Introduzione alle Logiche Descrittive

Silvio Ghilardi

22 Dicembre 2007

Queste note costituiscono una introduzione sintetica alle tematiche delle logiche descrittive. Se le informazioni contenute in questi appunti gli risultassero insufficienti o non abbastanza chiare, lo studente è invitato a prendere contatto col docente per farsi dare del materiale aggiuntivo. Indichiamo comunque come punto di riferimento il testo:

AAVV *The Description Logic Handbook*, Cambridge University Press, 2003

Per un adeguato supporto software, lo studente può riferirsi al sistema RACER:

http://www.sts.tu-harburg.de/∼r.f.moeller/racer/

Algoritmi di soddisfacibilità basati su traduzioni efficienti nella logica del primo ordine sono implementati nella versione 3.0 del prover SPASS(che viene utilizzato durante il corso di Logica per le Applicazioni).

ATTENZIONE: *questa versione della dispensa è preliminare, incompleta e tuttora in fase di elaborazione (controllate la data in alto per sapere se la versione in vostro possesso è l'ultima disponibile). In particolare, la stesura della seconda parte della dispensa è rimandata ad un momento successivo (la seconda parte della dispensa non conterrà comunque materiale da utilizzarsi all'interno del corso di Logica per le Applicazioni, ma solo argomenti relativamente più avanzati per il corso di Logica 2).* ***Questa dispensa copre solo una parte del programma per un esame****: si faccia riferimento ai siti dei relativi corsi corsi per le modalità di utilizzo. In caso di dubbi, si chiedano informazioni al docente (****orario di ricevimento: venerdì alle ore 11.30****).*

Indice

Capitolo 1

Parte I

1.1 Motivazioni

La ricerca nel settore della rappresentazione della conoscenza si concentra sulla possibilità di fornire descrizioni ad alto livello di fatti, gerarchie terminologiche e reti concettuali necessarie ad applicazioni 'intelligenti', ossia ad applicazioni in grado di ricavare conseguenze implicite (talvolta profonde o nascoste) di conoscenze esplicitamente disponibili o facilmente accessibili.

In tal senso, la *logica del primo ordine* (FOL) fornisce una prima interessante risposta a queste esigenze: essa si presenta come un formalismo facile, naturale ed espressivo spesso quanto basta per molte necessità pratiche. Tuttavia, il suo punto debole sta nelle prestazioni: fenomeni di indecidibilità e imprevedibili complessità nascoste possono rendere totalmente inefficaci ingenue implementazioni dirette. Per questo motivo, dagli anni 70 in poi, formalismi alternativi (ad esempio i cosiddetti 'semantic networks') sono stati sviluppati per racchiudere basi di conoscenza all'interno di opportune strutture-dati (grafi, ecc.) di facile esplorazione; tuttavia, gli algoritmi di comparazione strutturale impiegati in tali ambiti, si sono rivelati sì efficienti ma purtroppo anche incompleti se applicati ad esempio a problemi di sussunzione (in altre parole, in caso di risposta negativa, tali algoritmi non sono garantiti essere corretti). Analizzando più da vicino queste tematiche, è risultato man mano più chiaro che esiste un problema di fondo relativo al *bilanciamento* del *potere espressivo* da un lato e dell'*efficienza computazionale* dall'altro. Il grande successo (ormai più che decennale) delle *Logiche Descrittive* sta proprio in questo, ossia nel saper coniugare in modo sapiente

espressività ed efficienza: man mano che gli studi e le sperimentazioni relative alle Logiche Descrittive progrediscono, le nostre conoscenze e le nostre capacità di classificare in modo sottile i vari frammenti dei linguaggi logici si fanno sempre più profonde ed adeguate alle esigenze dei vari ambiti applicativi. Attualmente, le Logiche Descrittive vengono utilizzate in settori quali l'ingegneria del software, la diagnostica medica, le librerie digitali, le basi di dati e i sistemi informativi basati sul web (per saperne di più, si consultino i corrispondenti capitoli del citato "Description Logic Handbook"). Infine, ricordiamo che lo standard W3C dato dal linguaggio OWL non è nient'altro che una variante delle logiche descrittive che studieremo in queste note.

1.2 Linguaggi descrittivi

Un linguaggio descrittivo è più semplice di un linguaggio del primo ordine, perchè contiene solo concetti atomici, ruoli ('roles' in inglese) e nomi di oggetti. Un concetto atomico è ad esempio un nome comune come 'padre', 'moglie', ecc; un ruolo è una relazione binaria e un nome di oggetto rappresenta, appunto, un oggetto singolo.

Definizione 1.2.1 *Un* ***linguaggio descrittivo*** $\mathcal{L}$ *consiste di una terna di insiemi finiti* $\langle \mathcal{C}, \mathcal{R}, \mathcal{O}b \rangle$*. Gli elementi di* $\mathcal{C}$ *sono indicati con le lettere* $A, B, \ldots$ *e sono chamati* concetti atomici *di* $\mathcal{L}$*; gli elementi di* $\mathcal{R}$ *sono indicati con le lettere* $r, s, \ldots$ *e sono detti* ruoli *di* $\mathcal{L}$*, mentre gli elementi di* $\mathcal{O}b$ *sono indicati con le lettere* $a, b, \ldots$ *e sono detti* nomi di oggetti *di* $\mathcal{L}$*.*

Nei linguaggi del primo ordine, i concetti atomici vengono tradotti con lettere predicative unarie, i ruoli con lettere predicative binarie e i nomi di oggetti con costanti individuali.

Al momento, concentriamoci sui concetti atomici: essi sono usati per costruire concetti più complessi mediante opportuni costruttori di concetti. Fra i costruttori di concetti annoveriamo certamente gli operatori booleani che indichiamo con $\sim, \sqcap, \sqcup$ per differenziarli (sia pure in modo minimale) dai corrispondenti connettivi della logica proposizionale standard (che continueremo, se ce ne sarà bisogno, ad indicare con $\neg, \wedge, \vee$). Ci riserveremo anche di usare rispettivamente $\top$ e $\bot$ per il concetto universale (sempre soddisfatto) e per il concetto contraddittorio (mai soddisfatto).

Le logiche descrittive si differenziamo fra loro per i costruttori che ammettono sia sui concetti che sui ruoli. Nella logica descrittiva di base, chiamata

$\mathcal{ALC}$ ("Attribute Language with Complement") sono ammessi i costruttori $\exists_r, \forall_r$ per ogni ruolo r; in $\mathcal{ALCQ}$ sono ammessi anche i costruttori $(\geq_r^n), (\leq_r^n)$ per ogni ruolo r e per ogni numero naturale $n \geq 1$. In questa prima parte della dispensa tratteremo solo le logiche $\mathcal{ALC}$ e $\mathcal{ALCQ}$, lasciando per la seconda parte logiche descrittive più potenti che richiedono anche costruttori di ruoli.

Mediante i costruttori booleani possiamo formare ad esempio un concetto composto come $\mathsf{Mother} \sqcup \mathsf{Father}$ che starà ovviamente per "Genitore". Il costrutto $\forall_r$ detto 'restrizione dei valori', una volta applicato al concetto C, indica l'insieme degli enti i cui r-relati soddisfano tutti C. Similmente il costrutto $\exists_r$, detto 'restrizione esistenziale', applicato a C, denota gli enti che posseggono un r-relato che soddisfa C. Il seguente esempio di concetto composto

$$\mathsf{Woman} \sqcap \exists_{\mathsf{has-child}}\mathsf{Person} \sqcap \forall_{\mathsf{has-child}}\mathsf{Male} \tag{1.1}$$

dovrebbe chiarire la situazione. Costruendo $\mathsf{Woman} \sqcap \exists_{\mathsf{has-child}}\mathsf{Person}$ abbiamo costruito il concetto di "Madre", mentre con tutto (1.1) indichiamo il concetto "Madre-di-soli-figli-maschi". L'uso delle 'restrizioni numeriche qualificate' $(\geq_r^n), (\leq_r^n)$ può essere esemplicato dal seguente concetto

$$\mathsf{Woman} \sqcap \geq^4_{\mathsf{has-child}} \mathsf{Person} \tag{1.2}$$

che potrebbe essere proposto come formalizzazione della nozione di "Madre-di-famiglia-numerosa". Chiaramente non è sempre facile nè possibile definire i concetti che interessano la nostra ontologia in questo modo, tuttavia la limitazione a questi (ed altri simili) costrutti porta, come vedremo, significativi benefici sul fronte computazionale.

Definizione 1.2.2 *Sia dato un linguaggio descrittivo $\mathcal{L} = \langle \mathcal{C}, \mathcal{R}, \mathcal{O}b \rangle$. L'insieme degli $\mathcal{ALCQ}$-**concetti** di $\mathcal{L}$ (o, brevemente, dei concetti di $\mathcal{L}$) è definito ricorsivamente come segue:*

(i) *se $A \in \mathcal{C} \cup \{\top, \bot\}$, allora A è un concetto di $\mathcal{L}$;*

(ii) *se C, D sono concetti di $\mathcal{L}$, tali sono $(\sim C), (C \sqcap D), (C \sqcup D)$;*

(iii) *se $r \in \mathcal{R}$ e C è un concetto di $\mathcal{L}$, tali sono $(\exists_r C), (\forall_r C)$;*

(iv) *se $r \in \mathcal{R}$, $n \geq 1$ e C è un concetto di $\mathcal{L}$, tali sono $(\geq_r^n C), (\leq_r^n C)$.*

Nel seguito, tratteremo $\exists_r C$ e $(\geq_r^1 C)$ come sinonimi; gli $\mathcal{ALC}$-concetti si ottengono eliminando la clausola (iv) dalla definizione precedente (segnaliamo che nella letteratura sono considerati anche gli $\mathcal{ALCN}$-concetti che si ottengono dagli $\mathcal{ALCQ}$-concetti restringendo la clausola (iv) al caso in cui C è il concetto $\top$).

Per quanto riguarda l'eliminazione delle parentesi e le precedenze nella lettura dei concetti, stipuliamo che gli operatori unari $\sim, \forall_r, \exists_r, \geq_r^n, \leq_r^n$ leghino più strettamente degli operatori binari $\sqcup, \sqcap$ (questo vuol dire che, ad esempio, $\forall_r A \sqcup B$ va letto con $(\forall_r A) \sqcup B$ e non con $\forall_r (A \sqcup B)$).

1.3 Basi di conoscenza: ABox e TBox

Nel paragrafo precedente abbiamo visto come operare sui concetti atomici per costruire concetti più complessi; a questi ultimi si può dare anche un nome esplicito, così come del resto avviene nella vita quotidiana, o nella terminologia scientifica. Una *definizione esplicita* è una eguaglianza fra un concetto atomico e un concetto arbitrario come in

$$\mathsf{Mother} \;=\; \mathsf{Woman} \sqcap \exists_{\mathsf{has-child}} \mathsf{Person}. \tag{1.3}$$

Un **TBox** *è un insieme finito di definizioni esplicite* (la lettera 'T' di TBox sta per 'terminologia' o anche per 'tassonomia').

