Università degli Studi di Pisa

FACOLTÀ DI SCIENZE MATEMATICHE, FISICHE E NATURALI
Corso di Laurea triennale in Informatica L-31

Linguaggi su alfabeti infiniti

Candidato:
Michele Colombo

Relatore:
Pierpaolo Degano

Anno Accademico 2012–2013

Linguaggi su alfabeti infiniti

Michele Colombo

06 dicembre 2013

Indice

Capitolo 1

Introduzione

Nel presente elaborato andrò ad illustrare alcuni modelli per linguaggi formali generati a partire da alfabeti infiniti, per la comprensione di ciò è utile una conoscenza di base di teoria degli automi e dei linguaggi[1]; prima di introdurre i modelli però una breve panoramica atta a motivarne lo studio.

1.1 Necessità di linguaggi su alfabeti infiniti

Trovare modelli per trattare linguaggi con parole che spaziano su infiniti simboli è quantomai problematico ed ancor più difficile è trovare modelli che siano una base valida da cui ricavare strumenti con procedimenti effettivi, calcolabili e di complessità accettabile. Nonostante la difficoltà però ci sono forti motivazioni che spingono a compiere sforzi in tale direzione, sono molte le situazioni in cui si ha a che fare con queste tipologie di linguaggi, per lo più in problemi di *verifica formale* e *database*; alcuni esempi:

Verifica formale Dato un sistema per il quale si vuole verificare formalmente una qualche proprietà è comune che questo contenga una fonte di non finitezza[2] e che in un approccio basato su linguaggi formali questa si rifletta nell'alfabeto. Ad esempio appurare se una certa implementazione è conforme ad una determinata specifica si può riconduce ad un problema di contenimento tra linguaggi. Dato un automa A che formalizza il comportamento dell'implementazione e dato un automa B che formalizza il comportamento definito dalla specifica: l'implementazione è conforme alla specifica se e solo se $L(A) \subseteq L(B)$.

Analisi statica La verifica statica di certe proprietà di un programma può essere ricondotta ad un problema su linguaggi e dato che generalmente i valori in un programma spaziano su domini infiniti si ha un alfabeto infinito.
Basti pensare ad esempio ad un programma con delle funzioni a parametri interi che si chiamano ricorsivamente tra di loro, una sequenza specifica di chiamate si può rappresentare come una parola, e.g. $f_1(2)f_1(1)f_2(1)f_2(0)$. Sia $\mathcal{L}_1$ il linguaggio con tutte le possibili sequenze di chiamate effettuabili

[1] Un qualsiasi testo introduttivo è sufficiente, ad esempio [1].

[2] Dati con domini illimitati, tempo, ricorsione, componenti e/o risorse gestite dinamicamente e quindi potenzialmente illimitate, ...

dal programma e sia $\mathcal{L}_2$ il linguaggio con tutte le sequenze che violano una qualche specifica ψ. Dimostrare che il programma soddisfa ψ si riconduce quindi a dimostrare che $\mathcal{L}_1 \cap \mathcal{L}_2 = \emptyset$. [3]
Attualmente salvo in casi particolari, come in [5], si usano formalismi che limitano forzatamente i domini delle variabili per mantenere la decidibilità.

Trace log Controllare se un esecuzione di un programma rispetta determinate proprietà si può ricondurre al verificare se la traccia appartiene ad un certo linguaggio; nel caso, molto comune, che in tale traccia compaiano valori appartenenti a domini illimitati l'alfabeto del linguaggio è infinito. Ad esempio in [8] viene presentato un metodo per verificare, durante l'esecuzione del programma stesso, proprietà che coinvolgono risorse generate in numero potenzialmente illimitato, in particolare viene preso in considerazione il caso in cui queste risorse siano oggetti Java allocati.

MSC In un contesto di comunicazione interprocesso [7] i simboli dell'alfabeto rappresentano le singole *send e receive* dei processi, i quali sono in numero illimitato, mentre una parola rappresenta un Message Sequence Charts e descrive il comportamento della rete in un arco di tempo.

Navigazione web In [6] viene usato un automa a memoria finita - vedi cap.2 - per modellare un browser; l'alfabeto consiste in indirizzi di pagine web e una parola rappresenta un percorso di navigazione risultante da una sequenza finita di click.

Per altri esempi si veda la sezione 4.2.

Domini finiti Ci sono anche situazioni in cui un alfabeto pur non essendo infinito viene considerato tale, ad esempio perché eccessivamente grande, perché troppo complesso da determinare a priori o per avere una rappresentazione più compatta del modello.

1.2 Organizzazione dei contenuti

Nei prossimi due capitoli verranno presentati i due principali modelli basati su automi per il riconoscimento di linguaggi su alfabeti infiniti, nel capitolo 2 gli automi a memoria finita - *Finite Memory Automata* - uno dei primi modelli comparsi in letteratura che ha introdotto l'approccio basato su registri, nel capitolo 3 gli automi con un numero finito di variabili - *Variable Finite Automata* - di più recente comparsa; successivamente nel capitolo 4 verrà aperta una parentesi su una classe di linguaggi su alfabeti infiniti di particolare interesse i *data word languages*; in fine nell'ultimo capitolo verranno messi brevemente a confronto i due modelli presentati e sarà fatta una panoramica più ampia su altri modelli e risultati presenti in letteratura.
In generale non saranno riportate dimostrazioni formali le quali sono facilmente reperibili nella bibliografia, piuttosto si cercherà di fornire motivazioni intuitive per facilitare la comprensione.

[3] In realtà la dimostrazione fa riferimento ad $\mathcal{L}_1$ e non direttamente al programma; la correttezza della dimostrazione dipende quindi dalla correttezza della formalizzazione.

Capitolo 2

Automi a memoria finita

Gli automi a memoria finita, introdotti in [9], generalizzano il classico automa a stati finiti di Rabin-Scott [4] estendendo la classe di linguaggi riconosciuti con il naturale analogo dei regolari su alfabeti infiniti[1], i linguaggi *quasi-regular*.

Intuizione Un automa a memoria finita riesce a trattare linguaggi con infiniti simboli sfruttando una serie limitata di registri; ogni nuovo simbolo in input viene confrontato con i valori nei registri, in base a ciò e allo stato corrente viene deciso se salvare in qualche registro l'input e/o se fare una transizione di stato.

2.1 Definizione

Un automa a memoria finita, abbreviato con FMA, è una sestupla:

$$\mathcal{F} =< S, s_0, u, \rho, \mu, F >$$

- S un insieme finito, rappresentante gli stati dell'automa;
- $s_0 \in S$ lo stato iniziale;
- $u = \omega_1\omega_2 \ldots \omega_r \in (\Sigma \cup \{\#\})^r$ l'assegnamento iniziale degli r registri;
- $\rho : S \to \{1, 2, \ldots, r\}$ la funzione di riassegnamento definisce se[2] e dove salvare il simbolo in ingresso;
- $\mu \subseteq S \times \{1, 2, \ldots, r\} \times S$ la relazione di transizione definisce in che stati[3] deve transire l'automa in funzione dello stato corrente e del confronto del simbolo in ingresso con quelli memorizzati nei registri;
- $F \subseteq S$ il sottoinsieme degli stati finali.

[1] Non c'è una definizione formale di *naturale analogo*, è un concetto più che altro intuitivo.

[2] ρ è una funzione parziale, negli stati in cui è indefinita il simbolo in ingresso non viene salvato in alcun registro.

[3] Il modello generale qui introdotto è non deterministico.

