Programmazione Dichiarativa con Prolog, CLP, CCP, e ASP

Agostino Dovier
Dipartimento di Matematica e Informatica
Università di Udine
Via delle Scienze 206
I-33100 Udine (UD) Italy
http://www.dimi.uniud.it/dovier

Andrea Formisano
Dipartimento di Matematica e Informatica
Università di Perugia
Via Vanvitelli 1
I-06123 Perugia (PG) Italy
http://www.dipmat.unipg.it/~formis

Indice

Capitolo 1. Introduzione 7

Capitolo 2. Richiami di logica del prim'ordine 11
1. Sintassi 12
2. Semantica 15
3. Sostituzioni 20
4. Esercizi 25

Capitolo 3. Programmazione con clausole definite 29
1. Programmi con dominio vuoto—proposizionali 30
2. Programmi con dominio finito - database programming 31
3. Programmi con dominio infinito 33
4. Definizione di funzioni 35
5. Turing completezza 35
6. Esercizi 37

Capitolo 4. Unificazione 39
1. Preliminari 39
2. Il problema dell'unificazione 40
3. Algoritmo di unificazione 41
4. Osservazioni circa l'algoritmo $\mathsf{Unify}(C)$ 46
5. E-unificazione 51
6. Esercizi 56

Capitolo 5. SLD-risoluzione 57
1. SLD-derivazioni 59
2. Indipendenza dal non-determinismo nell'SLD-risoluzione 62
3. SLD-alberi e regole di selezione 67
4. Search rule e costruzione dell'SLD-albero mediante backtracking 70

Capitolo 6. Semantica dei programmi logici 75
1. Semantica osservazionale dei programmi definiti 75
2. Semantica logica (modellistica) di programmi definiti 77
3. Semantica di punto fisso di programmi definiti 82
4. Esercizi 87

Capitolo 7. Programmazione in Prolog 89
1. Liste 89
2. Alberi 91
3. Grafi 92

4. Automi finiti 93
5. Predicati built-in 95
6. Predicati di tipo e manipolazione di termini 97
7. Predicati metalogici o extralogici 99
8. Predicati di input e output 100
9. Il CUT 100
10. Il predicato FAIL 105
11. Operatori 106
12. Meta-variable facility 107
13. Esercizi 109

Capitolo 8. Programmi generali 113
1. Semantica operazionale della negazione 115
2. Confronti tra le tre regole 120
3. Negazione costruttiva 121
4. Implementazione della NaF 122
5. Esercizi 124

Capitolo 9. Programmazione dichiarativa 125
1. Esempi di programmazione ricorsiva 126
2. Approccio *generate and test* alla soluzione di problemi 130
3. Predicati di secondo ordine 136
4. Meta-interpretazione 139
5. Esercizi 141

Capitolo 10. Searching 143
1. Depth-first search 143
2. Hill climbing search 145
3. Breadth-first search 147
4. Best-first search 148
5. Lower cost first search e A* search 149
6. Esercizi 149

Capitolo 11. Parsing e DCG 151
1. Difference list 151
2. Definite clause grammar 152
3. Alcune estensioni 157
4. Esercizi 159

Capitolo 12. Answer set programming 161
1. Regole e programmi ASP 161
2. Una semantica alternativa per la negazione 162
3. Tecniche di programmazione in ASP 167
4. ASP-solver 171
5. Cenni al solver Cmodels 180
6. La negazione esplicita in ASP 181
7. La disgiunzione in ASP 184

8. Esercizi 185

Capitolo 13. Soluzione di problemi con ASP 189
1. Il *marriage problem* 189
2. Il problema delle *N regine* 191
3. Il problema della *zebra evasa* 192
4. Il problema del *map coloring* 195
5. Il problema del *circuito hamiltoniano* 196
6. Il problema della *k-clicca* 196
7. Il problema del *vertex covering* 197
8. Il problema della *allocazione di compiti* 198
9. Il problema del *knapsack* 199
10. Il problema dei *numeri di Schur* 202
11. Il problema della *protein structure prediction* 202
12. Esercizi 205

Capitolo 14. Planning 207
1. Azioni e loro rappresentazione 207
2. Proiezione temporale e calcolo delle situazioni in ASP 210
3. Planning e calcolo degli eventi in ASP 212
4. Una estensione: la esecuzione condizionata 214
5. Esempi di problemi di planning 215
6. Esercizi 222