Non tutti i TBox possono essere genuinamente interpretati come una collezione di definizioni: affinchè questo avvenga il TBox deve essere **aciclico**. Per essere aciclico, un TBox $\mathbf{T}$ deve soddisfare due condizioni: la prima è che uno stesso concetto non deve essere definito due volte (cioè non deve succedere che $\mathbf{T}$ contenga due *differenti* equazioni con lo *stesso* membro sinistro). La seconda condizione evita fenomeni di circolarità: in altre parole, nessun concetto deve essere definito - direttamente od indirettamente - in termini di se stesso. Per precisare formalmente questa ultima condizione, si deve fare così:

1. si costruisce il grafo diretto $G_{\mathbf{T}}$ che ha come nodi i concetti atomici che compaiono in $\mathbf{T}$ e come archi le coppie ordinate (A, A') tali che A' occorre nel membro destro della definizione esplicita di A in $\mathbf{T}$;

2. si chiede che $G_{\mathbf{T}}$ non contenga cicli, ossia che non ci sia in $G_{\mathbf{T}}$ un cammino che parte da un nodo A e finisce nello stesso nodo A.

$$
\begin{aligned}
\mathsf{Woman} &= \mathsf{Person} \sqcap \mathsf{Female} \\
\mathsf{Man} &= \mathsf{Person} \sqcap \sim\mathsf{Female} \\
\mathsf{Mother} &= \mathsf{Woman} \sqcap \exists_{\mathsf{has-child}}\mathsf{Person} \\
\mathsf{Father} &= \mathsf{Man} \sqcap \exists_{\mathsf{has-child}}\mathsf{Person} \\
\mathsf{Parent} &= \mathsf{Mother} \sqcup \mathsf{Father} \\
\mathsf{GrandFather} &= \mathsf{Father} \sqcap \exists_{\mathsf{has-child}}\mathsf{Parent} \\
\mathsf{Wife} &= \mathsf{Woman} \sqcap \exists_{\mathsf{married-to}}\mathsf{Man}
\end{aligned}
$$

Figura 1.1: Un TBox per la terminologia delle relazioni familiari.

Un esempio di TBox aciclico è presentato nella Figura 1 (che abbiamo ripreso con piccole modifiche dal "Description Logic Handbook", p. 52).

Un TBox aciclico può essere sostituito (a meno di equivalenza logica) con un TBox in cui nessun concetto atomico compare sia come membro destro che come membro sinistro di definizioni esplicite: per far questo, basta operare una serie di sostituzioni, tuttavia tale operazione non è indolore dal punto di vista della complessità (può causare un'esplosione esponenziale in memoria), per cui si preferisce usualmente mantenere la struttura originaria dei TBox e operare con essa in modo accorto durante l'esecuzione degli algoritmi di soddisfacibilità.

Un TBox che non sia aciclico non può essere considerato uno strumento per introdurre nuovi concetti, tuttavia ha senso considerare anche tali TBox: essi non definiscono nulla, ma semplicemente pongono vincoli all'interpretazione dei concetti. Ad esempio, potremmo aggiungere alla terminologia nella terminologia della Figura 1.1 un nuovo concetto $\mathsf{Relatives}$ ('Parente') e ulteriori specifiche del tipo $\mathsf{Mother} \sqsubseteq \mathsf{Relatives}$, $\mathsf{Wife} \sqsubseteq \mathsf{Relatives}$, ecc. Queste specifiche non determinano completamente il senso di $\mathsf{Relatives}$, ma lo precisano solo in parte. Si noti anche che abbiamo usato il 'simbolo di inclusione' $\sqsubseteq$ invece dell'uguaglianza: una scrittura del tipo $C \sqsubseteq D$ (dove C, D sono concetti arbitrari) è detta *inclusione generalizzata di concetti*. Un **TBox generalizzato** è un insieme finito di inclusioni generalizzate fra concetti.

TBox (aciclici o generalizzati) costituiscono il reticolato concettuale di

una base di conoscenza: quest'ultima si avvale però anche di fatti specifici che riguardano individui particolari. Dato un linguaggio descrittivo $\mathcal{L} = \langle \mathcal{C}, \mathcal{R}, \mathcal{O}b \rangle$, chiamiamo:

(i) *asserzioni di concetto* le scritture del tipo $C(a)$, dove C è un concetto arbitrario e $a \in \mathcal{O}b$;

(ii) *asserzioni di ruolo* le scritture del tipo $r(a, b)$, dove $r \in \mathcal{R}$ e $a, b \in \mathcal{O}b$.

$$\begin{array}{ccc} \mathsf{Person}(\mathsf{john}), & \sim\mathsf{Female}(\mathsf{john}), & \mathsf{Person}(\mathsf{mary}), \\ \mathsf{Female}(\mathsf{mary}), & \mathsf{Person}(\mathsf{annie}), & \mathsf{Female}(\mathsf{annie}), \\ \mathsf{married{-}to}(\mathsf{mary}, \mathsf{john}), & \mathsf{has{-}child}(\mathsf{mary}, \mathsf{annie}), & \mathsf{has{-}child}(\mathsf{john}, \mathsf{annie}) \end{array}$$

Figura 1.2: Un ABox per asserzioni su relazioni familiari.

Un **ABox** è un insieme finito di asserzioni di concetto o di ruolo; una **base di conoscenza** è una coppia costituita da un ABox e da un TBox (generalizzato o meno). Un esempio di ABox è dato dal contenuto della Figura 1.2 (mentre ovviamente i contenuti delle Figure 1.1 e 1.2 insieme costituiscono una base di conoscenza).

Nel prossimo paragrafo, diamo le necessarie nozioni semantiche e introduciamo i problemi di soddisfacibilità legati alle nostre basi di conoscenza.

1.4 Semantica e compiti di ragionamento

La semantica degli $\mathcal{ALC}$- e degli $\mathcal{ALCQ}$-concetti è definita insiemisticamente in termini di interpretazioni. Dato un linguaggio descrittivo $\mathcal{L} = \langle \mathcal{C}, \mathcal{R}, \mathcal{O}b \rangle$, una **interpretazione** per esso è una coppia $\mathcal{I} = (\Delta^{\mathcal{I}}, \cdot^{\mathcal{I}})$, dove $\Delta^{\mathcal{I}}$ è un insieme non vuoto (detto *dominio*) e $\cdot^{\mathcal{I}}$ è la *funzione di interpretazione*, che assegna a ogni concetto atomico $A \in \mathcal{C}$ un insieme $A^{\mathcal{I}} \subseteq \Delta^{\mathcal{I}}$, ad ogni ruolo $r \in \mathcal{R}$ una relazione binaria $r^{\mathcal{I}} \subseteq \Delta^{\mathcal{I}} \times \Delta^{\mathcal{I}}$ e ad ogni nome di oggetto $a \in \mathcal{O}b$ un elemento $a^{\mathcal{I}} \in \Delta^{\mathcal{I}}$. Faremo anche la cosiddetta *assunzione del nome unico* (UNA), secondo la quale se a e b sono diversi, allora $a^{\mathcal{I}} \neq b^{\mathcal{I}}$.

La funzione di interpretazione è estesa induttivamente ai concetti nel modo seguente:

$$\begin{aligned}
\top^{\mathcal{I}} &:= \Delta^{\mathcal{I}} \\
\bot^{\mathcal{I}} &:= \emptyset \\
(\sim C)^{\mathcal{I}} &:= \Delta^{\mathcal{I}} \setminus C^{\mathcal{I}} \\
(C \sqcap D)^{\mathcal{I}} &:= C^{\mathcal{I}} \cap D^{\mathcal{I}} \\
(C \sqcup D)^{\mathcal{I}} &:= C^{\mathcal{I}} \cup D^{\mathcal{I}} \\
(\forall_r C)^{\mathcal{I}} &:= \{d \in \Delta^{\mathcal{I}} \mid \forall d' \text{ (se } (d,d') \in r^{\mathcal{I}}, \text{ allora } d' \in C^{\mathcal{I}})\} \\
(\exists_r C)^{\mathcal{I}} &:= \{d \in \Delta^{\mathcal{I}} \mid \exists d' \ ((d,d') \in r^{\mathcal{I}} \text{ e } d' \in C^{\mathcal{I}})\}
\end{aligned}$$

Per gli $\mathcal{ALCQ}$-concetti occorre aggiungere anche le due seguenti clausole:

$$\begin{aligned}
(\geq_r^n C)^{\mathcal{I}} &:= \{d \in \Delta^{\mathcal{I}} \mid card(\{d' \in C^{\mathcal{I}} \mid (d,d') \in r^{\mathcal{I}}\}) \geq n\} \\
(\leq_r^n C)^{\mathcal{I}} &:= \{d \in \Delta^{\mathcal{I}} \mid card(\{d' \in C^{\mathcal{I}} \mid (d,d') \in r^{\mathcal{I}}\}) \leq n\}
\end{aligned}$$

dove con $card(X)$ si indica la cardinalità dell'insieme X. Una interpretazione $\mathcal{I}$ *soddisfa*:

- una definizione esplicita $A = C$ se e solo se $A^{\mathcal{I}} = C^{\mathcal{I}}$;
- una inclusione generalizzata fra concetti $C \sqsubseteq D$ se e solo se $C^{\mathcal{I}} \subseteq D^{\mathcal{I}}$;
- una asserzione di concetto $C(a)$ se e solo se $a^{\mathcal{I}} \in C^{\mathcal{I}}$;
- una asserzione di ruolo $r(a,b)$ se e solo se $(a^{\mathcal{I}}, b^{\mathcal{I}}) \in r^{\mathcal{I}}$.

Diciamo che $\mathcal{I}$ è **modello di un TBox** (generalizzato) se e solo se soddisfa tutte le definizioni esplicite (tutte le inclusioni generalizzate fra concetti) che esso contiene; $\mathcal{I}$ è **modello di un ABox** se e solo se soddisfa tutte le asserzioni (di concetti o di ruoli) che esso contiene. Infine $\mathcal{I}$ è **modello di una base di conoscenza** se e solo se è modello sia del TBox che dell'ABox che costituiscono tale base di conoscenza.

Vediamo ora quali sono i compiti di ragionamento ('reasoning tasks') che ha senso considerare per una data base di conoscenza $\mathbf{K}$ che supponiamo formata da un TBox $\mathbf{T}$ e da un Abox $\mathbf{A}$.

1. *Problema della Consistenza*: esiste un modello di $\mathbf{K}$?
2. *Problema della Soddisfacibilità*: dato un concetto C esiste un modello $\mathcal{I}$ di $\mathbf{K}$ tale che $C^{\mathcal{I}} \neq \emptyset$?

3. *Problema della Sussunzione*: data una inclusione generalizzata fra concetti $C \sqsubseteq D$, è vero che in ogni modello $\mathcal{I}$ di $\mathbf{K}$ si ha che $C^{\mathcal{I}} \subseteq D^{\mathcal{I}}$?

4. *Problema dell'Equivalenza*: dati due concetti C, D, è vero che in ogni modello $\mathcal{I}$ di $\mathbf{K}$ si ha che $C^{\mathcal{I}} = D^{\mathcal{I}}$?