Assegnamento Un assegnamento è una parola $\omega_1\omega_2\ldots\omega_r \in (\Sigma \cup \{\#\})^r$ in cui il simbolo ω_i rappresenta il contenuto dell'i-esimo registro; il simbolo speciale $\# \notin \Sigma$ viene usato per indicare che un registro è vuoto. L'insieme di tutti i simboli ω_i rappresentante il contenuto dei registri è denotato da $[\omega]$; in questa formalizzazione due registri non possono contenere lo stesso simbolo, con l'eccezione del simbolo vuoto, questo non inficia però la capacità espressiva del modello - vedi 2.6.1.

Computazione Se nello stato s l'automa riceve in input un simbolo σ_i la computazione procede in due fasi:

1. Riassegnamento: se $\rho(s)$ è definita e $\sigma_i \notin [\omega]$ salva il nuovo simbolo nel registro $\rho(s)$;

2. Transizione: se il simbolo in ingresso è contenuto nel k-esimo registro $\sigma_i = \omega_k$ ed è definita la transizione $(s, k, t) \in \mu$ allora l'automa transita nello stato t.

Le due fasi vengono eseguite in serie, è quindi legittimo che nella fase di riassegnamento il simbolo in ingresso venga salvato in un registro e nella seconda fase sia lo stesso registro a far scattare la transizione.

Configurazioni L'insieme S degli stati dell'automa è di cardinalità finita, però lo stato effettivo dell'automa non dipende solo da S, ma anche dal valore contenuto nei registri e dato che questi variano su un alfabeto infinito l'automa non è a stati finiti. Chiameremo *configurazione* dell'automa il suo stato effettivo (ω, s), dove $s \in S$ è lo stato nominale e $\omega = \omega_1\omega_2\ldots\omega_r \in (\Sigma \cup \{\#\})^r$ è l'assegnamento corrente.
Estendendo col concetto di configurazione anche gli altri elementi caratterizzanti l'automa otteniamo:

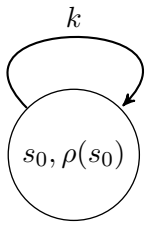
- S^c è l'insieme delle possibili configurazioni;
- s_0^c la configurazione iniziale;
- F^c l'insieme delle configurazioni finali;
- $\mu^c \subseteq S^c \times \Sigma \times S^c$ la relazione di transizione tra configurazioni indotta da μ in accordo con la definizione di computazione.

Essendoci infiniti stati effettivi non esiste una stringa sufficientemente lunga da garantire che l'automa attraversi una stessa configurazione più volte, il pumping lemma non si applica quindi ai linguaggi quasi-regular.

Linguaggio Come per i classici automi non deterministici il linguaggio riconosciuto da un automa a memoria finita è composto da tutte quelle stringhe per cui esiste una computazione terminante in uno stato finale.

Rappresentazione grafica Un automa a memoria finita è completamente definito dall'assegnamento iniziale e da un grafo diretto: un nodo è etichettato con lo stato che rappresenta e, se definito, con il valore della funzione di riassegnamento; un arco tra due nodi i, j rappresenta invece una transizione di stato ed è etichettato

con i registri $\{k : (i, k, j) \in \mu\}$ che se uguali al simbolo in ingresso provocano la transizione.



Assegnamento	
1	2
#	#

2.2 Potere espressivo

Pur avendo infiniti stati effettivi il potere espressivo di un automa a memoria finita è fortemente limitato[4] mediante i vincoli imposti sulla sua relazione di transizione tra configurazioni:

Siano $(\omega_1\omega_2 \ldots \omega_r,\ s)$, $(v_1v_2 \ldots v_r,\ t)$ due configurazioni e sia $\sigma \in \Sigma$ un simbolo dell'alfabeto, $((\omega,\ s),\ \sigma,\ (v,\ t)) \in \mu^c$ se e solo se:

- se $\sigma = \omega_j \in [\omega]$ allora $v = \omega$ e $(s, j, t) \in \mu$, il contenuto dei registri non varia in quanto il simbolo in ingresso è già memorizzato e la funzione di riassegnamento non si applica;
- se invece $\sigma \notin [\omega]$ la funzione di riassegnamento $\rho(s)$ è definita e l'unico valore per cui i registri differiscono è $v_{\rho(s)} = \sigma \neq \omega_{\rho(s)}$ e $(s, \rho(s), t) \in \mu$.
  Se $\rho(s)$ non fosse definita, nella fase di transizione il simbolo in ingresso non verrebbe trovato in alcun registro e per definizione non ci potrebbe essere alcuna transizione di stato.

Intuitivamente l'automa non può manipolare liberamente i registri, le uniche interazioni permesse sono quelle in accordo con una sorta di *meccanismo di sostituzione*: ad ogni passo è possibile confrontare il simbolo in ingresso con il valore nei registri ed eventualmente, se non già presente, sostituirlo a uno di questi.
Questo meccanismo vuole essere l'analogo del *test di eguaglianza* dell'automa classico con il quale il simbolo in ingresso viene confrontato con un qualche valore cablato nella relazione di transizione, approccio che però non è applicabile con alfabeti infiniti in quanto la relazione di transizione deve poter essere denotata in maniera finita e quindi può contenere al suo interno solo un sottoinsieme finito dell'alfabeto.
La non finitezza delle configurazioni quindi non influenza troppo il potere espressivo dell'automa, di fatto l'effettiva computazione avviene negli stati in S che sono finiti e non nei registri che servono solo per interfacciarsi alla non finitezza dell'alfabeto.

2.3 Relazioni con i linguaggi regolari

Gli automi a memoria finita, come altri modelli, si pongono come naturale analogo su alfabeti infiniti degli automi classici; non essendoci però alcuna definizione formale di

[4] Un automa con infiniti stati senza alcuna restrizione è in grado di riconoscere *qualsiasi* linguaggio, vedi [3].

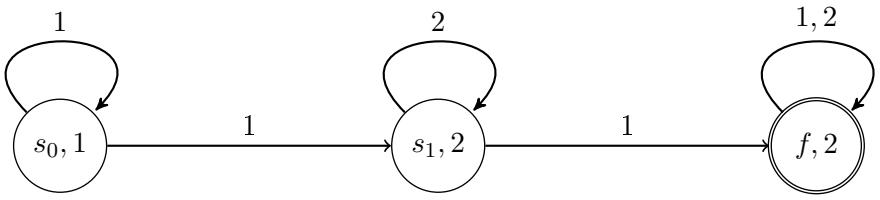
naturale analogo sta al gusto soggettivo decretare quale sia il modello più naturale. A prescindere da ciò c'è una stretta relazione tra linguaggi quasi-regular e linguaggi regolari:

- Come visto nella sezione precedente c'è un'analogia molto forte tra il meccanismo di sostituzione degli automi a memoria finita e il test di eguaglianza dell'automa classico;
- Preso un qualsiasi automa a memoria finita $\mathcal{F}$ e un sottoinsieme finito Σ^I dell'alfabeto Σ il linguaggio $L(\mathcal{F}) \cap \Sigma^{I*}$ è un linguaggio regolare su Σ^I;
- Dato un linguaggio regolare con relativo automa $\mathcal{A} =< S, s_0, \Sigma^I, \mu_\mathcal{A}, F >$ è possibile costruire un automa a memoria finita $\mathcal{F} =< S, s_0, u, \rho, \mu_\mathcal{F}, F >$ del tutto equivalente, tale che $L(\mathcal{A}) = L(\mathcal{F})$:
  - S, s_0 ed F coincidono;
  - Sia $\Sigma^I = \{\sigma_1, \sigma_2, \ldots, \sigma_r\}$ allora $u = \sigma_1\sigma_2 \ldots \sigma_r$, la memoria dell'automa è limitata ma anche l'alfabeto lo è, quindi possiamo iniettare contemporaneamente tutti i simboli dell'alfabeto nei registri durante fase di inizializzazione;
  - ρ è ovunque indefinita, non ci serve una memoria che cambi dinamicamente in quanto conosciamo già tutti i simboli;
  - Per ogni transizione $(s, \sigma_i, t) \in \mu_\mathcal{A}$ inseriamo una transizione (s, i, t) in $\mu_\mathcal{F}$.