Capitolo 15. Vincoli e loro risoluzione 225
1. Vincoli e Problemi vincolati 225
2. Risolutori di vincoli 227
3. Constraint Propagation 230
4. Alberi di ricerca 238
5. Esperimenti ed esercizi 243
6. Vincoli globali 245

Capitolo 16. Programmazione logica con vincoli 253
1. Sintassi e semantica operazionale 253
2. CLP(FD) in SICStus 257
3. Constraint reificati 263
4. CLP($\mathbb{R}$) in SICStus 264
5. CLP(FD) in GNU-Prolog 265
6. Esercizi 266

Capitolo 17. CLP(FD): la metodologia *constrain and generate* 269
1. Il problema delle *N regine* 269
2. Il problema del *knapsack* 270
3. Il problema del *map coloring* 271
4. Il *marriage problem* 272
5. SEND + MORE = MONEY 273
6. Uso del predicato `cumulative` 274

7. Il problema della *allocazione di compiti* 275
8. Il problema del *circuito hamiltoniano* 276
9. Il problema dei *numeri di Schur* 277
10. Esercizi 278

Capitolo 18. Concurrent constraint programming 279
1. Concurrent Constraint (Logic) Programming 279
2. Linda 280

Appendice A. Ordini, reticoli e punti fissi 283

Appendice B. Spigolature sull'uso di Prolog e degli ASP-solver 287
1. Prolog e CLP 287
2. ASP-solver 289

Appendice C. Soluzioni degli esercizi 291
1. Esercizi dal Capitolo 2 291
2. Esercizi dal Capitolo 4 291
3. Esercizi dal Capitolo 6 292
4. Esercizi dal Capitolo 7 292
5. Esercizi dal Capitolo 8 297
6. Esercizi dal Capitolo 9 298
7. Esercizi dal Capitolo 11 299
8. Esercizi dal Capitolo 12 300
9. Esercizi dal Capitolo 13 301
10. Esercizi dal Capitolo 16 302
11. Esercizi dal Capitolo 17 304

Appendice. Bibliografia 305

CAPITOLO 1

Introduzione

I primi calcolatori elettronici programmabili comparvero all'inizio degli anni '40 (ENIAC, BABY, EDVAC) ma l'uso effettivo del calcolatore divenne accessibile (a pochi) solo agli inizi degli anni '60. Volendo pertanto assegnare una durata temporale alla storia della programmazione dei calcolatori, della produzione del Software, non possiamo che attribuirle (ad oggi) una quarantina di anni. Per contro, si pensi alla durata della storia dell'architettura, dell'edilizia, delle costruzioni nautiche. La relativa novità della scienza del calcolatore ha come immediata ripercussione il fatto che, quando si intraprende l'attività di produzione del software spesso si sbagliano i *preventivi* di spesa, si valutano erroneamente i *tempi di produzione*, si forniscono prodotti con comportamenti *non previsti*, o con *errori* di funzionamento, o con scarse caratteristiche di *sicurezza*. Un'impresa edile che sbagliasse i preventivi, costruisse monolocali in luogo di tricamere, o edifici che cadessero dopo alcuni mesi non potrebbe sopravvivere. Una ditta che costruisse un'automobile che vi lascia per strada due volte al giorno fallirebbe. Un programma che fa *bloccare* il calcolatore un paio di volte al giorno è considerato nella norma e comunemente venduto e acquistato. Senza polemizzare sull'ostinazione di alcune software house ad utilizzare metodologie degli anni '50, è comunque evidente che non pare possibile accontentarsi di come la progettazione/produzione del Software sia attualmente organizzata. Per migliorare le cose ci vogliono tempo, esperienza e, forse, nuove idee.

Ripassiamo per un attimo quali sono le varie fasi del ciclo di vita del Software.

Analisi dei requisiti: Cliente e "analista" lavorano assieme per capire e ben definire il problema da risolvere.

Specifiche del sistema: Vengono fissate le tipologie di utenti e vengono stabilite quali funzioni debbano essere messe a disposizione per ciascuna tipologia. Vengono stabiliti dei requisiti di performance.

Progetto ad alto livello: Viene progettato il sistema con un (meta) linguaggio ad alto livello, solitamente mescolante aspetti logici, di teoria degli insiemi, e altri operazionali. Si cerca di restare vicino al linguaggio naturale ma di evitarne le ambiguità.