5. *Problema della Disgiunzione*: dati due concetti C, D, è vero che in ogni modello $\mathcal{I}$ di $\mathbf{K}$ si ha che $C^{\mathcal{I}} \cap D^{\mathcal{I}} = \emptyset$?

6. *Problema dell'Istanza*: data un'asserzione (di ruolo o di concetto), è vero che ogni modello $\mathcal{I}$ di $\mathbf{K}$ è modello anche di essa?

7. *Problema del 'Retrieval'*: dato un concetto C, trovare tutti i nomi di oggetto $a \in \mathcal{O}b$ tali che ogni modello $\mathcal{I}$ di $\mathbf{K}$ è modello anche dell'asserzione $C(a)$.

Tutti questi problemi possono essere ridotti al primo problema, ossia al Problema della Consistenza, grazie al fatto che abbiamo a disposizione tutti gli operatori booleani; infatti

- C è soddisfacibile se e solo se la base di conoscenza ampliata $\mathbf{K} \cup \{C(a)\}$ è consistente (qui a rappresenta un nome nuovo, cioè non già compreso in $\mathcal{O}b$);
- D sussume C se e solo se $\sim C \sqcap D$ non è soddisfacibile;
- D è equivalente a C se e solo se $(\sim C \sqcap D) \sqcup (\sim D \sqcap C)$ non è soddisfacibile;
- D e C sono disgiunti se e solo se $C \sqcap D$ non è soddisfacibile;
- a è un'istanza di C rispetto a $\mathbf{K}$ se e solo se $\mathbf{K} \cup \{\sim C(a)\}$ non è consistente (un'osservazione analoga vale per le asserzioni di ruolo e un algoritmo -seppur non ottimizzato- di retrieval si può ottenere risolvendo tanti problemi dell'istanza).

In sostanza, potremo concentrarci sul solo **problema della consistenza**; tale problema è decidibile, come vedremo, in $\mathcal{ALC}$ e $\mathcal{ALCQ}$, ma la complessità dell'algoritmo di soluzione dipende in modo significativo dal fatto che il TBox su cui $\mathbf{K}$ è basata sia ciclico o generalizzato.

1.5 La traduzione in FOL

La semantica indicata nel paragrafo precedente consente di tradurre tutti i compiti di ragionamento per le logiche descrittive che abbiamo visto nella logica del primo ordine. A tal proposito, vanno però osservati alcuni fatti: innanzitutto, esistono logiche descrittive potenti che non sono traducibili nella logica del primo ordine e il cui potere espressivo è incomparabile con FOL. In secondo luogo, una traduzione diretta ed ingenua (come quella che proporremo in questo paragrafo) ha ben poche possibilità di produrre prestazioni competitive. In effetti, i tool correnti per le logiche descrittive implementano efficienti algoritmi diretti (ne vedremo uno nel prossimo paragrafo) e non si richiamano a strumenti generali per la logica del primo ordine. Fa eccezione SPASS 3.0, che comunque utilizza una traduzione molto sofisticata in FOL, sperimentata per anni su un tool specializzato apposito (chiamato MSPASS) e trasferita all'interno di SPASS solo nel 2007. Non entreremo nei dettagli di tale traduzione sofisticata, però siccome SPASS è un tool già utilizzato nei nostri corsi, richiamiamo in breve la traduzione ingenua e spieghiamo come *utilizzare* SPASS 3.0 per problemi di soddisfacibilità nelle logiche descrittive.

Ad un linguaggio descrittivo $\mathcal{L} = \langle \mathcal{C}, \mathcal{R}, \mathcal{O}b \rangle$ associamo il linguaggio del primo ordine che contiene:

- per ogni concetto atomico A un predicato unario T_A;
- per ogni ruolo r una relazione binaria T_r;
- per ogni nome di oggetto a una costante individuale che chiamiamo ancora a.

T_A e T_R sono detti essere le 'traduzioni' di A e r; osserviamo che per $\mathcal{ALCQ}$ avremo bisogno anche del predicato binario dell'identità. Traduciamo ora nella sintassi di FOL concetti, TBox, ABox e basi di conoscenza:

(i) *Traduzione dei concetti.* Dato un concetto C e una variabile del primo ordine x, definiamo la formula $ST(C, x)$ per induzione nel modo seguente (la formula $ST(C, x)$ è detta **traduzione relazionale standard** di C

e ha x come unica variabile libera):

$$\begin{aligned} ST(\top, x) &= \top \\ ST(\bot, x) &= \bot \\ ST(\mathrm{A}, x) &= T_A(x) \\ ST(\sim C, x) &= \neg ST(C, x) \\ ST(C \sqcup D, x) &= ST(C, x) \vee ST(D, x) \\ ST(C \sqcap D, x) &= ST(C, x) \wedge ST(D, x) \\ ST(\forall_r C, x) &= \forall y(T_r(x, y) \rightarrow ST(C, y)) \\ ST(\exists_r C, x) &= \exists y(T_r(x, y) \wedge ST(C, y)) \end{aligned}$$

Per gli $\mathcal{ALCQ}$-concetti occorre aggiungere anche le due seguenti clausole:

$$ST((\geq_r^n C), x) = \exists y_1 \cdots \exists y_n (\bigwedge_{i<j} y_i \neq y_j \wedge \bigwedge_{i=1}^{n} (T_r(x, y_i) \wedge ST(C, y_i)))$$

$$ST((\leq_r^n C), x) = \forall y_1 \cdots \forall y_{n+1} (\bigwedge_{i=1}^{n+1} (T_r(x, y_i) \wedge ST(C, y_i)) \rightarrow \bigvee_{i<j} y_i = y_j)$$

(ii) *Traduzione dei TBox.* Una definizione esplicita $A = C$ è tradotta con $\forall x(ST(A, x) \leftrightarrow ST(C, x))$ e una inclusione generalizzata fra concetti $C \sqsubseteq D$ è tradotta con $\forall x(ST(C, x) \rightarrow ST(D, x))$. La traduzione $ST(\mathbf{T})$ di un TBox $\mathbf{T}$ si ottiene facendo la congiunzione delle traduzioni di tutte le definizioni esplicite/inclusioni generalizzate fra concetti che esso contiene.

(iii) *Traduzione degli ABox.* Una asserzione di concetto $C(a)$ si traduce con $ST(C, a/x)$ (ossia sostituendo a a tutte le occorrenze libere di x in $ST(C, x)$); una asserzione di ruolo $r(a, b)$ si traduce con $T_r(a, b)$. La traduzione $ST(\mathbf{A})$ di un ABox $\mathbf{A}$ si ottiene facendo la congiunzione delle traduzioni di tutte le asserzioni che esso contiene.

La *traduzione* $ST(\mathbf{K})$ *di una base di conoscenza* $\mathbf{K} = (\mathbf{T}, \mathbf{A})$ sarà ovviamente costituita dall'enunciato $ST(\mathbf{T}) \wedge ST(\mathbf{A})$. Il seguente Teorema non rappresenta certamente un risultato profondo, ma offre la possibilità di utilizzare (in prima approssimazione) i dimostratori automatici per FOL per risolvere i nostri problemi di consistenza:

Teorema 1.5.1 *La base di conoscenza* $\mathbf{K}$ *è consistente se e solo se l'enunciato* $ST(\mathbf{K})$ *è consistente (nella semantica della logica del primo ordine).*

A titolo esemplificativo, diamo alcune informazioni pratiche su come utilizzare il dimostratore automatico SPASS 3.0: nella Figura 1.3 abbiamo riportato un file eseguibile[1] in cui abbiamo immesso la base di conoscenza fornita dal TBox aciclico della Figura 1.1 e dall'ABox della Figura 1.2.

Diamo alcune indicazioni sommarie sulla sintassi accettata da SPASS 3.0 per le logiche descrittive, prendendo spunto proprio dal file della Figura 1.3 (per maggiori informazioni, si veda la documentazione inclusa nella distribuzione). La prima parte del file consiste di documentazione ad uso solo dell'utente (l'abbiamo racchiusa fra due linee di commento auto-esplicative). Nella seconda parte del file vanno dichiarati tutti i simboli usati nel problema e le loro traduzioni nella logica del primo ordine. Va anche indicata la mappa precisa di traduzione (si vedano le righe `translpairs[.....]`): noi abbiamo scelto di premettere la lettera `T` alla traduzione di tutti i simboli, ad esempio `TMan` traduce `Man`, `Thas_child` traduce `has_child`, ecc. I concetti atomici vanno dichiarati come simboli di arietà zero mentre le loro traduzioni vanno (più propriamente!) dichiarate di arietà 1; i ruoli vanno anch'essi dichiarati di arietà zero, mentre le loro traduzioni vanno dichiarate di arietà 2. I nomi di oggetti vanno dichiarati una volta sola con arietà zero.

La base di conoscenza va inserita fra la linea

```
list_of_special_formulae(axioms, DL).
```

e la successiva `end_of_list.` Le formule relative all'ABox vanno scritte come formule del primo ordine: ad esempio l'asserzione di ruolo

$$\mathsf{married\_to}(\mathsf{mary}, \mathsf{john})$$

viene scritta con

```
formula( Tmarried_to( mary, john ) ).
```

(attenzione ad usare il simbolo tradotto!). Invece per i TBox si usa la sintassi

```
concept_formula(equiv( concetto1, concetto2))).
```

(oppure `concept_formula(implies( concetto1, concetto2))).` per le inclusioni generalizzate fra concetti). Gli operatori booleani vanno scritti in notazione prefissa e sono `and, or, not`, mentre le restrizioni di valore si indicano con `some(ruolo, concetto)` e con `all(ruolo, concetto)`.

[1] È essenziale utilizzare la versione 3.0 di SPASS: le versioni precedenti non supportano la sintassi delle Logiche Descrittive.

```
begin_problem(dispensa).  % inizio intestazione
list_of_descriptions.
name({* Dispensa *}).
author({* S G *}).
status(unknown).
description({* A TBox-ABox example.  *}).
end_of_list.  % fine intestazione
list_of_symbols.  % inizio dichiarazione simboli
functions[ (john,0), (mary,0), (annie,0) ].
predicates[(Person,0), (TPerson,1), (Female,0), (TFemale,1), (Woman,0),
(TWoman,1), (Man,0), (TMan,1), (Mother,0), (TMother,1),
(Father,0), (TFather,1), (Parent,0), (TParent,1), (Grandfather,0),
(TGrandfather,1), (Wife,0), (TWife,1), (has_child,0),
(Thas_child,2), (married_to,0), (Tmarried_to,2) ].
translpairs[(Person,TPerson), (Female,TFemale), (Woman,TWoman),
(Man,TMan), (Mother,TMother), (Father,TFather), (Parent,TParent),
(Grandfather,TGrandfather), (Wife,TWife), (has_child,Thas_child),
(married_to,Tmarried_to)].
end_of_list.  % fine dichiarazione simboli
list_of_special_formulae(axioms, DL).% inizio base di conoscenza
concept_formula(equiv(Woman, and(Person, Female))).  % TBox
concept_formula(equiv(Man, and(Person, not(Female)))).
concept_formula(equiv(Mother, and(Woman, some(has_child,Person)))).
concept_formula(equiv(Father, and(Man, some(has_child,Person)))).
concept_formula(equiv(Parent, or(Mother, Father))).
concept_formula(equiv(Grandfather,and(Father,some(has_child,Parent)))).
concept_formula(equiv(Wife, and(Woman, some(married_to,Man)))).
formula( TPerson( john ) ).  % ABox
formula( not(TFemale( john )) ).
formula( TPerson( mary ) ).
formula( TFemale( mary ) ).
formula( TPerson(annie ) ).
formula( TFemale( annie ) ).
formula( Tmarried_to( mary, john ) ).
formula( Thas_child( mary, annie) ).
formula( Thas_child( john, annie) ).
end_of_list.  % fine base di conoscenza
list_of_special_formulae(conjectures, DL). % problema
formula( TWife(mary) ).  % valido
% formula( TWife( annie ) ).  % non valido
end_of_list.
end_problem.
```

Figura 1.3: Eseguibile di SPASS 3.0.