  Intuitivamente invece che avere i simboli dell'alfabeto direttamente cablati nella relazione di transizione questi vengono spostati nei registri e dereferenziati mediante l'indice.

2.4 Esempi

Esempio 1



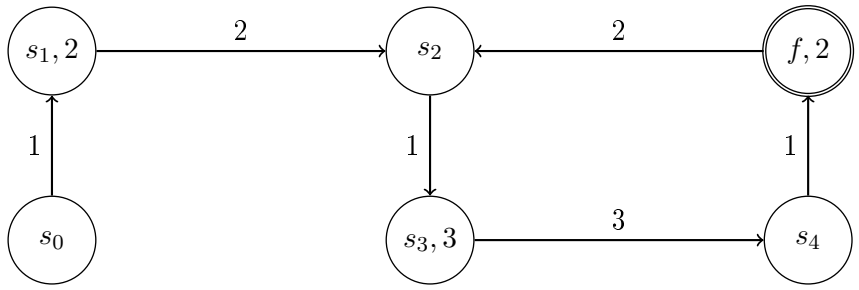
Assegnamento	
1	2
#	#

Il linguaggio dell'automa così definito consiste in tutte le stringhe con qualche simbolo che si ripete almeno due volte. Il non determinismo del primo nodo permette di considerare tutti i simboli in ingresso come candidati alla ripetizione, mentre il secondo nodo semplicemente aspetta di rivedere il simbolo scelto non deterministicamente

e salvato nel primo registro al passo precedente; formalmente:

$$L_1 = \{\sigma_1\sigma_2 \dots \sigma_n \in \Sigma^* : \exists i, j.\ i \neq j,\ \sigma_i = \sigma_j\}$$

Esempio 2

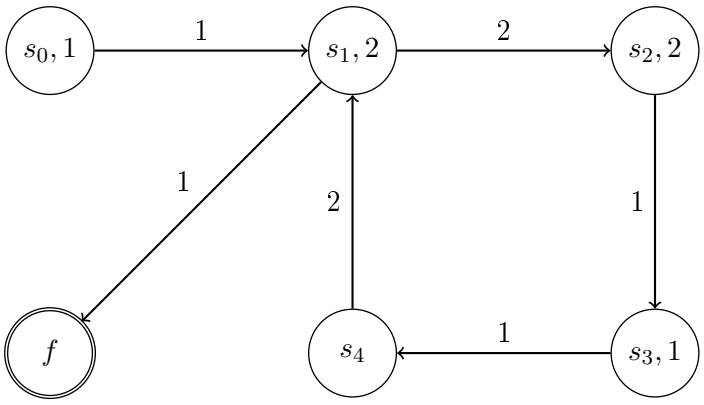


Assegnamento		
1	2	3
a	#	#

Questo esempio mostra com'è possibile legare parti delle stringhe del linguaggio a dei simboli ben precisi mediante l'assegnamento iniziale. L'automa riconosce tutte le parole delimitate dal simbolo a, al cui interno si alternano a e simboli *non* a tali da essere diversi dal simbolo *non* a successivo; formalmente:

$$L_2 = \{a\sigma_1 a\sigma_2 a \dots \sigma_{2n} a \in \Sigma^* : \forall i \in [2, 2n].\ a \neq \sigma_{i-1} \neq \sigma_i \neq a,\ n \geq 1\}$$

Esempio 3



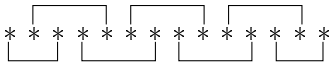
Assegnamento	
1	2
#	#

Consideriamo solo una sottoclasse del linguaggio accettato dall'automa in quanto è più semplice da denotare ed è comunque sufficiente al fine dell'esempio, ovvero mostrare che il *pumping lemma* non sempre si applica ai linguaggi quasi-regular.

Siano $\tau_0, \tau_1, \dots, \tau_{2n}$ elementi tutti distinti di Σ, fanno parte del linguaggio accettato dall'automa le stringhe $\sigma_1\sigma_2 \dots \sigma_{4n+2}$ tali che:

- $n \geq 1$;
- $\sigma_1 = \sigma_3 = \tau_0$;
- $\sigma_{4n} = \sigma_{4n+2} = \tau_{2n}$;
- $\forall i \in \{1, 2, \ldots, 2n-1\}\ \sigma_{2i} = \sigma_{2i+3} = \tau_i$.

Le relazioni tra i simboli di una stringa così definita sono chiarificate dallo schema seguente dove gli $*$ rappresentano generici simboli e i collegamenti rappresentano la relazione di uguaglianza tra due simboli.



Se, per assurdo, il pumping lemma si applicasse potremo scomporre una stringa in tre parti xyz con y non vuota e tale che per $i \geq 1$ anche la stringa xy^iz appartenga al linguaggio.
Dal diagramma dell'automa si evince che qualsiasi stringa accettata ha lunghezza $4k+2$, a prescindere da quante volte venga ripetuta y la lunghezza della stringa risultante deve sempre verificare tale condizione, quindi $|y| = 4l$ per $l \geq 1$.
Una volta letto il prefisso xy, comune a tutte le stringhe ottenute col lemma, l'automa si può trovare in uno qualsiasi dei quattro stati del ciclo quando comincia a scorrere la y successiva[5]:

- Se si trova in s_1 o s_3 salva il simbolo y_1 e passa allo stato successivo dal quale può uscire solo se il prossimo simbolo, y_2, è uguale al simbolo salvato tre passi prima, ovvero y_{4l-1};
- Se si trova in s_2 o s_4 può andare avanti solo se il prossimo simbolo, y_1, è uguale a quello salvato tre passi prima ovvero y_{4l-2}.

In entrambi i casi si ha un assurdo, per costruzione tutti i simboli di y sono distinti: se $l = 1$ i simboli non coincidono $y_2 \neq y_3,\ y_1 \neq y_2$, se $l \geq 2$ i primi tre simboli sono per costruzione diversi dagli ultimi tre.

2.4.1 Linguaggi non esprimibili

Oltre a riportare esempi che illustrano le potenzialità espressive degli automi a memoria finita guardiamo alcuni linguaggi che invece non sono riconoscibili da alcuno:

- Il linguaggio le cui stringhe hanno l'ultimo simbolo diverso da tutti quelli precedenti non può essere riconosciuto in quanto sarebbe necessario un numero illimitato di registri per salvare tutti i simboli da confrontare poi con l'ultimo.

[5] Essendo $|y| \geq 4$ e potendo aumentare i a piacere non vengono considerati i casi che contemplano stato iniziale e finale.

2.5 Proprietà

2.5.1 Indistinguibilità

Il modello tratta infiniti simboli, ma la sua capacità di osservazione è finita: consente di denotare un numero finito di simboli mediante l'assegnamento iniziale e durante la computazione in ogni istante può ricordare al più r simboli distinti.
Un nuovo simbolo è sempre trattato allo stesso modo, a prescindere da che simbolo effettivamente sia; di fatto l'automa è in grado di distinguere al più $r + 1$ simboli, tutti quelli nei registri ed il simbolo *nuovo*. Questa *indistinguibilità* ha importanti conseguenze sul linguaggio riconosciuto e viene formalizzata e motivata con la chiusura su automorfismi - vedi sezione successiva.

2.5.2 Chiusure

Chiusura su automorfismi

Sia $\mathcal{F}$ un FMA con assegnamento iniziale u e sia I è l'insieme di tutti gli automorfismi: $\Sigma \to \Sigma$ che sono identità su $[u]$.
Una parola $\sigma \in \Sigma^*$ è accettata dall'automa se e solo se $\forall \iota \in I : \iota(\sigma) \in L(\mathcal{F})$.