Implementazione: Il progetto del punto precedente viene tramutato in un *programma* scritto in un linguaggio di programmazione. Parti diverse possono essere sviluppate da diverse unità. L'assenza o l'inconsistenza di informazioni in questa fase può far ritornare indietro a ciasuna delle fasi precedenti.

Test/Integrazione tra le parti: Le varie parti vengono integrate e si inizia una fase di *testing* del programma scritto. Ogni errore individuato può far ritornare indietro a ciascuna delle fasi precedenti.

Assistenza/Miglioramento: Il prodotto viene fornito all'utente, il quale con l'utilizzo può verificare l'assenza o l'inesattezza di alcune funzionalità. Può evidenziare

il mal funzionamento generale e può comunque richiedere nuove specifiche. Per ciascuna di queste osservazioni si è costretti a ritornare indietro in qualcuna delle fasi precedenti.

La metodologia sopra descritta si può suddividere in due grosse fasi (i primi tre e i secondi tre punti sopra descritti):

(1) Nella prima fase si cerca di definire *COSA* sia il problema. Il problema viene *dichiarato*.
(2) Nella seconda fase si fornisce una soluzione al problema: si affronta *COME* risolvere il problema dato.

Purtroppo è solo nelle fasi finali che ci si accorge di eventuali errori/inesattezze delle specifiche della prima parte. Ciò comporta un elevato aumento dei costi di produzione e un allungarsi dei tempi di produzione.

Nella prima fase è necessario essere

Formali: il più possibile per evitare ambiguità.
Astratti: il più possibile per permettere libertà e modificabilità nell'affrontare la seconda parte dello sviluppo, tipicamente affrontata con un linguaggio di programmazione imperativo.

Il linguaggio naturale è astratto ma ambiguo e poco conciso e pertanto inadeguato. I tradizionali linguaggi di programmazione imperativi sono orientati a formulare il *come* risolvere un problema piuttosto che a permettere di definirlo in modo astratto. Poiché chi compie questo tipo di lavoro è solitamente di formazione scientifica, il linguaggio che meglio si presta in questa fase è il linguaggio della logica di patrimonio comune, meglio se provvisto di primitive provenienti dalla teoria degli insiemi. Il significato (la semantica) di una proposizione logica è chiaro e non ambiguo. Vi è la possibilità di scendere a livelli di dettaglio senza perdere in concisione (con il linguaggio naturale questo non è di solito possibile). E' inoltre possibile dimostrare in modo formale e rigoroso proprietà dei programmi.

L'idea di base della *programmazione dichiarativa* è pertanto quella di:

- utilizzare un linguaggio di specifiche e progetto basato sulla logica (del prim'ordine)
- fornire un interprete/compilatore per tale linguaggio in modo che la specifica sia *eseguibile*.

In questo modo viene ipoteticamente ad annullarsi il divario tra la fase *cosa* e la fase *come* della ciclo di vita del software. Allo stato attuale ciò non permetterà di avere in modo automatico (quasi magico) un prodotto software con efficienza comparabile a quella di un prodotto nato in modo tradizionale con anni uomo di lavoro. Tuttavia tale metodo di operare permette di avere, in un tempo di progettazione estremamente basso, un *prototipo* in grado di mimare in tutto e per tutto il prodotto finale e che si discosta da questo solo per efficienza. In altri termini, in pochissimi giorni l'utente può testare il prodotto per verificarne la funzionalità ed eventualmente fornire/modificare alcune specifiche. Una volta che l'utente è soddisfatto della funzionalità il prototipo può essere ottimizzato lavorando in modo top-down utilizzando sia il linguaggio stesso che in alcune parti linguaggi tradizionali o pacchetti preesistenti.

Le idee di base della programmazione dichiarativa sono confluite nel linguaggio più utilizzato per la stessa, il Prolog. Il Prolog fu sviluppato agli inizi degli anni '70 grazie agli sforzi congiunti di Bob Kowalski, che iniziò a dimostrare che la logica predicativa poteva essere vista come un linguaggio di programmazione [**Kow74**], e a quelli implementativi del gruppo di Marsiglia guidato da Alain Colmerauer e da quello di Edimburgo guidato da David H. D. Warren. Il tutto fu reso possibile dai lavori sulla *risoluzione* nati nel filone del Theorem-Proving ed in particolare dal metodo sviluppato da J. A. Robinson nel 1965. Spinte in tale direzione vennero anche dal settore dell'intelligenza artificiale (Green, Hewitt 1969). L'implementazione effettiva di Prolog, sulla macchina astratta di Warren (WAM) [**War80**] fu resa possibile dagli studi sull'algoritmo di unificazione, in particolare la procedura di Paterson-Wegman [**PW78**] e l'algoritmo di Martelli-Montanari [**MM82**], che permisero di risolvere il problema in tempo lineare (risp., quasi lineare).

