-Kuk Kim, editors. *Proceedings of the 32nd International Conference on Very Large Data Bases, Seoul, Korea, September 12-15, 2006*. ACM, 2006.
- [FKK04] Jirí Fiala, Václav Koubek, and Jan Kratochvíl, editors. *Mathematical Foundations of Computer Science 2004, 29th International Symposium, MFCS 2004, Prague, Czech Republic, August 22-27, 2004, Proceedings*, volume 3153 of *Lecture Notes in Computer Science*. Springer, 2004.
- [FPS] J. Nathan Foster, Benjamin C. Pierce, and Alan Schmitt. A Logic Your Typechecker Can Count On: Unordered Tree Types in Practice, booktitle = PLAN-X, year = 2007, pages = 80-90, ee = <http://www.plan-x-2007.org/plan-x-2007.pdf>, crossref = DBLP:conf/planX/2007, bibsource = DBLP, <http://dblp.uni-trier.de>.
- [GCS07] Giorgio Ghelli, Dario Colazzo, and Carlo Sartiani. Efficient Inclusion for a Class of XML Types with Interleaving and Counting. In *DBPL*, 2007.
- [GCS08] Giorgio Ghelli, Dario Colazzo, and Carlo Sartiani. Linear Time Membership for a Class of XML Types with Interleaving and Counting. In *PLAN-X*, 2008.
- [GMN07] Wouter Gelade, Wim Martens, and Frank Neven. Optimizing Schema Languages for XML: Numerical Constraints and Interleaving. 2007.
- [GN07] Wouter Gelade and Frank Neven. Succinctness of Pattern-Based Schema Languages for XML. 2007.

- [JJ79] J.E. Hopcroft and J.D. Ullman. *Introduction to Automata Theory, Languages and Computation*. Addison-Wesley, 1979.
- [MLMK05] Makoto Murata, Dongwon Lee, Murali Mani, and Kohsuke Kawaguchi. Taxonomy of XML schema languages using formal language theory. *ACM Trans. Internet Techn.*, 5(4):660–704, 2005.
- [MNS04] Wim Martens, Frank Neven, and Thomas Schwentick. Complexity of decision problems for simple regular expressions. In Fiala et al. [FKK04], pages 889–900.
- [MNS07] Wim Martens, Frank Neven, and Thomas Schwentick. Simple off the shelf abstractions for XML schema. *SIGMOD Record*, 36(3):15–22, 2007.
- [MNSB06] Wim Martens, Frank Neven, Thomas Schwentick, and Geert Jan Bex. Expressiveness and Complexity of XML Schema. 2006.
- [MS94] Alain J. Mayer and Larry J. Stockmeyer. The Complexity of Word Problems - This Time with Interleaving. *Information and Computation*, 115:293–311, 1994.
- [MWM07] Manizheh Montazerian, Peter T. Wood, and Seyed R. Mousavi. XPath query satisfiability is in PTIME for real-world DTDs. In Denilson Barbosa, Angela Bonifati, Zohra Bellahsène, Ela Hunt, and Rainer Unland, editors, *XSym*, volume 4704 of *Lecture Notes in Computer Science*, pages 17–30. Springer, 2007.
- [SD03] Dal Zilio Silvano and Lugiez Denis. XML Schema, Tree Logic and Sheaves Automata. In *RTA 2003 – 14th International Conference on Rewriting Techniques and Applications*, volume 2706 of *Lecture Notes in Computer Science*, pages 246–263. Springer-Verlag, June 2003.
- [SW03] Jérôme Siméon and Philip Wadler. The essence of XML. In *POPL '03: Proceedings of the 30th ACM SIGPLAN-SIGACT symposium*

on Principles of programming languages, pages 1–13, New York, NY, USA, 2003. ACM.

- [SWK⁺01] A. R. Schmidt, F. Waas, M. L. Kersten, D. Florescu, I. Manolescu, M. J. Carey, and R. Busse. The XML Benchmark Project. Technical Report INS-R0103, CWI, Amsterdam, The Netherlands, April 2001.
- [TBMM04] Henry S. Thompson, David Beech, Murray Maloney, and Noah Mendelsohn. XML Schema Part 1: Structures Second Edition. Technical report, World Wide Web Consortium, Oct 2004. W3C Recommendation.
- [Woo03] Peter T. Wood. Containment for xpath fragments under DTD constraints. In Calvanese et al. [CLM02], pages 300–314.
- [ZLM04] Silvano Dal Zilio, Denis Lugiez, and Charles Meyssonnier. A Logic You Can Count On. *SIGPLAN Not.*, 39(1):135–146, 2004.