Un automa a memoria finita quindi *riconosce pattern di simboli* e l'assegnamento iniziale è l'unico strumento che abbiamo per legare a qualche simbolo preciso le variabili in questi pattern.

Altre chiusure

La classe dei linguaggi quasi-regular è chiusa su unione, intersezione, concatenazione e rispetto alla stella di Kleene; non lo è invece rispetto al complemento e al reversing.

2.5.3 Calcolabilità

Emptiness

Grazie all'indistinguibilità caratteristica dell'automa è sufficiente considerare un sottoinsieme finito dell'alfabeto per verificare se il linguaggio sia vuoto, così facendo ci si riconduce ad un linguaggio regolare - vedi 2.3 - per il quale è decidibile.

$$L(\mathcal{F}) = \emptyset \Leftrightarrow L(\mathcal{F}) \cap \Sigma^{I*} = \emptyset$$

Dove $\Sigma^I \subset \Sigma$ è un insieme finito di esattamente r simboli distinti tra cui quelli non vuoti dell'assegnamento iniziale. Questo è sufficiente in quanto per l'indistinguibilità se c'è una computazione dell'automa che accetta una stringa σ di lunghezza n esiste una computazione che accetta una stringa υ di pari lunghezza ma con al più solo r simboli distinti.
Sia C_σ una computazione su σ che porta l'automa in uno stato finale, υ può essere costruita ad hoc affinché C_υ passi esattamente dagli stessi stati. Ogni volta che c'è una transizione causata dal simbolo σ_i ci sono due possibilità:

- σ_i viene salvato nel registro k che poi fa scattare la transizione, se lo stesso registro al passo $i-1$ di C_υ è non vuoto υ_i assumerà il valore presente nel registro, altrimenti prenderà come valore uno qualsiasi degli r che ancora non sono stati memorizzati.

- σ_i è presente nel registro k il quale fa scattare la transizione, υ_i assumerà il valore presente nel registro k in C_υ al passo $i-1$. Il fatto che il registro k nella computazione di υ non sia vuoto è garantito dal punto precedente.

Non-emptiness

Controllare che il linguaggio accettato contenga almeno una stringa diversa da ϵ, la stringa vuota, è decidibile [11] ed appartiene alla classe di complessità[6] NP.

Appartenenza

Data una stringa per decidere se questa appartenga o no al linguaggio occorre simulare un esecuzione dell'automa, essendo questo potenzialmente non deterministico il problema appartiene ad NP [11].

Universalità e inclusione [13]

Non è decidibile se un automa a memoria finita sia universale ovvero accetti tutte le stringhe in Σ^*. Dati due automi a memoria finita non si può verificare se il linguaggio accettato da uno includa quello accettato dall'altro o se siano equivalenti[7].

2.6 Varianti

2.6.1 Multi-assegnamento

I M-FMA permettono che i valori nei registri possano essere non distinti;
sia $\mathcal{R} = \mathcal{P}\{1, 2, \ldots, r\} \setminus \emptyset$, le componenti dell'automa che sono influenzate da questa estensione sono:

- $\rho : S \to \mathcal{R}$ diviene la funzione di M-assegnamento, definisce in quali registri copiare il nuovo simbolo a prescindere se sia già in qualche registro;

- $\mu \subseteq S \times \mathcal{R} \times S$ analogamente la relazione di M-transizione farà il test d'eguaglianza con un sottoinsieme dei registri, non più uno solo.

L'estensione così fatta è puramente sintattica, permette di ottenere un modello più maneggevole che in certi casi rende ragionamenti e dimostrazioni più agevoli ma non va a modificare il potere espressivo, la classe dei linguaggi riconoscibili è la stessa.

[6] Classe di problemi decidibili in tempo non deterministico polinomiale.
[7] L'unico caso in cui si riesce è se l'automa più espressivo ha solo due registri[9].

2.6.2 Deterministico

Definizione Un automa a memoria finita può essere limitato imponendo vincoli di determinismo:

- La funzione di riassegnamento ρ dev'essere totale, definita su tutti gli stati.
- La relazione di transizione deve essere una funzione ovunque definita, per ogni coppia stato-registro s, k deve esistere esattamente uno stato t tale che $(s, k, t) \in \mu$.

La classe di linguaggi riconoscibili da un automa a memoria finita deterministico è un sottoinsieme stretto dei linguaggi quasi-regular, i vincoli di determinismo restringono l'espressività del modello. Come per la versione non deterministica il modello può essere esteso sintatticamente introducendo gli assegnamenti multipli.

Proprietà

- **Chiusure** I linguaggi quasi-regular deterministici sono chiusi su tutte le operazioni booleane ma non rispetto al reversing, alla stella di Kleene e alla concatenazione.
- **Decidibilità** Rispetto ad universalità ed inclusione non è stato possibile trovare alcun risultato in letteratura, anche riguardo al problema di determinare se il linguaggio accettato dall'automa sia vuoto non sono stati trovati risultati, essendo però decidibile nel modello non deterministico lo è anche nella versione più vincolata. In [11] viene dimostrata la decidibilità del problema dell'appartenenza di una stringa al linguaggio e del determinare se il linguaggio contenga almeno una stringa diversa dalla stringa vuota, si dimostrano rispettivamente appartenenti alle classi di complessità P e NP.
- **Automa minimale** In [12] viene fornita una caratterizzazione analoga al teorema di Myhill-Nerode anche per i linguaggi quasi-regular deterministici, successivamente in [14] viene approfondito il discorso per una classe di automi leggermente più generale andando a definire una rappresentazione canonica dell'automa che si dimostra minimale nel numero di stati e registri, inoltre viene anche fornito un procedimento effettivo per ottenerla.

2.6.3 Linguaggi liberi dal contesto

Con lo stesso approccio qui adottato per i linguaggi regolari in [10] vengono introdotti i linguaggi *quasi-context-free* definiti mediante automi a pila e grammatiche libere dal contesto modificate con registri.
Il principale risultato è che seppur modificati con registri i due modelli, automi a pila e grammatiche libere dal contesto, mantengono l'equivalenza che li caratterizzava anche su alfabeti finiti; a prova di ciò vengono forniti procedimenti effettivi per trasformare uno nell'altro e viceversa.
Come per i linguaggi quasi-regular anche nei quasi-context-free il pumping lemma non si applica e vale l'indistinguibilità sui nuovi simboli per cui ogni linguaggio quasi-context-free è chiuso su automorfismi che sono identità sull'assegnamento iniziale;

inoltre viene dimostrata la chiusura della classe rispetto ad unione, concatenazione e stella di Kleene. Riguardo ai principali problemi decisionali gli unici risultati riguardano la decidibilità dell'appartenenza di una stringa al linguaggio e del problema di determinare se un linguaggio sia vuoto.

Capitolo 3

Variable Finite Automata

Gli automi con un numero finito di variabili sono introdotti in [15] nel tentativo di trovare un modello che possa essere considerato il naturale analogo su alfabeti infiniti del classico automa a stati finiti e che sia più semplice degli altri già presenti in letteratura.

Intuizione Un VFA lavora esplicitamente sui pattern delle possibili parole del linguaggio mediante un normale automa a stati finiti nel cui alfabeto ci sono anche delle variabili.

3.1 Definizione

Un automa con un numero finito di variabili, abbreviato con VFA, è una coppia:

$$\mathcal{V} =< \Sigma, \mathcal{A} >$$

Σ è l'alfabeto infinito e $\mathcal{A}$ è un normale automa a stati finiti $< S, s_0, \Gamma, \mu, F >$, chiamato *pattern automaton* di $\mathcal{V}$ in quanto di fatto definisce quale sia la struttura delle stringhe appartenenti al linguaggio.