In questo corso si presenterà in dettaglio la semantica operazionale della programmazione basata su clausole definite (Prolog "puro"). Si presenteranno la semantica denotazionale e modellistica dello stesso e si mostreranno i risultati di equivalenza. Provata la Turing-completezza del linguaggio, si forniranno esempi di programmazione dichiarativa in Prolog, prima in modo puro e ricorsivo, poi utilizzando anche le primitive built-in ed extra logiche di Prolog.

Negli ultimi anni è stato evidenziato come sia estremamente naturale effettuare la dichiarazione di problemi facendo uso di *vincoli*. Tale metodologia di programmazione dichiarativa è denominata *Constraint Logic Programming*—CLP [**JL86, JM94**]. Si descriveranno le possibilità di programmazione con vincoli offerte dalle moderne implementazioni di Prolog e si mostrerà come formulare e risolvere in modo dichiarativo, ma con efficienza comparabile a qualunque altro metodo, diversi problemi pratici.

Si illustrerà come sia naturale programmare in modo concorrente con linguaggi con vincoli. Lo schema che si ottiene viene denominato *Concurrent Constraint Programming*—CCP [**SRP01**].

Si mostrerà una nuova filosofia di programmazione dichiarativa denominata *Answer Set Programming*—ASP, che si può far risalire al lavoro sulla Stable Model Semantics di Gelfond e Lifschitz [**GL88**]. Le risposte ad un programma ASP, invece di essere calcolate e restituite una alla volta (mediante varianti della risoluzione), vengono calcolate tutte in una volta, mediante delle procedure bottom-up. Tale metodologia trova impiego qualora l'insieme delle risposte sia finito, come ad esempio, nei problemi NP-completi una volta fissata la dimensione dell'input o nei problemi di Planning.

Il contenuto del corso si basa ma non esclusivamente su diversi testi [**Apt97, MS98, Llo87, SS97, Apt03**] e articoli (si veda la Bibliografia). Si è cercato di fornire una conoscenza di base piuttosto ampia sulle problematiche teorico/semantiche e una discreta conoscenza pratica su come affrontare dichiarativamente la progettazione del Software.

CAPITOLO 2

Richiami di logica del prim'ordine

In questo capitolo richiameremo le principali nozioni relative alla sintassi ed alla semantica della logica del prim'ordine. Verrà data particolare enfasi ai concetti base utilizzati nei prossimi capitoli nello studio della programmazione dichiarativa.

Iniziamo introducendo in modo intuitivo ed informale le nozioni che verranno trattate nelle prossime sezioni con maggior rigore.

Consideriamo le seguenti affermazioni espresse in linguaggio naturale:

(1) Maria è una attrice.
(2) Carlo è più basso di Marco.
(3) 5 è numero primo.
(4) 15 è il prodotto di 5 e 3.
(5) Roma è capitale d'Italia e capoluogo del Lazio.

In queste affermazioni compaiono dei nomi che denotano oggetti o entità, quali Maria, Carlo, 15, Italia, ecc. Inoltre si menzionano anche delle proprietà che vengono attribuite a queste entità, come "essere attrice" o "essere il prodotto di".

In logica dei predicati adotteremo una particolare formalizzazione per esprimere questi concetti in modo rigoroso. Introdurremo degli opportuni simboli *di costante*, ad esempio $a, b, c, d, \ldots$ per denotare specifici oggetti. Parimenti, utilizzeremo dei simboli *di predicato* (ad esempio $p, q, r, s, \ldots$) per riferirci alle proprietà degli oggetti. Seguendo queste convenzioni, le affermazioni precedenti potrebbero essere scritte così:

(1) $p(a)$ se a denota l'entità "Maria" mentre p denota la proprietà "essere attrice".
(2) $q(b, c)$ se b e c denotano rispettivamente l'entità "Carlo" e l'entità "Marco", mentre q denota la proprietà "essere più basso".
(3) $r(a)$ se a denota l'entità "5" mentre r denota la proprietà "essere numero primo".
(4) $p_1(b, c, d)$ se b, c e d denotano le entità "5", "3" e "15", mentre p_1 rappresenta la relazione che sussiste tra due numeri e il loro prodotto.
(5) $q_2(a_1, a_2) \wedge q_3(a_1, a_3)$ se a_1, a_2 e a_3 denotano rispettivamente l'entità "Roma", l'entità "Italia" e l'entità "Lazio". Mentre q_2 denota la proprietà "essere capitale di" e q_3 denota la proprietà "essere capoluogo di".