Infine, nella parte finale del file, sotto la riga

```
list_of_special_formulae(conjectures, DL).
```

(e prima della successiva riga `end_of_list.`) si mette l'asserzione (di concetto, di ruolo, di inclusione fra concetti) che si vuol sapere se è implicata o meno dal resto della base di conoscenza. Si faccia attenzione che SPASS interpreta il tutto come una congettura da provare, perciò per sottoporgli un problema di soddisfacibilità occorre ricorrere ad una opportuna espressione negata (per esempio, per provare se il concetto atomico[2] C è o meno soddisfacibile modulo il resto della base di conoscenza, bisognerà scrivere

```
formula(neg (TC(a))).
```

dove `TC` traduce C e a è un nuovo nome di oggetto).

1.6 Un algoritmo per la soddisfacibilità locale

Lo strumento maestro per affrontare i problemi di consistenza nelle logiche descrittive sono opportuni tableaux: in realtà, non si tratta di tableaux in senso proprio, anche perchè le implementazioni effettive utilizzano, anzichè meccanismi di splitting sintattico tipici dei tableaux, tutte le metodologie e le euristiche dei moderni SAT-solvers. In questo paragrafo introduciamo un tableaux astratto (basato sull'algoritmo KWord noto dalla logica modale) che risulta semplice e flessibile. Rimandiamo invece alla seconda parte delle note l'introduzione di tableaux più flessibili e più tecnici, adattabili a logiche descrittive di maggiore espressività.

Cominciamo coll'enunciare il seguente risultato generale (che non dimostreremo, ma che ci servirà da guida):[3]

Teorema 1.6.1 *Il problema della consistenza di una base di conoscenza* **K** *è* ExpTime*-completo per il caso dei TBox generalizzati e* PSpace*-completo per il caso dei TBox aciclici.*

Ricordiamo che un problema ExpTime è risolvibile usando risorse esponenziali in tempo da una macchina deterministica, mentre un problema

[2] Se C non è atomico ed è piuttosto lungo converrà introdurre un nuovo concetto atomico A, aggiungere $A = C$ al TBox e poi testare A per la soddisfacibilità.

[3] Per la dimostrazione si può consultare ad esempio Cap. 6 del testo P. Blackburn, M. de Rijke, Y. Venema *Modal Logic*, Cambridge University Press, 2001.

PSPACE è risolvibile usando risorse polinomiali di memoria da una macchina (deterministica o non).[4] Un problema PSPACE è anche EXPTIME, il viceversa non è noto.

Non tratteremo qui il caso dei TBox generalizzati (si veda la seconda parte della dispensa per questo ed altro); daremo invece istruzioni dettagliate su come risolvere un problema di consistenza per una base di conoscenza $\mathbf{K} = (\mathbf{T}, \mathbf{A})$ basata su un TBox $\mathbf{T}$ aciclico. Procederemo in modo molto graduale, cominciando dal caso in cui $\mathbf{T}$ é vuoto e $\mathbf{A}$ consiste di una singola asserzione di un $\mathcal{ALC}$-concetto. Prima ancora, abbiamo bisogno di una fase di preprocessamento per portare tutti i nostri concetti in forma normale negativa.

1.6.1 Forme normali negative

Un concetto è in **forma normale negativa** (FNN) se contiene il simbolo di negazione $\sim$ solo davanti ai concetti atomici. È possibile trasformare (in tempo lineare) un concetto C qualunque in un altro che chiamiamo $FNN(C)$ che è forma normale negativa e che è equivalente a C (l'equivalenza significa che si ha $C^{\mathcal{I}} = FNN(C)^{\mathcal{I}}$ per ogni interpretazione $\mathcal{I}$). Per fare tale trasformazione basta applicare le seguenti regole di riscrittura in un ordine qualunque, finchè nessuna regola è più applicabile:[5]

$$\begin{array}{rcl}
\sim\sim C & \leadsto & C \\
\sim(C \sqcup D) & \leadsto & \sim C \sqcap \sim D \\
\sim(C \sqcap D) & \leadsto & \sim C \sqcup \sim D \\
\sim\forall_r C & \leadsto & \exists_r \sim C \\
\sim\exists_r C & \leadsto & \forall_r \sim C \\
\sim(\geq_r^{n+1} C) & \leadsto & (\leq_r^n C) \\
\sim(\leq_r^n C) & \leadsto & (\geq_r^{n+1} C).
\end{array}$$

Assumeremo d'ora in poi che *tutti i concetti che consideriamo siano in FNN.*

[4] Per un noto risultato di teoria della complessità, per i problemi PSPACE non fa differenza utilizzare un modello deterministico o meno di calcolo.

[5] Assumiamo $n \geq 1$ in quanto segue; si ricordi che $(\geq_r^1 C)$ è identificato con $\exists_r C$.

1.6.2 Tipi e profondità

Una prima nozione che ci sarà utile è quella di **profondità** di un concetto: la profondità conta il numero di operatori $\forall_r, \exists_r, \leq_r^n, \geq_r^n$ innestati all'interno di un concetto. Formalmente, la definizione è la seguente:

Definizione 1.6.2 *Ad ogni concetto C associamo un numero positivo $d(C)$ nel modo seguente, per induzione sulla lunghezza di C:*

- $d(A) = d(\sim A) = 0$ *per ogni concetto atomico A;*
- $d(C \sqcap D) = d(C \sqcup D) = max(d(C), d(D))$*;*
- $d(\exists_r C) = d(\forall_r C) = d(\leq_r^n C) = d(\geq_r^n C) = d(C) + 1$.

Dato un insieme finito di concetti X, definiamo *profondità di X* il massimo numero fra i $d(D)$, al variare di $D \in X$. La profondità degli insiemi di concetti (in particolare, dei tipi) costituirà un parametro di complessità utile a provare la terminazione dei nostri algoritmi. Un altro parametro importante è il seguente: dato un ruolo r, l'*indice di r-ampiezza* di X è il numero di concetti del tipo $\exists_r D$ che X contiene.

La nozione di **tipo** (o insieme di Hintikka) risulterà cruciale in quanto segue. Per introdurre la nozione di tipo, occorre fissare un insieme finito di concetti di riferimento che sia chiuso per sotto-concetti: sia $\mathbf{S}$ un tale insieme.

Definizione 1.6.3 *Un $\mathbf{S}$-tipo (o, semplicemente, un tipo) è un sottoinsieme Γ di $\mathbf{S}$ che soddisfa le seguenti proprietà:*

(i) *Γ non contiene simultaneamente A e $\sim A$ per nessun concetto atomico A e inoltre Γ non contiene mai nè $\bot$ nè $\sim \top$;*

(ii) *se Γ contiene $C \sqcap D$, contiene sia C che D;*

(iii) *se Γ contiene $C \sqcup D$, contiene C oppure D.*

Ottenere esempi di tipi è molto semplice: data una interpretazione $\mathcal{I} = (\Delta^{\mathcal{I}}, \cdot^{\mathcal{I}})$ e dato $w \in \Delta^{\mathcal{I}}$, l'insieme $\Gamma_w := \{C \in \mathbf{S} \mid w \in C^{\mathcal{I}}\}$ è un tipo. Tendenzialmente, ci serviremo però dei tipi per costruire le interpretazioni, non viceversa: quindi i tipi dovremo costruirceli 'a mano', tramite procedure accorte di enumerazione per casi.

Nelle applicazioni ci servirà una procedura non deterministica `Type` che, avuto in ingresso un insieme di concetti X restituisca in uscita un tipo Γ tale che $\Gamma \supseteq X$ e $d(X) = d(\Gamma)$. La procedura deve essere *non deterministica*, nel senso che essa deve consistere in una serie di istruzioni che, eseguite in modo esaustivo, danno il risultato voluto: il non determinismo è dovuto al fatto che alcune di queste istruzioni lasceranno libertà di scegliere fra più alternative. Chiediamo però che, eseguendo le istruzioni in tutti i modi possibili, si ottenga comunque un insieme di tipi Γ che includa tutti i tipi *minimali* che estendono X e hanno lo stesso grado di X.[6] Un primo metodo semplice (utilizzato nei tableaux tradizionali) di implementare `Type` è il seguente: dato X, lo si completa eseguendo le istruzioni seguenti

(i) tutte le volte che X contiene un concetto del tipo $C \sqcap D$, si aggiungono a X sia C che D;

(ii) tutte le volte che X contiene un concetto del tipo $C \sqcup D$, si aggiunge a X il concetto C oppure il concetto D;

(iii) se X contiene sia A che $\sim A$ (o se X contiene $\bot$ o $\sim \top$), X viene scartato.

Ribadiamo che applicando (i)-(ii)-(iii) in ordine qualunque ma esaustivamente, si possono ottenere da un X di partenza più tipi (ma anche nessuno, se tutte le alternative vengono scartate da (iii)). Ad esempio, se $X = \{A_1 \sqcap (A_2 \sqcup (A_3 \sqcup \sim A_1))\}$ abbiamo che `Type`(X) può restutuire in uscita

$$\begin{aligned} X_1 &= \{A_1 \sqcap (A_2 \sqcup (A_3 \sqcup \sim A_1)), A_1, A_2 \sqcup (A_3 \sqcup \sim A_1), A_2\} \\ X_2 &= \{A_1 \sqcap (A_2 \sqcup (A_3 \sqcup \sim A_1)), A_1, A_2 \sqcup (A_3 \sqcup \sim A_1), A_3 \sqcup \sim A_1, A_3\} \end{aligned}$$

(si noti che (ii) dava anche una terza alternativa, rimossa però da (iii)).

Invece di utilizzare le (i)-(iii), si può osservare che, se trattiamo i concetti del tipo $\forall_r C$ e $\exists_r C$ alla stregua di concetti atomici, i tipi non sono altro che assegnamenti booleani a sotto-concetti, quindi la ricerca di tipi che soddisfino certe condizioni altro non è che un'istanza di un problema SAT e come tale può essere gestita da un opportuno enumeratore di assegnamenti booleani (tali enumeratori sono usualmente impliciti negli attuali SAT-solvers). In tal modo si possono ottenere implementazioni più efficienti della procedura

[6] La minimalità significa che ogni tipo che estende X e ha lo stesso grado di X deve includere un tipo fornito dalla procedura.