Variabili L'automa $\mathcal{A}$ riesce a denotare linguaggi su Σ, nonostante la sua non finitezza, mediante le variabili contenute nel suo alfabeto $\Gamma = \Sigma_{\mathcal{A}} \cup X \cup \{y\}$:

- $\Sigma_{\mathcal{A}}$ è un sottoinsieme finito di Σ che rappresenta le costanti che possono comparire nei pattern;
- X è un insieme finito di variabili che vengono legate non deterministicamente su $\Sigma \setminus \Sigma_{\mathcal{A}}$, il numero di queste variabili definisce la *larghezza* di $\mathcal{A}$;
- y è una variabile libera che può assumere qualsiasi valore che non sia una costante in $\Sigma_{\mathcal{A}}$ o il valore legato ad una variabile in X.

Pattern Sia υ una parola $\upsilon_1\upsilon_2 \ldots \upsilon_n \in \Gamma^*$ chiamata *pattern* e ω una parola $\omega_1\omega_2 \ldots \omega_n \in \Sigma^*$ chiamata *istanza*, ω è un istanza valida di υ e viceversa υ è un pattern autenticante ω se e solo se:

- $\upsilon_i = \omega_i \; \forall \upsilon_i \in \Sigma_A$, se nel pattern appare una simbolo costante anche nell'istanza deve comparire il medesimo simbolo;

- $\forall \upsilon_i, \upsilon_j \in X, \; \forall \omega_i, \omega_j \notin \Sigma_{\mathcal{A}} : \; \omega_i = \omega_j \Leftrightarrow \upsilon_i = \upsilon_j$, se ad una certa variabile è stato legato un simbolo ad ogni occorrenza di quel simbolo nell'istanza deve corrispondere un occorrenza della variabile nel pattern e viceversa, inoltre due variabili non possono essere legate allo stesso simbolo;

- $\upsilon_i = y, \upsilon_j \neq y \Rightarrow \omega_i \neq \omega_j$, la variabile libera può assumere un valore diverso ad ogni apparizione, l'importante è che non sia il valore di una costante o quello legato ad una variabile.

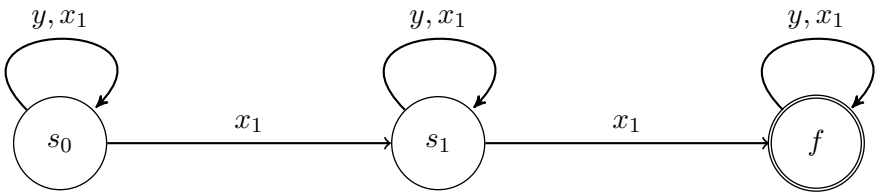
In generale un'istanza può avere più pattern autenticanti e un pattern può avere infinite istanze valide.

Computazione Una computazione di $\mathcal{V}$ su una parola $\omega \in \Sigma^*$ consiste in una computazione di $\mathcal{A}$ su una parola $\upsilon \in \Gamma^*$ che sia un pattern autenticante ω.
Il linguaggio accettato da $\mathcal{V}$ è il sottoinsieme di Σ^* per cui esiste almeno un pattern autenticante in $L(\mathcal{A})$.

Rappresentazione grafica Un VFA è completamente definito dalla rappresentazione grafica del suo *pattern automaton*, con l'accortezza che non ci siano ambiguità nelle etichette degli archi in modo che sia chiaro quali sono le costanti, quali le variabili legate e quale quella libera.

3.2 Esempi

Esempio 1

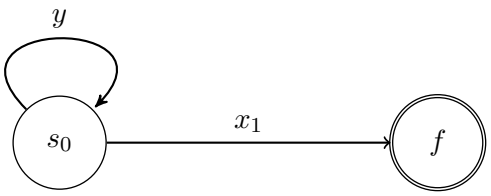


Ogni istanza valida dei pattern riconosciuti dal pattern automaton così definito conterrà un simbolo ripetuto almeno due volte; formalmente:

$$L_1 = \{\sigma_1\sigma_2 \ldots \sigma_n \in \Sigma^* : \exists i, j. \; i \neq j, \; \sigma_i = \sigma_j\}$$

Inoltre rimuovendo da tutti i cicli x_1 si otterrebbe il linguaggio in cui un qualche simbolo è ripetuto esattamente due volte, questo a causa del vincolo che obbliga variabili diverse ad assumere valori diversi.

Esempio 2



Il pattern automaton così definito autentica tutte le istanze in cui l'ultimo simbolo è diverso da tutti quelli precedenti; formalmente:

$$L_2 = \{\sigma_1\sigma_2 \dots \sigma_n \in \Sigma^* : \forall i < n.\ \sigma_i \neq \sigma_n\}$$

Operativamente questo è possibile in quanto il valore legato alla variabile x_1 viene scelto non deterministicamente prima di scorrere la stringa, non è necessario ricordarsi tutti i simboli passati per poi confrontarli sull'ultimo, difatti l'ultimo è noto a priori e può essere confrontato man mano con tutti gli altri simboli.

3.2.1 Linguaggi non esprimibili

- Sia L un linguaggio per il quale non sia possibile trovare un altro linguaggio L_P regolare, tale che L_P contenga dei pattern autenticanti tutte e sole le parole di L; non potendo costruire un pattern automaton L non è riconoscibile da alcun VFA. Ad esempio il classico linguaggio non regolare adattato su alfabeti infiniti $\{\sigma^n\upsilon^n :\ \sigma, \upsilon \in \Sigma\}$ non è riconoscibile.

- Il linguaggio in cui i simboli compaiono sempre in coppia, le cui parole sono della forma $\sigma_1\sigma_1\sigma_2\sigma_2 \dots \sigma_n\sigma_n$, non è riconoscibile in quanto sarebbero necessarie un numero illimitato di variabili in X.
  Per garantire che due simboli siano uguali è necessario associarli mediante il pattern ad una stessa variabile x_i, per ogni simbolo distinto nella parola in ingresso sarebbe però necessaria una distinta variabile dato che il valore di queste non può cambiare durante la computazione; l'alfabeto dei simboli è illimitato e quindi così è anche il numero delle variabili necessarie.

- Molti altri esempi di linguaggi non riconoscibili si possono ottenere da linguaggi in cui tutti i simboli delle parole debbano soddisfare qualche proprietà locale, in tal caso è molto facile trovarsi in una situazione analoga a quella nel punto precedente; per esempio il linguaggio nelle cui parole ogni simbolo è diverso dai due adiacenti non è esprimibile.

3.3 Relazioni con i linguaggi regolari

Gli automi con un numero finito di variabili vengono proposti come un modello più semplice, di quelli già presenti in letteratura, per il riconoscimento dell'estensione dei linguaggi regolari su alfabeti infiniti; a provare una qualche relazione con tali linguaggi ci sono vari fatti:

- I pattern accettati da un VFA sono riconosciuti da un automa a stati finiti e quindi costituiscono un linguaggio regolare[1];

- Dato un automa a stati finiti A, il linguaggio regolare $L(A)$ è riconoscibile da un VFA che utilizza come pattern automaton esattamente A e i cui alfabeti $\Sigma = \Sigma_{\mathcal{A}} = \Gamma$ coincidono con quello di A;[2]

- Limitando l'alfabeto Σ di un VFA ad un suo sottoinsieme finito Σ_f il linguaggio riconosciuto diventa regolare.
  Per ogni possibile assegnamento valido delle variabili X costruiamo un automa a stati finiti modificando le etichette sulle transizioni del pattern automaton:

  – Le costanti rimangono invariate;

  – le variabili x_i vengono sostituite dal valore legato loro dall'assegnamento;

  – y viene sostituita con tutti i valori dell'alfabeto finito che non sono costanti e non fanno parte dell'assegnamento.