Si noti che affermazioni elementari possono essere combinate per costruire affermazioni più complesse tramite l'impiego di *connettivi logici*. Ciò accade ad esempio nell'ultimo caso sopra riportato, dove la congiunzione di due affermazione viene espressa utilizzando il connettivo $\wedge$.

In generale quindi avremo bisogno di un linguaggio che preveda simboli per denotare oggetti e relazioni tra oggetti, oltre a dei connettivi per costruire congiunzioni ($\wedge$), disgiunzioni ($\vee$), implicazioni ($\rightarrow$), ... di affermazioni.

Consideriamo ora la affermazione

La madre di Claudio è attrice

In questo caso la proprietà "essere attrice" non è attribuita ad una entità esplicitamente menzionata nella affermazione. Per questo motivo utilizzare un simbolo di costante per denotare l'entità "la madre di Claudio", seppur possibile, non rifletterebbe adeguatamente il significato della affermazione. Notiamo però che nella affermazione precedente, l'oggetto che gode della proprietà "essere attrice" può venir univocamente determinato in base alla relazione di maternità che lo lega all'entità "Claudio". Si tratta di una relazione di tipo funzionale. Questo genere di relazioni vengono espresse nella logica predicativa tramite particolari simboli, detti appunto *di funzione*. Assumendo quindi di denotare l'entità "Claudio" con il simbolo a, la proprietà "essere attrice" con il simbolo p, e il legame di maternità con il simbolo f, possiamo esprimere nella logica predicativa la precedente affermazione come $p(f(a))$. Essa può essere letta come: "all'oggetto denotato da a corrisponde tramite una funzione f un altro oggetto, $f(a)$, che gode della proprietà "essere attrice".

Abbiamo quindi, costanti, predicati e funzioni. Tuttavia tutto ciò non basta ad esprimere in modo formale concetti come:

(1) Esiste un uomo.
(2) Tutti gli uomini sono mortali.
(3) Non tutti gli uomini sono italiani.
(4) Ogni numero naturale è pari o è dispari.

La difficoltà in questo caso risiede nel fatto che con queste proposizioni non solo si attribuiscono proprietà a delle entità, ma tramite le parole *tutti*, *esiste*, *non tutti*, *ogni*, si predica sulla "quantità" di oggetti che godono di tali proprietà.

Per esprimere formalmente questi concetti si introducono le nozioni di variabile (per indicare generici oggetti, solitamente tramite i simboli $X, Y, Z, \ldots$) e di quantificatori *universale* ($\forall$) e *esistenziale* ($\exists$). Le frasi sopra riportate sono espresse quindi così:

(1) $\exists X\, p(X)$, dove la variabile X rappresenta una generica entità mentre p denota la proprietà "essere uomo".
(2) $\forall Y\, (p(Y) \rightarrow m(Y))$, dove la variabile Y rappresenta una generica entità mentre p denota la proprietà "essere uomo" e m denota la proprietà "essere mortale".
(3) $\neg\forall X\, (p(X) \rightarrow i(X))$, dove la variabile X rappresenta una generica entità mentre p denota la proprietà "essere uomo" e i denota la proprietà "essere italiano".
(4) $\forall Y\, (n(Y) \rightarrow (p(Y) \vee d(Y)))$, dove la variabile Y rappresenta una generica entità mentre n denota la proprietà "essere numero naturale" e p denota la proprietà "essere pari" e d denota la proprietà "essere dispari".

Abbiamo così introdotto in modo informale i principali ingredienti che compongono il linguaggio della logica dei predicati. Nelle prossime sezioni daremo una trattazione più rigorosa di questi concetti. Iniziamo dalla sintassi di un linguaggio predicativo.