`Type` (si osservi però che l'algoritmo di soddisfacibilità che esporremo si situa ad un livello piuttosto astratto, non dipendente cioè da particolari scelte implementative relative a sottoprocedure come `Type`).

Data una interpretazione $\mathcal{I} = (\Delta^{\mathcal{I}}, \cdot^{\mathcal{I}})$ e dati $w, v \in \Delta^{\mathcal{I}}$ tali che $(w, v) \in r^{\mathcal{I}}$, si ha che il tipo Γ_v è r-accessibile dal tipo Γ_w nel senso della definizione seguente:[7]

Definizione 1.6.4 *Sia $r \in \mathcal{R}$ e siano Γ, Δ due tipi. Diciamo che il tipo Δ è* ***r-accessibile da*** *Γ qualora $\Gamma^{-r} \subseteq \Delta$ (dove abbiamo posto $\Gamma^{-r} = \{C \mid \forall_r C \in \Gamma\}$).*

Dalla definizione risulta chiaro che applicando `Type` a Γ^{-r} si ottengono tutti i tipi minimali r-accessibili da Γ; si osservi anche che, se `Type` è implementata mediante le (i)-(ii)-(iii) (o mediante accorti enumeratori di assegnamenti booleani) essa restituisce sempre in uscita tipi con profondità strettamente *minore* di Γ.

1.6.3 Soddisfacibilità di $\mathcal{ALC}$-concetti

Ci occupiamo ora di testare la soddisfacibilità di singoli concetti: in altre parole, ci occupiamo di basi di conoscenza con TBox vuoto e con ABox consistente di una singola asserzione di un $\mathcal{ALC}$-concetto $C(a)$. Questo caso raggiunge già il limite inferiore di complessità PSPACE del caso generale (per le basi di conoscenza basate su TBox aciclici).

L'idea fondamentale degli algoritmi che vedremo in questo paragrafo è legata al fatto che per queste logiche descrittive è sufficiente limitarsi alle **interpretazioni costruite su alberi**. Supponiamo di avere a disposizione un albero finito T etichettato sui nodi da insiemi di atomi e sui lati da ruoli (chiamiamo ℓ la funzione che associa ad ogni nodo un insieme di atomi e ad ogni lato un ruolo): da tale albero ricaviamo una interpretazione $\mathcal{I} = (\Delta^{\mathcal{I}}, \cdot^{\mathcal{I}})$ fatta nel modo seguente. $\Delta^{\mathcal{I}}$ é l'insieme dei nodi dell'albero, $A^{\mathcal{I}}$ è costituita dai nodi w tali che $A \in \ell(w)$ e $r^{\mathcal{I}}$ è costituita dagli archi (w, v) tali che $\ell(w, v) = r$.

Si può provare che se esiste un'interpetazione che è modello della nostra asserzione $C(a)$ allora ne esiste una costruita su un albero, in cui per di più $a^{\mathcal{I}}$ è proprio la radice dell'albero; si può anche provare che ci si può limitare

[7] Per convincersene, si ricordi, dalle definizioni del paragrafo 1.4, che se $w \in (\forall_r C)^{\mathcal{I}}$ e $(w, v) \in r^{\mathcal{I}}$, allora $v \in C^{\mathcal{I}}$.

agli alberi di profondità al più $d(C)$ e con non più di $|C|$ diramazioni per ogni nodo (qui $|C|$ indica la lunghezza di C). Queste osservazioni sarebbero di per sè sufficienti a costruire un algoritmo per risolvere il nostro problema, però si tratterebbe di un algoritmo inefficiente, ben lontano dal limite di complessità PSPACE che vogliamo ottenere.

Peraltro non è possibile dire molto di più sulle dimensioni dell'albero che serve a costruire l'interpretazione validante per $C(a)$: ci sono casi in cui non si può ottenere un albero (nè in generale un modello) con meno di una quantità esponenziale di nodi. L'unica risorsa che abbiamo sta nel limitare il nostro spazio di ricerca e, soprattutto, nell'esplorare l'albero che dobbiamo costruire *localmente*, senza mai memorizzarlo tutto intero insieme.

Abbiamo già osservato che, data un'interpretazione $\mathcal{I} = (\Delta^{\mathcal{I}}, \cdot^{\mathcal{I}})$ qualunque (ad albero o meno) e dato $w \in \Delta^{\mathcal{I}}$, l'insieme dei sotto-concetti D di C tali che $w \in D^{\mathcal{I}}$ forma un tipo (che abbiamo chiamato Γ_w). Prendiamo come insieme di concetti di riferimento per la costruzione dei tipi proprio l'insieme $\mathbf{S}$ dei sotto-concetti di C: il primo passo nella costruzione del nostro modello ad albero sarà quello di trovare un tipo Γ tale che $C \in \Gamma$ (questo Γ sarà il tipo associato alla radice). L'idea è di continuare così, cioè di associare tipi ai nodi dell'albero che man mano costruiamo.

Disegniamo l'algoritmo usando un modello di macchina non deterministica. Ad ogni passo la macchina procede applicando delle procedure che possono dare più di un risultato: se il risultato scelto ci porta ad un vicolo cieco, rifacciamo tutto e tentiamo in un altro modo, utilizzando cioè altri risultati possibili delle procedure che abbiamo invocato. Se proprio non ce la si fa in nessun modo, vuol dire che $C(a)$ non era soddisfacibile; se invece riusciamo ad andare fino in fondo, avremo costruito la nostra intepretazione ad albero.[8] Dal punto di vista della specifica dell'algoritmo, questa trattazione è molto comoda: ci basterà spiegare cosa deve succedere affinchè il problema della soddisfacibilità di $C(a)$ sia risolto *positivamente*, senza preoccuparci di nient'altro. Faremo anche in modo che le modifiche per i casi che dovremo affrontare poi (TBox non vuoti, restrizioni numeriche, ecc.) siano facili e minimali.

Riassumendo: vogliamo costruire un'interpretazione ad albero e ai nodi degli alberi vogliamo assegnare un tipo (quando associamo un tipo Γ ad un

[8] Questo modo di procedere è giustificato dal fatto che, come si è detto, un problema è PSPACE per una macchina non deterministica se e solo se lo è per una (altra, opportunamente costruita) macchina deterministica.

nodo w, vogliamo che nel modello che verrà costruito Γ risulti contenuto in Γ_w). Per procedere ci servono due procedure non deterministiche, una per costruire tipi e una per costruire gli insiemi di nodi-figli: essendo non deterministiche, tali procedure sono specificate una volta che siano precisati i valori che possono restituire per ogni ingresso.

- La procedura `Type` prende in ingresso un insieme X di sottoconcetti di C e restituisce (se possibile) in uscita un tipo Γ tale che $X \subseteq \Gamma$ e $d(X) = d(\Gamma)$.

- La procedura `Succ` prende in ingresso un ruolo r e un tipo Γ e restituisce (se possibile) in uscita un insieme di N tipi $\Theta_1, \ldots, \Theta_N$, che siano tutti r-accessibili da Γ e di profondità *minore* di Γ; qui N è l'indice di r-ampiezza di Γ e se $\exists_r D_i$ è l'i-esimo concetto r-esistenziale di Γ, si richiede che il tipo Θ_i contenga D_i.

Le procedure `Type` e `Succ`, eseguite in tutti i modi possibili, devono essere in grado di produrre almeno tutti i tipi o le N-ple di tipi *minimali* con le proprietà specificate.

Abbiamo già visto nel paragrafo 1.6.2 che `Type` può essere realizzato (in prima approssimazione) mediante un completamento sintattico ingenuo basato sulle (i)-(ii)-(iii).

Si noti che l'input di `Succ` è una coppia del tipo (r, Γ) e l'output di `Succ` è un insieme di tipi. Per implementare $\texttt{Succ}(r, \Gamma)$ basta applicare `Type` N-volte alla lista $\langle \Gamma^{-r} \cup \{D_1\}, \ldots, \Gamma^{-r} \cup \{D_N\} \rangle$ (qui $\exists_r D_1, \ldots, \exists_r D_N$ sono i concetti r-esistenziali che compaiono in Γ); schematizzando, possiamo scrivere (un po' impropriamente)

$$\texttt{Succ}(r, \Gamma) := \langle \texttt{Type}(\Gamma^{-r} \cup \{D_1\}), \ldots, \texttt{Type}(\Gamma^{-r} \cup \{D_N\}) \rangle.$$

Avendo a disposizione `Type` e `Succ` il nostro algoritmo non deterministico opera nel seguente modo: (1) applica `Type` a $\{C\}$ ottenendo un primo tipo Γ; (2) per ogni ruolo r, applica `Succ` a (r, Γ) ottenendo un insieme di tipi $\mathcal{S}$; (3) richiama ricorsivamente (2) per ogni ruolo r su ciascuno dei $\Theta \in \mathcal{S}$.

Riportiamo qui oltre lo pseudocodice dell'algoritmo; ribadiamo che le procedure invocate alle righe 1 e 3 sono non deterministiche, per cui occorre fare backtracking nel caso in cui i valori scelti non deterministicamente portino in una situazione di stallo. **L'asserzione $C(a)$ è soddisfacibile se e solo se si riesce a portare a termine l'algoritmo con successo**.

Algorithm 1

1: $\Gamma \leftarrow \textsc{Type}(\{C\})$
2: **procedure** ALCSAT(Γ)
3: $\quad \mathcal{S} \leftarrow \bigcup_{r \in \mathcal{R}} \textsc{Succ}(r, \Gamma)$
4: $\quad$ **for all** $\Theta \in \mathcal{S}$ **do**
5: $\quad\quad$ ALCSAT(Θ)
6: $\quad$ **end for**
7: **end procedure**

Prima di concludere il paragrafo, facciamo alcune riflessioni ulteriori sull'Algoritmo 1 (consigliamo il lettore di ritornare su di esse solo dopo essersi bene impadronito dell'algoritmo stesso, ad esempio leggendo preventivamente gli esercizi svolti del prossimo paragrafo 1.6.4).

Dimostriamo (in modo colloquiale, ma in realtà abbastanza preciso) che l'algoritmo soddisfa le specifiche intese.

1. *L'algoritmo termina.* Da quanto richiesto sopra, `Succ` ha la proprietà che per ogni r, Γ, e $\Theta \in \texttt{Succ}(r, \Gamma)$, si ha che $d(\Gamma) > d(\Theta)$. In questo modo (se anche le invocazioni di `Succ` hanno tutte successo), `Succ` si troverà prima o poi a dover gestire un tipo che non contiene concetti del tipo $\exists_r D$ e quindi produrrà la lista vuota in uscita, causando la terminazione dell'algoritmo.