  Il linguaggio accettato dall'unione di tutti gli automi così ottenuti è esattamente il linguaggio accettato dal VFA su Σ_f; essendo gli NFA chiusi sull'unione ed essendo l'alfabeto finito, tale linguaggio è regolare.

Riguardo alla *semplicità* del modello non si può dire molto, è quanto mai soggettiva; in generale l'utilizzo esplicito di pattern introduce una certa facilità di utilizzo, permettendo di esprimere e comprendere il linguaggio riconosciuto dall'automa in maniera più immediata, ciò non toglie che altri modelli possano risultare di più semplice utilizzo, dipende dall'utilizzatore.

3.4 Proprietà

3.4.1 Chiusure

Gli automi con un numero finito di variabili sono chiusi rispetto ad unione e intersezione, ma non rispetto al complemento.

3.4.2 Calcolabilità

Emptiness e Non-emptiness Decidere se il linguaggio di un VFA sia vuoto (o con almeno una parola non vuota) è semplice in quanto si riconduce a controllare se lo sia il linguaggio del pattern automaton, un normale automa a stati finiti per il quale il problema è NL-Completo.[3]

[1] Pur essendo gli automi a stati finiti chiusi sull'unione, l'insieme di tutti i pattern riconoscibili non è regolare, per considerare tutte le possibili costanti l'alfabeto $\Sigma_{\mathcal{A}}$ dovrebbe coincidere con Σ perdendo la sua finitezza.

[2] In certi casi usando anche le variabili è possibile che il linguaggio sia denotabile in forma più compatta.

[3] Classe di complessità di problemi risolubili non deterministicamente in spazio logaritmico.

Appartenenza Controllare se una parola ω appartiene al linguaggio è NP-Completo, più complesso ma sempre decidibile. Occorre verificare se esiste un modo di legare i simboli di ω alle variabili del pattern automaton in modo tale da ottenere un pattern autenticante per ω.

Universalità e inclusione Non è decidibile né se un VFA accetti tutte le possibili parole in Σ^*, né tanto meno se dati due automi il linguaggio di uno sia incluso in quello dell'altro.

3.5 VFA deterministici

Imponendo i vincoli di determinismo si ottiene un modello più semplice ma che conserva comunque una certa espressività.

Definizione Un VFA è deterministico (DVFA) se per ogni parola $\omega \in \Sigma^*$ esiste al più un pattern autenticante $\upsilon \in L(\mathcal{A})$ tale che la sua computazione su $\mathcal{A}$ sia unica; non è quindi condizione sufficiente che il pattern automaton sia deterministico, potrebbero comunque esistere più pattern per una singola parola di Σ^*.

Proprietà I DFVA sono chiusi rispetto a tutte le operazioni booleane: unione, intersezione e anche complemento; inoltre i principali problemi decisionali risultano tutti decidibili:

- **Emptiness e Non-emptiness** Come per il modello generale è sufficiente controllare se sia vuoto (o non vuoto) il linguaggio del pattern automaton;
- **Universalità** segue dalla chiusura rispetto al complemento e dal punto precedente, un automa è universale se e solo se il linguaggio accettato dal suo complemento è vuoto, NL-Completo;
- **Appartenenza** verificare se un stringa faccia parte del linguaggio richiede tempo polinomiale, è sufficiente simulare l'esecuzione dell'automa controllando se i vincoli sulle variabili vengono rispettati;
- **Inclusione** dati due automi deterministici il problema di decidere se il linguaggio accettato da uno è incluso in quello accettato dall'altro appartiene alla classe co-NP. [4]

3.5.1 Da VFA a DFVA

Stabilire se un generico automa con un numero finito di variabili sia deterministico è un problema NL-Completo, nel caso non lo sia però non è possibile determinare quale sia, o se ci sia, un equivalente deterministico. Solo per una sotto classe questo è possibile: se non sono presenti y-transizioni, ovvero non viene usata la variabile libera, allora l'automa ha sempre un equivalente deterministico ed è noto un procedimento effettivo per ottenerlo [15].

[4] Classe di complessità di problemi il cui complemento appartiene ad NP.

Capitolo 4

Data words

I *data word languages* sono una classe di linguaggi particolarmente interessante perché, pur essendo l'alfabeto infinito questi ha una componente finita che è molto utile considerare per sfruttare appieno la conoscenza sulla struttura del linguaggio; questa classe di linguaggi modella bene molte situazioni in cui la fonte di non finitezza proviene dai dati mentre la parte di controllo è finita - da qui data word.

4.1 Definizione

Data word Una parola è una stringa $\omega = \omega_1\omega_2 \dots \omega_n \in (\Sigma \times D)^*$

- D è la componente infinita dell'alfabeto, i suoi elementi vengono riferiti come *data values* e non possono essere cablati direttamente nel modello, difatti dovendo questo avere una rappresentazione finita può denotare direttamente solo un numero finito di simboli, è per questo che sono necessari meccanismi come registri e variabili;
- Σ rappresenta la componente finita dell'alfabeto e può essere acceduta direttamente come per i normali linguaggi su alfabeti finiti. I suoi elementi vengono generalmente riferiti come *label* perché difatto etichettano i simboli dell'alfabeto infinito inducendo della classi di equivalenza.

In pratica i simboli dell'alfabeto sono strutturati, ognuno è composto da una coppia di simboli (σ_i, d_i): l'etichetta $\sigma_i \in \Sigma$ e il dato $d_i \in D$; i modelli per data words sfruttano esplicitamente questa caratteristica del linguaggio.

Linguaggio su Σ La presenza delle etichette introduce notevoli vantaggi in quanto considerando solo queste si ha un linguaggio su alfabeto finito, facilmente manipolabile. Per una prima analisi grossolana di una data word si possono considerare solo le etichette, difatti se due etichette sono diverse i simboli complessivi sono sicuramente diversi e non c'è alcun bisogno di controllare i data value. La stringa ottenuta prendendo solo la componente finita di una data word è chiamata *string projection*:

$$\omega = (\sigma_1, d_1)(\sigma_2, d_2) \dots (\sigma_n, d_n)$$

$$STR(\omega) = \sigma_1\sigma_2 \dots \sigma_n$$

Data tree In certi casi è più funzionale considerare oggetti più strutturati di una semplice sequenza di simboli, le parole di un linguaggio su data tree sono *alberi* i cui nodi sono coppie di simboli (σ_i, d_i). Per esempio è immediato che un documento XML si modelli meglio come un albero, piuttosto di una stringa che invece si presta bene per denotare un *path* all'interno del documento-albero.

4.2 Esempi

C'è molto interesse intorno a questa classe di linguaggi in quanto si ritrovano in molti contesti reali, per lo più inerenti a database e verifica formale. In generale sono un ottimo formalismo di basso livello per rappresentare flussi di dati, tracce di programmi, serializzazione di documenti strutturati e comportamento di sistemi concorrenti; è quindi facile immaginare molte situazioni in cui risulta naturale modellare il problema mediante data words. Alla luce della definizione anche molti degli esempi nel capitolo introduttivo possono essere ricondotti a linguaggi di data word.