1. Sintassi

DEFINIZIONE 2.1. Un linguaggio del primo ordine è caratterizzato in maniera univoca da un *alfabeto* (signature) Σ costituito da:

- Un insieme Π di *simboli di predicato*;

- Un insieme $\mathcal{F}$ di *simboli di funzione* (e di *costante*);
- Un insieme infinito numerabile $\mathcal{V}$ di simboli di *variabile.*

Ai simboli di Σ viene associata la funzione di *arità* $ar : \Sigma \longrightarrow \mathbb{N}$ tale che:

- $ar(X) = 0$ per ogni $X \in \mathcal{V}$;
- $ar(f) \geq 0$ per ogni $f \in \mathcal{F}$.
- $ar(p) \geq 0$ per ogni $p \in \Pi$.

Un simbolo $c \in \mathcal{F}$ è un simbolo di *costante* se $ar(c) = 0$.

Per definire gli oggetti del linguaggio si fa uso di parentesi aperte e chiuse, virgole, e dei connettivi logici $\wedge$, $\vee$, $\neg$, $\rightarrow$, $\leftrightarrow$, $\forall$, $\exists$ che si assumono inclusi in ogni linguaggio.

DEFINIZIONE 2.2. Un *termine* è definito in maniera ricorsiva:

- Una variabile è un *termine*;
- Se $t_1, \ldots, t_n$ sono termini e $f \in \mathcal{F}$, $ar(f) = n$, allora $f(t_1, \ldots, t_n)$ è un *termine.*

Dal secondo punto della definizione precedente si deduce che anche le costanti costituiscono termini.

Simboli tipografici tipicamente utilizzati sono: f, g, h per simboli di funzioni, a, b, c per simboli di costante, p, q, r per simboli di predicato, X, Y, Z per le variabili, r, s, t per i termini.

ESEMPIO 2.1. Secondo la Definizione 2.2 avremo che $f(X, f(a, b))$, con $ar(f) = 2$ e $ar(b) = ar(a) = 0$, è un termine. Anche $+(1, \cdot(3, 5))$ è un termine, mentre la scrittura ab non è un termine.

ESEMPIO 2.2. Consideriamo l'alfabeto Σ costituito da: un solo simbolo di predicato binario, $\Pi = \{d\}$; due simboli di funzione, $\mathcal{F} = \{s, z\}$, con $ar(z) = 0$ e $ar(s) = 1$; un insieme infinito $\mathcal{V} = \{X_1, X_2, X_3, \ldots\}$ di variabili. I termini di questo linguaggio sono: $z, s(z), s(s(z)), \ldots\ X_1, s(X_1), s(s(X_1)), \ldots\ X_2, s(X_2), s(s(X_2)), \ldots\ X_3, s(X_3), s(s(X_3)), \ldots$

DEFINIZIONE 2.3. Sia t un termine. Allora diremo che s è un *sottotermine* di t se s è una sottostringa di t ed è a sua volta un termine. Un sottotermine s di t si dice *proprio* se è diverso da t.

ESEMPIO 2.3. Dato il termine $f(f(a, b), g(X, b))$, i suoi sottotermini sono: a, b, X, $f(a, b)$, $g(X, b)$, oltre al termine stesso. La scrittura $f(f(a,$ non è un sottotermine.

È naturale associare un albero finito ad un termine. Tale albero avrà come nodi i simboli che costituiscono il termine stesso. Ad esempio si può rappresentare il termine $f(f(a, b), g(X, b))$ con l'albero in Figura 2.1. Da tale albero si evince la relazione tra sottotermini e sottoalberi dell'albero.

Indicheremo che due termini s e t sono sintatticamente uguali scrivendo $s \equiv t$, mentre con $s \not\equiv t$ indichiamo che sono sintatticamente diversi.

Dato un termine t con $vars(t)$ denotiamo l'insieme delle variabili che occorrono in t. Se $vars(t) = \emptyset$ allora t è detto *ground.*

L'insieme di tutti i termini ground ottenibili da simboli di funzione in $\mathcal{F}$ è denotato con $T(\mathcal{F})$, mentre $T(\mathcal{F}, \mathcal{V})$ denota l'insieme dei termini ottenibili usando simboli di funzione in $\mathcal{F}$ e simboli di variabili in $\mathcal{V}$.