2. *L'algoritmo è corretto.* Se $C(a)$ è soddisfacibile (cioè se esiste $\mathcal{I}$ con $C^{\mathcal{I}} \neq \emptyset$), c'è il modo di eseguire l'algoritmo fino in fondo. In effetti, alla linea 1, `Type` potrà produrre in uscita un $\Gamma \subseteq \Gamma_w$ minimale, dove w è un qualsiasi nodo tale che $w \in C^{\mathcal{I}}$. Per guidare $\texttt{Succ}(r, \Gamma)$, possiamo per ogni $\exists_r D \in \Gamma$ trovare[9] v tale che $(w, v) \in r^{\mathcal{I}}$, $v \in D^{\mathcal{I}}$, e considerare un sottoinsieme minimale di concetti di Γ_v prodotto da $\texttt{Succ}(r, \Gamma)$, ecc.

3. *L'algoritmo è completo.* Se si riesce a portare a termine l'algoritmo con successo fino in fondo, $C(a)$ è soddisfacibile. Per verificarlo, proviamo per induzione su $d(\Gamma)$ che se la procedura $\texttt{ALCSat}(\Gamma)$ della linea 3 ha successo, allora si può costruire una interpretazione basata su un albero $\mathcal{I} = (\Delta^{\mathcal{I}}, \cdot^{\mathcal{I}})$ tale che, detta w la radice di tale albero, si ha che $w \in D^{\mathcal{I}}$ per ogni $D \in \Gamma$. Siccome la procedura ha successo, per ipotesi di induzione, per ogni r e per ogni $\Theta \in \texttt{Succ}(\mathtt{r}, \Gamma)$ esistono interpretazioni ad albero con le rispettive radici $w_{r,\Theta}$ che hanno la proprietà indicata (rispetto a Θ). Basta ora metterle tutte insieme e aggiungere una nuova radice w; la funzione di interpretazione $\cdot^{\mathcal{I}}$ verrà estesa aggiungendo alle $r^{\mathcal{I}}$ le coppie $(w, w_{r,\Theta})$ e agli $A^{\mathcal{I}}$ tali che $A \in \Gamma$ la radice w (gli altri $A^{\mathcal{I}}$ restano invariati). La verifica che vale $w \in D^{\mathcal{I}}$ per ogni $D \in \Gamma$ si fa per induzione sulla lunghezza di D e non offre difficoltà alcuna (si utilizzino la definizione di tipo, di tipi r-accessibili e la specifica di `Succ`).

[9] Siccome $\exists_r D \in \Gamma_w$, vale che $w \in (\exists_r D)^{\mathcal{I}}$ e quindi esiste v tale che $(w, v) \in r^{\mathcal{I}}$ e $v \in D^{\mathcal{I}}$ (si ricordino sempre le definizioni del paragrafo 1.4).

Diamo qualche cenno su un'implementazione più precisa dell'algoritmo che abbiamo visto. Una struttura-dati plausibile potrebbe essere costituita da liste

$$(\Gamma_0, \ldots, \Gamma_n)$$

di tipi di profondità strettamente decrescente. Accanto a questa lista va mantenuta una lista parallela di puntatori

$$(p_0, \ldots, p_n)$$

il cui ruolo è il seguente: p_i punta ad una formula del tipo $\exists_r D$ che appartiene a Γ_i (supponiamo che un ordine totale qualunque fra tali formule sia stato prefissato - ci servirà anche un ordine totale qualunque fra i tipi).[10] Se p_i punta a $\exists_r D$, il tipo Γ_{i+1} deve contenere D ed essere r-accessibile da Γ_i. Il sistema si evolve nel modo seguente: si cerca un tipo Γ_{n+1} che sia accessibile da Γ_n e che contenga D, se $\exists_r D$ è il sottoconcetto puntato da p_n: se lo si trova, lo si aggiunge alla lista, insieme al puntatore p_{n+1} al primo sottoconcetto del tipo esistenziale di Γ_{n+1}. Se in Γ_{n+1} non sono presenti concetti esistenziali (ma se comunque Γ_{n+1} è stato trovato) si sposta p_n sulla prossima formula esistenziale di Γ_n (se non ce ne sono, si cancella Γ_n e si fa lo stesso con Γ_{n-1}, ecc.- se proseguendo cosí si cancella Γ_0, si termina). Se non si trova il tipo Γ_{n+1} con le caratteristiche richieste, si cancella Γ_n e lo si rimpiazza con il prossimo tipo che abbia le caratteristiche richieste per sostituirlo; se non è possibile, si cancella anche Γ_{n-1}, ecc. - se proseguendo così si cancella tutta la lista, si termina segnalando l'insoddisfacibilità. Si noti che, con questa struttura dati, l'occupazione di memoria é molto limitata: si possono utilizzare puntatori anche per specificare gli elementi dei Γ_i e, siccome ogni occorrenza di un sottoconcetto di C può essere puntata una sola volta all'interno dei vari Γ_i (a seconda di quanti operatori del tipo $\exists_r, \forall_r$ si devono attraversare per raggiungere da essa il connettivo principale di C), otteniamo una stima di complessità (che sarebbe ancora lievemente migliorabile) $O(n \cdot Log\, n)$ per quanto riguarda l'occupazione di memoria.

Osservazione 1.6.5 *È importante essere consapevoli che lo schema sopra esposto è ben lontano dal rappresentare un'implementazione efficiente della nostra procedura di soddisfacibilità. Per raggiungere prestazioni apprezzabili, è necessario ristrutturare l'algoritmo in modo da importare tecniche ed euristiche impiegate dai moderni SAT-solvers. In particolare, occorre un meccanismo di splitting semantico nell'implementazione di* `Type`*, unito a precise strategie di controllo del backtracking (da questo punto di vista, le specifiche non deterministiche che ci hanno reso semplice l'esposizione della procedura, vanno poi abbandonate quando si scende ad un livello più concreto). Ad esempio, lo splitting semantico operato su una formula atomica o modalizzata A (durante la ricerca di un assegnamento booleano richiesto dalla sottoprocedura* `Type`*) diventa efficace qualora, rilevata l'inconsistenza di un ramo di calcolo seguito all'assunzione A, si prosegue assumendo $FNN(\sim A)$ nel ramo successivo.*

[10] Quest'ultimo si realizza facilmente considerando i tipi come assegnamenti booleani sui sottoconcetti e ordinando per esempio lessicograficamente tali assegnamenti booleani.

1.6.4 Esempi

In questo paragrafo esaminiamo alcuni esempi e mostriamo contemporaneamente come gestire graficamente in modo efficace ed intuitivo (per gli esercizi svolti manualmente) le esecuzioni dell'Algoritmo 1.

Esempio 1.6.6 *Mostriamo la soddisfacibilità del concetto*

$$\forall_r(A_1 \sqcup A_2) \sqcap (\exists_r \sim A_1) \sqcap \sim(\forall_r A_2).$$

Applicando la procedura di riduzione in FNN, otteniamo il concetto

$$\forall_r(A_1 \sqcup A_2) \sqcap (\exists_r \sim A_1) \ \sqcap (\exists_r \sim A_2)$$

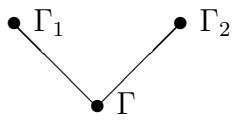
che chiamiamo C per semplicità. Applicando `Type` a C, c'è un solo risultato minimale possibile, ossia il tipo Γ dato da

$$\{C, \forall_r(A_1 \sqcup A_2), \exists_r \sim A_1, \exists_r \sim A_2\};$$

applicando `Succ` alla coppia (r, Γ) si ottiene solo

$$\langle \Gamma_1, \Gamma_2 \rangle := \langle \{A_1 \sqcup A_2, A_2, \sim A_1\}, \ \{A_1 \sqcup A_2, A_1, \sim A_2\} \rangle$$

come possibile risultato. Qui l'algoritmo termina perchè `Succ` *applicato ad un tipo di r-ampiezza zero per ogni r restituisce l'insieme vuoto di tipi come risultato.* L'interpretazione che abbiamo costruito è un albero con tre nodi (che per comodità chiamiamo come i tre tipi), formato da una radice Γ e da due r-figli Γ_1, Γ_2:



Siccome A_1 appartiene solo a Γ_2, avremo $A_1^{\mathcal{I}} = \{w_2\}$ e, similmente, avremo $A_2^{\mathcal{I}} = \{w_1\}$. Graficamente, si può evidenziare tutto questo, se si vuole, etichettando i due archi dell'albero con r, la radice con $\emptyset$ e i due nodi-figli con $\{A_1\}$ e $\{A_2\}$. $\dashv$

Esempio 1.6.7 *Mostriamo la insoddisfacibilità del concetto*

$$\forall_r(A_1 \sqcap A_2) \sqcap \sim((\forall_r A_1) \sqcap (\forall_r A_2)).$$

Applicando la procedura di riduzione in FNN, otteniamo il concetto

$$\forall_r(A_1 \sqcap A_2) \sqcap ((\exists_r \sim A_1) \sqcup (\exists_r \sim A_2))$$

che chiamiamo C per semplicità. Applicando `Type` a C, si ottengono due tipi minimali possibili

$$\Gamma_1 := \{C, \forall_r(A_1 \sqcap A_2), \exists_r \sim A_1\};$$

$$\Gamma_2 := \{C, \forall_r(A_1 \sqcap A_2), \exists_r \sim A_2\}.$$

Tuttavia, in entrambi i casi, l'applicazione di `Succ` produce fallimento: ad esempio, nel caso di $\texttt{Succ}(r, \Gamma_1)$, ciò è dovuto al fatto che non esiste nessun tipo che estende $\{A_1 \sqcap A_2, \sim A_1\}$. Quindi *non riusciamo a portare a termine il nostro algoritmo in nessun modo*, il che vuol dire che il concetto C non era soddisfacibile. ⊣

In esempi più complessi, la necessità di fare backtracking complica notevolmente le cose. *Suggeriamo di operare nel seguente modo, utilizzando in parallelo più "finestre".* Se vogliamo testare la soddisfacibilità di un certo concetto C, per prima cosa applichiamo `Type` a C in tutti i modi possibili, ottenendo dei tipi $\Gamma_1, \ldots, \Gamma_n$. Mettiamo questi tipi in n finestre $F_1, \ldots, F_n$ che conterranno all'inizio un solo tipo ciascuna: ci sarà un risultato di soddisfacibilità se l'algoritmo si applica con successo ad almeno una delle finestre correnti. Per processare una finestra F, scegliamo $\Gamma \in F$ e lo sostituiamo con $\texttt{Succ}(r, \Gamma)$.[11] Si noti che $\texttt{Succ}(r, \Gamma)$ può dare più di un risultato: se ad esempio $\texttt{Succ}(r, \Gamma)$ dà k risultati, si devono fare k copie della finestra F ed esaminarle in sequenza. Se $\texttt{Succ}(r, \Gamma)$ dà zero risultati, la finestra F viene rimossa; infine, se Γ non contiene concetti del tipo $\exists_r D$, allora $\texttt{Succ}(r, \Gamma)$ dà l'insieme vuoto come risultato e Γ viene semplicemente cancellato. Si noti che *il caso di soddisfacibilità si ha quando una finestra viene svuotata completamente e il caso di insoddisfacibilità si ha quando tutte le finestre correnti sono rimosse.*

[11] Se ci sono più ruoli, va calcolata $\texttt{Succ}(r, \Gamma)$ per ogni r. Si noti che Γ viene cancellato una volta calcolati i vari $\texttt{Succ}(r, \Gamma)$ (invece di cancellarlo, lo si può semplicemente asteriscare, in modo da non perdere traccia di tutti i passi di esecuzione dell'algoritmo).