XML È sempre più diffuso ed ormai si sta affermando come uno standard de facto per lo scambio di dati [28, 29]. Questa crescente importanza ha dato una notevole spinta alla ricerca di soluzioni opportune per i problemi associati ai documenti XML; tra i vari modelli studiati a tal fine compaiono anche i linguaggi di data word e data tree. In generale, senza semplificazioni, in un documento XML non si può limitare il dominio su cui spaziano i valori degli attributi o i contenuti dei tag, questo in diversi modelli comporta la presenza di una componente infinita che non è facile da gestire e che porta facilmente all'indecidibilità [21]. A causa di questa difficoltà molti approcci evitano di considerare tali parti del documento, o ne limitano artificiosamente il dominio, i data values sono però una parte importante di un documento XML e necessitano di essere considerati. Due dei principali problemi in cui ci si scontra con questa non finitezza riguardano la valutazione di query XPath e la validazione di un documento rispetto ad una specifica. [30, 31, 32, 33, 34]

Graph Database Mentre i più comuni database relazionali organizzano i dati in tabelle, i database a grafo li organizzano in grafi dove i nodi rappresentano entità e gli archi relazioni, entrambi hanno associate coppie chiave-valore per le proprietà di interesse. L'analogia tra un percorso all'interno del grafo e una data word è molto forte, ed in fatti certi problemi sono strettamente legati [27, 24].

Risorse dinamiche In un contesto dove ci sono delle risorse che vengono create o acquisite dinamicamente[1] e su cui è possibile applicare un numero finito di operazioni, una data word rappresenta perfettamente una sequenza di azioni dove la componente illimitata identifica le risorse; un linguaggio invece si presta bene per denotare tutte le sequenze con una qualche proprietà. Problemi relativi alle politiche di utilizzo di tali risorse si possono quindi ricondurre a problemi su data word languages [16].

[1] Processi, thread, file, oggetti allocati, URI, URL , connessioni, ...

4.3 Modelli e Risultati

Qualsiasi modello generale che consideri un generico alfabeto infinito può essere usato con linguaggi su data word, così facendo però non si sfrutta appieno la conoscenza sulla struttura del linguaggio e si appiattisce la semantica dei simboli.
Una panoramica dei principali risultati viene fornita prima in [17] e successivamente in [26], nel primo inoltre viene esteso il modello degli automi a memoria finita, precedentemente introdotto nella versione generale, per lavorare su linguaggi di data words; un estensione per gli automi con un numero finito di variabili si può trovare invece in [16].

Ci sono differenti approcci a questi sistemi infiniti, ma principalmente sono estensioni di automi, logica del prim'ordine o logica temporale; qui di seguito alcuni risultati:

- Il potere espressivo di diverse varianti di automa a memoria finita e logica viene considerato in [18];
- Un frammento[2] di *logica del prim'ordine* applicata alle data words per cui è decidibile la soddisfacilità viene presentato in [19], inoltre viene fornito un automa equivalente, *Data Automata*;
- La classe di linguaggi riconoscibile dai Data Automata viene analizzata più a fondo in [20], inoltre viene semplificato il modello fino ad arrivare ai *Class Memory Automata*, equivalenti ma più semplici.
- Generalmente si limita il potere di manipolazione del modello sui data values al solo test di eguaglianza, però ci sono anche casi in cui si assumono conoscenze aggiuntive, in [22] viene introdotto un modello per data tree che sfrutta anche una relazione di ordinamento sui data values, *Ordered-data tree automata*;
- In [23] vengono introdotti due modelli che estendono le espressioni regolari a data word, *Regular expression with memory* e *Regular expression with equality*;
- La logica temporale lineare viene estesa in [25] con i *freeze quantifier* che permettono di memorizzare in un registro il data value nella posizione corrente così da poterlo usare più avanti per fare confronti;

[2] È un frammento della logica del prim'ordine che utilizza due variabili sugli indici dei simboli della parola, con un predicato binario $x \sim y$ per l'uguaglianza dei simboli in posizione x e y, uno per l'ordinamento tra indici $<$ e uno per il successore $+1$.

Capitolo 5

Conclusioni

5.1 Confronto FMA - VFA

Gli automi a memoria finita sono stati uno dei primi modelli per linguaggi su alfabeti infiniti introdotti in letteratura ed hanno avuto un notevole impatto. Essi hanno dato il via a tutto un filone di modelli basati sull'uso di registri e sono stati successivamente usati come elemento di confronto per gli altri modelli. Gli automi con un numero finito di variabili invece sono comparsi più di recente ma sembrano aver riscosso comunque attenzione dalla comunità, nonostante siano stati presentati in un momento dove la produzione di modelli per questi linguaggi è molto alta, si veda la sezione successiva per una breve panoramica.

Dal punto di vista del potere espressivo i due modelli non sono comparabili, ci sono linguaggi esprimibili da uno ma non dal l'altro e viceversa. Il secondo linguaggio d'esempio per gli automi a memoria finita non è riconoscibile da alcun VFA:

$$L_1 = \{a\sigma_1 a\sigma_2 a \dots \sigma_{2n} a \in \Sigma^* : \forall i \in [2, 2n].\ a \neq \sigma_{i-1} \neq \sigma_i \neq a,\ n \geq 1\}$$

Viceversa il secondo esempio per gli automi con un numero finito di variabili non è esprimibile da alcun FMA:

$$L_2 = \{\sigma_1\sigma_2 \dots \sigma_n \in \Sigma^* : \forall i < n.\ \sigma_i \neq \sigma_n\}$$

Sul piano della calcolabilità e complessità dei principali problemi un confronto preciso è difficile in quanto per certi aspetti di uno e per certi aspetti dell'altro non è stato possibile trovare approfondimenti in letteratura; anche rispetto alle proprietà di chiusura, a parte quelle booleane, non si riesce a mettere a confronto i modelli. Nella seguente tabella sono riassunti i principali risultati fin ora noti:

	FMA	VFA	DFMA	DVFA
Chiusure				
Unione	v	v	v	v
Intersezione	v	v	v	v
Complemento	x	x	v	v
Concatenazione	v	?	x	?
Kleene*	v	?	x	?
Reversing	x	?	x	?
Problemi				
Emptiness	v	NL	v	NL
Non-emptiness	NP	NL	NP	NL
Appartenenza	NP	NP	P	P
Universalità	x	x	?	NL
Inclusione	x	x	?	co-NP

I VFA, come ben evidenziato in [15], si propongono come un modello più semplice da capire e utilizzare; sebbene questo dipenda dal gusto soggettivo è innegabile che sia un modello più immediato ed elegante. I FMA hanno un approccio molto più operativo e vanno a modificare notevolmente l'automa classico, dal quale non sempre è banale risalire ad una rappresentazione compatta del linguaggio riconosciuto; i VFA invece definiscono esplicitamente i pattern riconosciuti dal linguaggio utilizzando di fatto un normale automa a stati finiti senza alcuna modifica, essendo questo un modello ben noto risulta più semplice ed immediato lavorarci.

Infine entrambi i modelli hanno varie proprietà interessanti che però non sono state affrontate per la controparte e quindi non si possono confrontare, ad esempio la caratterizzazione in stile Myhill-Nerode degli automi a memoria finita, o lo studio più approfondito della versione deterministica dei VFA e il loro rapporto con il modello non deterministico.

5.2 Modelli e risultati

Di seguito alcuni dei modelli proposti in letteratura:

- In [13] vengono analizzate e confrontate varianti di *Register automata*[1], *logica* e *Pebble automata*, un modello che si basa sulla possibilità di manipolare un numero limitato di puntatori ai simboli della parola mediante una pila;
- *Fresh variable automata*[37] modifica i VFA introducendo la possibilità di slegare le variabili durante la computazione per poi potergli assegnare un nuovo valore;
- *Fresh register automata*[40] estendono gli automi a memoria finita con il concetto di *global freshness*, un valore può essere memorizzato solo se non lo è mai stato prima;

[1] Altro modo di riferire l'approccio basato su registri degli automi a memoria finita.