Esempio 1.6.8 *Mostriamo la soddisfacibilità del concetto*

$$\exists_r[(\forall_r\bot \sqcap \exists_r A) \sqcup [\forall_r(\sim A \sqcup B) \sqcap \exists_r A \sqcap \exists_r(\sim A \sqcap \exists_r B)]]$$

che chiamiamo ancora C e che questa volta è già in FNN. Applicando `Type` a $\{C\}$ otteniamo semplicemente $\Gamma := \{C\}$ come risultato, ma applicando `Succ` abbiamo due risultati possibili

$$\Gamma_1 := \{\forall_r\bot, \exists_r A, \dots\};$$

$$\Gamma_2 := \{\forall_r(\sim A \sqcup B), \exists_r A, \exists_r(\sim A \sqcap \exists_r B), \dots\}.$$

Osserviamo che, quando si scrive la descrizione dei tipi che si ottengono applicando le procedure `Type` e `Succ`, non è necessario riportare per intero i concetti che essi contengono: basta limitarsi solo a quelli che sono atomici o che cominciano con uno degli operatori $\exists_r, \forall_r$ (gli altri concetti, pur se necessari per ottenere i tipi in questione, non svolgono più alcun ruolo nel prosieguo dell'algoritmo).[12]

Per schematizzare la situazione, usiamo due finestre come nella tabella che segue:

Finestre	$\mathbf{F_1}$	$\mathbf{F_2}$
Tipi	${}^*\Gamma,\ \Gamma_1$	${}^*\Gamma,\ \Gamma_2$
Alberi	$\Gamma \to \Gamma_1$	$\Gamma \to \Gamma_2$

Ripetiamo che basterà portare a conclusione con successo l'algoritmo su una delle due finestre per concludere la soddisfacibilità; nella prima riga di ogni finestra sono riportati i tipi (quelli non asteriscati sono ancora da analizzare, cioè ad essi va ancora applicata la procedura `Succ`), mentre nella terza riga - che ha solo valore informativo - abbiamo riportato gli archi dell'albero che stiamo costruendo nella rispettiva finestra.

Procediamo ora analizzando la prima finestra; purtroppo $\texttt{Succ}(r, \Gamma_1)$ non riesce a produrre nessun risultato[13] quindi dobbiamo eliminare la prima finestra:

[12] Nel nostro caso, a Γ_1 andava aggiunto anche il concetto $(\forall_r\bot \sqcap \exists_r A) \sqcup [\forall_r(\sim A \sqcup B) \sqcap \exists_r A \sqcap \exists_r(\sim A \sqcap \exists_r B)]$, mentre a Γ_2 andava aggiunto anche il concetto $\forall_r(\sim A \sqcup B) \sqcap \exists_r A \sqcap \exists_r(\sim A \sqcap \exists_r B)$. Si noti che questi concetti cominciano con gli operatori booleani $\sqcap, \sqcup$ (non sono quindi nè atomici nè quantificati).

[13] Infatti $\texttt{Succ}(r, \Gamma_1)$ dovrebbe trovare un tipo che contiene i concetti $\{A, \bot\}$, ma nessun tipo può contenere $\bot$.

Finestre	$\mathbf{F_2}$
Tipi	${}^*\Gamma,\ \Gamma_2$
Alberi	$\Gamma \to \Gamma_2$

Se analizziamo la finestra superstite, ci accorgiamo che $\texttt{Succ}(r, \Gamma_2)$ produce due risultati possibili, ciascuno dei quali è una coppia di tipi: altrimenti detto, un risultato è costituito dalla coppia

$$\Delta := \{A, B, \dots\} \quad \Theta_1 := \{\sim A, \exists_r B, \dots\}$$

mentre l'altro è costituito dalla coppia[14]

$$\Delta := \{A, B, \dots\} \quad \Theta_2 := \{B, \sim A, \exists_r B, \dots\}.$$

Dovremo riaprire una seconda finestra e riorganizzare i nostri dati come segue:

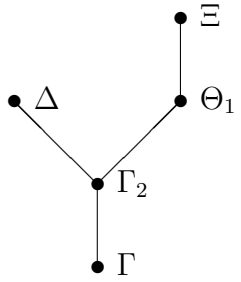
Finestre	$\mathbf{F_{21}}$	$\mathbf{F_{22}}$
Tipi	${}^*\Gamma,\ {}^*\Gamma_2, \Delta, \Theta_1$	${}^*\Gamma,\ {}^*\Gamma_2, \Delta, \Theta_2$
Alberi	$\Gamma \to \Gamma_2; \Gamma_2 \to \Delta; \Gamma_2 \to \Theta_1$	$\Gamma \to \Gamma_2; \Gamma_2 \to \Delta; \Gamma_2 \to \Theta_2$

Concentriamoci ora sulla finestra $\mathbf{F_{21}}$: occorre applicare $\texttt{Succ}$ a Δ e a Θ_1. Siccome Δ ha profondità zero, $\texttt{Succ}(r, \Delta)$ dà l'insieme vuoto come risultato, il che produce (a livello della nostra finestra) il semplice asteriscamento di Δ; invece $\texttt{Succ}(r, \Theta_1)$ dà come unico possibile risultato $\Xi := \{B\}$ e l'ulteriore finale applicazione di $\texttt{Succ}$ a Ξ lo fa semplicemente asteriscare. La situazione diventa la seguente:

Finestre	$\mathbf{F_{21}}$	$\mathbf{F_{22}}$
Tipi	${}^*\Gamma,\ {}^*\Gamma_2, {}^*\Delta, {}^*\Theta_1, {}^*\Xi$	${}^*\Gamma,\ {}^*\Gamma_2, \Delta, \Theta_2$
Alberi	$\Gamma \to \Gamma_2; \Gamma_2 \to \Delta; \Gamma_2 \to \Theta_1; \Theta_1 \to \Xi$	$\Gamma \to \Gamma_2; \Gamma_2 \to \Delta; \Gamma_2 \to \Theta_2$

[14] Il motivo per cui si genera questa situazione è che il secondo tipo di ogni coppia deve contenere $\sim A \sqcup B$ e $\sim A \sqcap \exists_r B$. Quindi, per completarlo, siamo costretti a scegliere una volta $\sim A$ e una volta B, anche se in entrambi i casi dovremo poi prendere sia $\sim A$ che $\exists_r B$. Chiaramente la seconda scelta potrebbe essere eliminata, applicando apposite euristiche per scoprire le ridondanze: ottimizzazioni come questa sono recepite nelle implementazioni e sono legate ad accorte enumerazioni degli assegnamenti booleani a sotto-concetti usati per costruire i tipi.

La finestra $\mathbf{F_{21}}$ è stata esaustivamente analizzata con successo: di fatto, essa è vuota, perchè tutti i suoi tipi sono stati asteriscati. Ne concludiamo la soddisfacibilità del nostro concetto di input C. Per costruire formalmente l'interpretazione che soddisfa C, utilizziamo le informazioni presenti nella finestra $\mathbf{F_{21}}$: il dominio dell'interpretazione $\mathcal{I}$ è costituito dall'albero seguente



L'interpretazione $A^{\mathcal{I}}$ del concetto atomico A comprende il solo nodo Δ, mentre l'interpretazione $B^{\mathcal{I}}$ del concetto atomico B comprende i nodi Ξ, Δ (questo perchè Δ è l'unico tipo che contiene A e Ξ, Δ sono gli unici tipi che contengono B). ⊣

Esercizi 1.6.9 *Di ciascuno dei seguenti concetti dire se è soddisfacibile o meno e, in caso affermativo, costruirne un'interpretazione basata su un albero:*

1. $\forall_r A \sqcap \sim\forall_r \forall_r A$;
2. $\forall_r A \sqcap \sim\forall_r (\sim B \sqcup A)$;
3. $\exists_r (\sim A \sqcup (B_1 \sqcap B_2)) \sqcap ((\forall_r A \sqcap \sim\exists_r B_1) \sqcup (\forall_r A \sqcap \sim\exists_r B_2))$;
4. $\exists_r \exists_r (\sim A \sqcup (B_1 \sqcap B_2)) \sqcap ((\forall_r A \sqcap \sim\exists_r \exists_r B_1) \sqcup (\forall_r A \sqcap \sim\exists_r \exists_r B_2))$;
5. $(\forall_r \forall_r \sim B \sqcup \sim\forall_r (\sim\forall_r A \sqcup B)) \sqcap \sim\forall_r (\sim\forall_r B \sqcup A)$.

1.6.5 TBox aciclici

L'Algoritmo 1 che abbiamo introdotto nel paragrafo precedente faceva leva sui seguenti *parametri di specifica*:

- le definizioni di profondità e di indice di r-ampiezza;

- la definizione dell'insieme di concetti-ambiente **S**, all'interno dei quali ritagliare i tipi;
- la definizione di tipo;
- i valori possibili della procedura non deterministica `Succ`.

Faremo vedere che con piccole modifiche a questi parametri possiamo trattare anche casi via via più generali, mantenendo inalterata la struttura dell'Algoritmo 1.

In questo paragrafo ci occupiamo della consistenza di basi di conoscenza $\mathbf{K} = (\mathbf{T}, \mathbf{A})$, dove **T** é un TBox aciclico e **A** consiste ancora dell'asserzione di un singolo $\mathcal{ALC}$-concetto $C(a)$.

Essendo aciclico, il nostro TBox **T** sarà del tipo

$$A_1 = D_1, \ldots, A_n = D_n$$

dove gli $A_1, \ldots, A_n$ sono tutti distinti e la condizione di aciclicità è soddisfatta. Come segnalato nel paragrafo 1.3, sarebbe possibile sostituire **T** con un altro TBox equivalente

$$A_1 = D'_1, \ldots, A_n = D'_n$$

in cui i concetti 'definiti' $A_1, \ldots, A_n$ non compaiono nei concetti 'definiens' $D'_1, \ldots, D'_n$. Questa operazione non va fatta (produrrebbe un'esplosione esponenziale in memoria) però, anche senza svolgerla esplicitamente, è possibile calcolare la profondità di $D'_1, \ldots, D'_n$: queste nuove profondità vengono *riassegnate* rispettivamente agli $A_1, \ldots, A_n$ al posto della profondità zero che ora spetta solo ai concetti non definiti. Con questa sola modifica, otteniamo la **nuova definizione di profondità** che ci serve (useremo questa nuova nozione da qui fino alla fine del capitolo).

Anche la definizione dell'insieme di concetti-ambiente **S** deve essere cambiate e anche ora la cambieremo in modo definitivo per il resto del capitolo. Il **nuovo S** contiene tutti i sottoconcetti che compaiono nei concetti di $\mathbf{T} \cup \mathbf{A}$ e *nelle FNN delle negazioni* dei concetti che compaiono in $\mathbf{T} \cup \mathbf{A}$.

Infine, ci sarà anche una **nuova** (definitiva) **nozione di tipo**: in aggiunta alle condizioni della Definizione 1.6.3, un tipo Γ dovrà anche soddisfare le seguenti due condizioni:

(iv) se $A = D$ appartiene al TBox **T** e $A \in \Gamma$, allora $D \in \Gamma$;

(v) se $A = D$ appartiene al TBox $\mathbf{T}$ e $\sim A \in \Gamma$, allora $FNN(\sim D) \in \Gamma$.

Con queste sole modifiche si può provare che l'Algoritmo 1 è in grado di decidere la soddisfacibilità dell'asserzione $C(a)$ rispetto ad un TBox aciclico $\mathbf{T}$.

Diamo qualche informazione in più circa i TBox generalizzati. In presenza di un TBox generalizzato (ma anche di un semplice TBox non aciclico), non possiamo più definire una nozione adeguata di profondità e quindi non siamo più in grado di provare la terminazione dell'Algoritmo 1. Il problema è sostanziale: siamo, come si è detto, in un'altra classe di complessità e non possiamo esimerci dal mantenere in memoria *tutto* il modello che stiamo costruendo. Mantenendolo in memoria tutto, possiamo evitare di invocare la procedura della linea 3, cioè `ALCSat`(Γ), qualora tale procedura sia stata già attivata in precedenza sullo stesso Γ (o anche semplicemente su un Γ più piccolo). Questo modo di procedere si può provare corretto, tuttavia la natura non deterministica dell'Algoritmo 1 ci porta ad una procedura di decisione di classe NExpTime che non è ottimale. L'alternativa è di costruire un automa (non deterministico) che lavora su alberi (anche infiniti) e che accetta precisamente gli alberi che corrispondono alle interpretazioni che ci interessano: si tratterà poi di testare se l'automa è vuoto o meno, test che di solito non è computazionalmente pesante. L'automa si costruisce grosso modo così: si usano l'insieme delle parti dei concetti atomici come alfabeto, i tipi come stati e la specifica di `Succ` come funzione di transizione (non c'è bisogno di specificare stati accettanti e non accettanti, l'automa accetta sempre se non si blocca). La procedura che si ottiene in questo modo ha la complessità ExpTime giusta (che cioè si accorda con il limite inferiore noto), tuttavia presenta vari inconvenienti pratici, il maggiore dei quali sta nel fatto che richiede *sempre e comunque* la costruzione di un oggetto combinatorio (l'automa) di dimensioni esponenziali. Per questo motivo, la procedura non deterministica viene usualmente preferita nelle implementazioni. Riprenderemo tutto il discorso nel corso del Capitolo 2.

1.6.6 Soddisfacibilità di $\mathcal{ALCQ}$-concetti

Ampliamo il nostro problema di consistenza a basi di conoscenza $\mathbf{K} = (\mathbf{T}, \mathbf{A})$, dove $\mathbf{T}$ é un TBox aciclico e $\mathbf{A}$ consiste dell'asserzione di un singolo concetto $C(a)$, che però ora è un $\mathcal{ALCQ}$-concetto. Per trattare questo ampliamento manterremo tutti i cambiamenti dei parametri di specifica dell'Algoritmo 1 fatti nel precedente paragrafo e cambieremo però anche la definizione di indice di r-ampiezza e la specifica dei valori possibili della procedura `Succ`.

Prima di continuare, occorre una precisazione relativa al modo di misurare la complessità dei problemi che coinvolgono restrizioni numeriche sui concetti ("qualified number restrictions" in inglese, da cui la lettera $\mathcal{Q}$ usata nella sigla $\mathcal{ALCQ}$). Quando scriviamo concetti come $(\geq_r^n D)$ utilizziamo numeri scritti presumibilmente con la solita notazione ad esempio decimale: così però un numero come 145798 contribuisce con il valore di lunghezza 6 alla misura di

complessità dell'input e ottenere procedure di soddisfacibilità che lavorano in spazio polinomiale rispetto a questa misura di complessità dell'input è sì possibile, ma piuttosto laborioso. Per questo motivo, misureremo i numeri in notazione unaria (n sarà visto come tale, conterà cioè come n stanghette consecutive), rimandando il lettore interessato alla letteratura specializzata per le necessarie ottimizzazioni.

Dato un ruolo r e dato un insieme finito di concetti X, il **nuovo indice di r-ampiezza N di X** sarà la somma di tutti gli n tali che $(\geq_r^n D)$ appartiene a X: precisiamo che per ottenere N, lo stesso n verrà sommato tante volte quanti sono i concetti del tipo $(\geq_r^n D)$ che occorrono in X (si ricordi anche che $\exists_r D$ conta come $(\geq_r^1 D)$).

Dato un ruolo r e dato un tipo Γ con indice di r-ampiezza N, diamo una **nuova specifica per i valori possibili di** $\texttt{Succ}(r, \Gamma)$: $\texttt{Succ}(r, \Gamma)$ restituirà una **lista** di $k \leq N$ tipi

$$\Theta_1, \ldots, \Theta_k$$

(tutti di profondità minore di Γ e r-accessibili da Γ) tali che:

(i) se $(\leq_r^n E)$ appartiene a Γ, allora ogni Θ_i contiene E oppure $FNN(\sim E)$;

(ii) se $(\leq_r^n E)$ appartiene a Γ, allora al più n fra i $\Theta_1, \ldots, \Theta_k$ contengono E;

(iii) se $(\geq_r^n D)$ appartiene a Γ, allora almeno n fra i $\Theta_1, \ldots, \Theta_k$ contengono D.

Si noti che la condizione (i) è importante, perchè in sua assenza potremmo non avere informazioni sul fatto che E appartenga o meno a Θ_i (e non potremmo quindi sapere se E sarebbe vero o meno nell'i-esimo nodo successore all'interno del modello che costruiremmo in caso di risultato positivo dell'algoritmo).

Con queste modifiche non è difficile vedere che l'Algoritmo 1 è in grado di decidere la soddisfacibilità dell'$\mathcal{ALCQ}$-concetto C rispetto ad un TBox aciclico $\mathbf{T}$.

1.6.7 Basi di conoscenza acicliche in $\mathcal{ALCQ}$

Infine ci occupiamo del problema della consistenza di basi di conoscenza

$$\mathbf{K} = (\mathbf{T}, \mathbf{A})$$

dove **T** é un TBox aciclico e **A** è un ABox arbitrario (che comprende sia asserzioni di ruoli che asserzioni di concetti). L'idea è di riutilizzare l'Algoritmo 1 con i parametri di specifica introdotti nei due paragrafi precedenti, facendolo però precedere da una fase di **completamento** che pre-processi il problema. Tutta la procedura richiede i seguenti tre passi.

- *Primo Passo.* Supponiamo che **A** contenga n nomi di oggetti $a_1, \ldots, a_n$.[15] Mediante la funzione `Type`, costruiamo non deterministicamente dei tipi $\Gamma_1, \ldots, \Gamma_n$ che abbiano le seguenti proprietà per ogni $i, j \in \{1, \ldots, n\}$ (ovviamente, se non è possibile farlo, la base di conoscenza è subito dichiarata inconsistente):
  - se $C(a_i) \in \mathbf{A}$, allora $C \in \Gamma_i$;
  - se $r(a_i, a_j) \in \mathbf{A}$, allora Γ_j è r-accessibile da Γ_i;
  - se $r(a_i, a_j) \in \mathbf{A}$ e se $(\leq_r^n E)$ occorre in Γ_i, allora Γ_j contiene o E oppure $FNN(\sim E)$.
- *Secondo Passo.* Ottenuti questi tipi $\Gamma_1, \ldots, \Gamma_n$, applichiamo (di nuovo non deterministicamente) una variante di `Succ` per cercare i loro tipi successori (per i successori di questi successori, basterà invece la `Succ` che abbiamo specificato nel paragrafo 1.6.6). Per ogni i e per ogni ruolo r, occorre ripetere la seguente procedura. Siano $a_{j_1}, \ldots a_{j_l}$ i nomi di oggetti tali che $r(a_i, a_{j_1}) \in \mathbf{A}, \ldots, r(a_i, a_{j_l}) \in \mathbf{A}$; sia N l'indice di r-ampiezza di Γ_i. Si indovinano un $s \leq N$ e una lista di s tipi

  $$\Theta_1, \ldots, \Theta_s$$

  (tutti r-accessibili da Γ_i) in modo che la lista

  $$\Theta_1, \ldots, \Theta_s, \Gamma_{j_1}, \ldots, \Gamma_{j_l}$$

  soddisfi le condizioni (i)-(ii)-(iii) del paragrafo 1.6.6.
- *Terzo Passo.* Mettiamo i tipi ottenuti nel Passo 2 (ma non i tipi $\Gamma_1, \ldots, \Gamma_n$ che erano stati ottenuti nel Passo 1) tutti in uno stesso

[15] Ricordiamo che in virtù della (UNA) del paragrafo 1.4, questi nomi devono essere interpretati su n oggetti distinti (lasciar cadere la (UNA) è possibile, ma occorre un ulteriore passo preventivo che indovini non deterministicamente una partizione su questi nomi di oggetti).

insieme S. Per ogni $\Delta \in S$, testiamo la soddisfacibilità della congiunzione dei concetti che compaiono in Δ rispetto al TBox $\mathbf{T}$ utilizzando l'Algoritmo 1, con i parametri di specifica come modificati nei paragrafi 1.6.5-1.6.6 (si tratta di tanti problemi di soddisfacibilità di un singolo $\mathcal{ALCQ}$-concetto rispetto ad un TBox aciclico che sappiamo già risolvere).

Ampliando lo schema di ragionamento utilizzato nel paragrafo 1.6.3, è possibile provare che, se $\mathbf{T}$ è aciclico, la base di conoscenza $\mathbf{K} = (\mathbf{T}, \mathbf{A})$ **è consistente se e solo se è possibile eseguire con successo la successione dei tre passi sopra descritti.**

Si noti che, mancando le restrizioni numeriche, la procedura si semplifica in modo significativo: in altre parole, per testare la consistenza di una base di conoscenza basata su un TBox aciclico in $\mathcal{ALC}$, non è necessario il secondo passo e basta passare direttamente le congiunzioni dei concetti appartenenti agli n-tipi $\Gamma_1, \ldots, \Gamma_n$ al test di soddisfacibilità dell'Algoritmo 1, con i parametri di specifica del paragrafo 1.6.5.

Capitolo 2

Parte II

Al momento, questa parte della dispensa non è ancora stata scritta. Il lettore interessato a materiale accessibile, ampio e di qualità sugli argomenti relativi può consultare le slides del corso ESSLI 2002 disponibili all'indirizzo http://lat.inf.tu-dresden.de/∼clu/esslli.html. Per un quadro completo e via via aggiornato dei risultati su tutto il settore, si consulti il 'Description Logic Navigator' alla pagina web http://www.cs.man.ac.uk/∼ezolin/dl.

2.1 Necessità di formalismi più espressivi

2.2 La logica descrittiva $\mathcal{SHIQ}$

2.3 Tableaux per $\mathcal{SHIQ}$

2.4 La dimostrazione di completezza

2.5 Limiti di Complessità