- *History register automata*[39] modello che utilizza un alcuni insiemi *history* in cui memorizzare selettivamente i simboli per poi compararli successivamente;
- *Guarded variable automata*[38] estende i VFA introducendo delle congiunzioni di uguaglianze e diseguaglianze come guardia alle transizioni, ottiene così un modello che cattura sia gli automi a memoria finita sia i VFA;
- *Usage automata*[41] un automa ad hoc per verificare se una sequenza di azioni rispetta o no una certa politica di utilizzo delle risorse, siano queste statiche o generate dinamicamente;
- *Quasi regular expression*[35] adatta le espressioni regolari per gli alfabeti infiniti ottenendo un modello equivalente agli *automi basati sull'unificazione* [36] e leggermente meno espressivo degli automi a memoria finita;

5.3 Situazione attuale

Il framework teorico attuale è molto frammentato, i molti modelli proposti in letteratura sono spesso incomparabili tra di loro e di conseguenza, in assenza di risultati unificanti, anche quale sia la naturale estensione della classe dei linguaggi regolari[2] non è chiaro, in realtà non è chiaro neppure se tale definizione abbia senso riportata ai linguaggi su alfabeti infiniti.
È difficile trovare modelli *robusti* con buone proprietà di chiusura, *espressivi* abbastanza da catturare linguaggi interessanti e con problemi *decidibili* e di *complessità* accettabile; in molti casi in cui si sono ottenuti risultati soddisfacenti, questo è stato possibile sfruttando fortemente la struttura del singolo problema e usando argomentazioni ad hoc, un framework di carattere più generale non è ancora stato trovato.

[2] E in generale anche delle altre classi di linguaggi, ad esempio in [10] viene usato lo stesso approccio, basato su registri, degli automi a memoria finita per estendere le grammatiche libere dal contesto e gli automi a pila.

Bibliografia

[1] Hopcroft J.E., Motwani R., Ullman J.D.:
Introduction to Automata Theory, Languages, and Computation 3rd edition.
Prentice Hall. (2006)

[2] Papadimitriou C.H.:
Computational Complexity.
Addison-Wesley Pub. (1994)

[3] Ipser E.A.:
The infinite state acceptor and its application to AI.
SIGART Newsletter 107, 29-31. (1989)

[4] Rabin M.O., Scott D.:
Finite automata and their decision problems.
IBM J. Res. Develop. 3, 114-125. (1959)

[5] Bouajjani A., Habermehl P., Mayr R.:
Automatic Verification of Recursive Procedures with one Integer Parameter.
Theoretical Computer Science 295. (2003)

[6] Bielecki M., Hidders J., Paredaens J., Spielmann M., Tyszkiewicz J., Van den Bussche J.:
The navigational power of web browser.
Theory of Computing Systems 50, 213-240. (2012)

[7] Bolling B., Leucker M., Noll T.:
Generalized regular MSC languages.
Technical Report, Department of Computer Science, Aachen University of Technology. (2002)

[8] Grigore R., Distefano D., Petersen R.L., Tzevelekos N.:
Runtime Veri?cation Based on Register Automata.
LNCS 7795, 260-276. (2013)

[9] Kaminsky M., Francez N.:
Finite-memory automata.
TCS 134(2), 329-363. (1994)

[10] Cheng E.Y.C, Kaminsky M.:
Context-free languages over infinite alfabets.
Acta Inf. 35(3), 245-267. (1998)

[11] Sakamoto H., Daisuke I.:
Intractability of decision problems for finite-memory automata.
TCS 231, 297-308. (2000)

[12] Francez N., Kaminsky M.:
An algebraic characterization of deterministic regular languages over infinite alphabets.
TCS 306, 155-175. (2003)

[13] Neven F., Schwentick T., Vianu V.:
Finite state machines for strings over infinite alphabets.
ACM Trans. Comput. Logic, 15(3):403435. (2004)

[14] Benedikt M., Ley C., Puppis G.:
Minimal memory automata.
`http://www.cs.ox.ac.uk/michael.benedikt/papers/myhilldata.pdf`
(2010)

[15] Grumberg O., Kupferman O., Sheinvald S.:
Variable automaton over infinite alphabets.
In: Dediu A.H., Fernau H., Martìn-Vide C. (eds) LATA. LNCS vol 6031, 561-572 Springer (2010)

[16] Degano P., Ferrari G., Mezzetti G.:
Nominal Automata for Resource Usage Control.
LNCS 7381, 125-137. (2012)

[17] Segoufin L.:
Automata and Logics for Words and Trees over an Infinite Alphabet
CSL 2006, LNCS 4207, 41-57. Springer (2006)

[18] Benedikt M., Ley C., Puppis G.:
Automata vs. Logics on Data Words.
CSL 2010, LNCS 6247, 110-124. Springer (2010)

[19] Bojańczyk M., David C., Muscholl A., Schwentick T.:
Two-Variable Logic on Data Words.
LICS 2006, 7-16 (2006)

[20] Björklund H., Schwentick T.:
On notions of regularity for data languages.
TCS 411, 702-705. (2010)

[21] Alon N., Milo T., Neven F., Suciu D., Vianu V.:
XML with Data Values: Type-checking revisited.
In Proceedings of the Twentieth ACM symposium on Principles of Database Systems, 138-149. (2001)

[22] Tan T.:
An Automata Model for Trees with Ordered Data Values.
In LICS'12. (2012)

[23] Libkin L., Vrgoč D.:
Regular Expressions for Data Words.
In LNCS 7180, 274-288. (2012)

[24] Libkin L., Tan T., Vrgoč D.:
Regular Expressions with Binding over Data Words for Querying Graph Databases.
In LNCS 7907, 325-337. (2013)

[25] Demri S., Lazić R.:
LTL with the Freeze Quantifier and Register Automata.
In LICS'06, 17-26. (2006)

[26] Manuel A., Ramanujam R.:
Automata over Infinite Alphabets.
Modern Application of Automata Theory, volume 2 of IISc Research Monographs Series. World Scientific. (2011)

[27] Libkin L., Vrgoč D.:
Regular Path Queries on Graphs with Data.
ICDT'12. (2012)

[28] *List of XML markup languages*
`http://en.wikipedia.org/wiki/List_of_XML_markup_languages`

[29] *List of XML schemas*
`http://en.wikipedia.org/wiki/List_of_XML_schemas`

[30] Bojańczyk M., David C., Muscholl A., Schwentick T., Segoufin L.:
Two-Variable Logic on Data Trees and XML Reasoning.
In PODS'06, 10-19. (2006)

[31] Bojańczyk M., Parys P.:
XPath evaluation in linear time.
In PODS'08, 241-250. (2008)

[32] Benedikt M., Koch C.:
XPath Leashed.
ACM Computing surveys 41. (2008)

[33] Bojańczyk M., Lasota S.:
An extension of data automata that captures XPath.
LICS 2010, 243-252. (2010)

[34] Arenas M., Fan W., Libkin L.:
On the complexity of verifying consistency of XML specifications.
SIAM J. Comput. 38(3), 841-880. (2008)

[35] Kaminsky M., Tan T.:
Regular Expressions for Languages over Infinite Alphabets.
Journal Fundamenta Informaticae 69, 301-318. (2006)

[36] Tal A.:
Decidability of inclusion for unification based automata.
Master thesis, Department of Computer Science, Technion - Israel Istitute of Technology. (1999)

[37] Belkhir W., Chevalier Y., Rusinowitch M.:
Fresh-Variable Automata for Service Composition.
arXiv:1302.4205 (2013)

[38] Belkhir W., Chevalier Y., Rusinowitch M.:
Guarded Variable Automata over Infinite Alphabets.
arXiv:1304.6297 (2013)

[39] Grigore R., Tzevelekos N.: *History-Register Automata.*
LNCS 7794, 17-33. (2013)

[40] Tzevelekos N.:
Fresh-Register Automata
In proceedings of POPL'11, 295-306. (2011)

[41] Bartoletti M., Degano P., Ferrari G., Zunino R.:
Model checking usage policies.
LNCS 5474, 19-35. (2009)