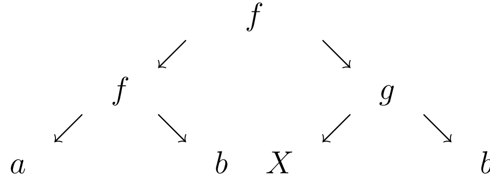


FIGURA 2.1. Albero associato al termine $f(f(a,b),g(X,b))$

DEFINIZIONE 2.4. Se $p \in \Pi$ con $ar(p) = n$ e $t_1, \ldots, t_n$ sono termini, allora $p(t_1, \ldots, t_n)$ è detta *formula atomica* (o *atomo*). Se $t_1, \ldots, t_n$ sono termini ground, allora $p(t_1, \ldots, t_n)$ è detta formula atomica *ground*.

Le formule si definiscono in modo induttivo:

DEFINIZIONE 2.5.

- Una formula atomica è una *formula*.
- Se φ è una formula, allora $\neg\varphi$ è una *formula*.
- Se φ e ψ sono formule, allora $\varphi \vee \psi$ è una *formula*.
- Se φ è una formula e $X \in \mathcal{V}$ allora $\exists X \varphi$ è una *formula*.

Si assumono inoltre le consuete abbreviazioni per i restanti connettivi logici:

- $\varphi \wedge \psi$ sta per $\neg(\neg\varphi \vee \neg\psi)$
- $\varphi \rightarrow \psi$ sta per $\neg\varphi \vee \psi$
- $\varphi \leftrightarrow \psi$ sta per $(\varphi \rightarrow \psi) \wedge (\psi \rightarrow \varphi)$
- $\forall X \varphi$ sta per $\neg\, \exists X \neg\varphi$

Definiamo inoltre un *letterale* come una formula atomica o la negazione di una formula atomica. In particolare un atomo è detto anche *letterale positivo* mentre la negazione di un atomo è detta *letterale negativo*.

DEFINIZIONE 2.6. Diciamo che una variabile X *occorre libera* in una formula φ se sussiste una delle seguenti condizioni:

- φ è formula atomica del tipo $p(t_1, \ldots, t_n)$ e $X \in vars(t_1) \cup \cdots \cup vars(t_n)$
- φ è della forma $\neg\psi$ e X occorre libera in ψ
- φ è della forma $\psi \vee \eta$ e X occorre libera in ψ o X occorre libera in η
- φ è della forma $\exists Y \psi$ se X occorre libera in ψ e $X \neq Y$.

(La definizione si estende facilmente ai restanti connettivi logici $\wedge$, $\rightarrow$, $\leftrightarrow$, $\forall$.)
Se X è presente in φ e X non occorre libera in φ allora diremo che X occorre *legata* in φ.

Se φ è una formula, allora $vars(\varphi)$ denota l'insieme delle variabili che occorrono libere in φ. Se $vars(\varphi) = \emptyset$ allora φ è detta *enunciato*. Se $vars(\varphi) = \{X_1, \ldots, X_n\}$ allora con $\vec{\forall}\varphi$ si denota in breve l'enunciato $\forall X_1 \forall X_2 \cdots \forall X_n\, \varphi$. Similmente, con con $\vec{\exists}\varphi$ si denota l'enunciato $\exists X_1 \exists X_2 \cdots \exists X_n\, \varphi$.

Siano t ed s due termini e X una variabile che occorre libera in t. Allora con la scrittura $t[X/s]$ denotiamo il termine che si ottiene da t sostituendo X con il termine s.

Esempio 2.4. Consideriamo il linguaggio dell'Esempio 2.2. Le seguenti sono formule di questo linguaggio: $d(z,z)$, $d(z,s(s(s(z))))$, $\forall X_2 \exists X_3 d(s(s(X_2)), s(X_3))$, $\exists X_1 d(X_2, s(X_1))$. Le prime tre sono enunciati, mentre X_2 occorre libera nella quarta formula.

2. Semantica

Per assegnare un significato agli oggetti definiti dalla sintassi di un linguaggio dobbiamo scegliere un dominio del discorso ed associare ad ogni oggetto sintattico un opportuno oggetto del dominio. Piu formalmente abbiamo le seguenti definizioni.

Definizione 2.7. Un'*interpretazione* (o *struttura*) $\mathcal{A} = \langle A, (\cdot)^{\mathcal{A}} \rangle$ per un alfabeto Σ consiste di:

- un insieme non vuoto A detto *dominio*,
- una funzione $(\cdot)^{\mathcal{A}}$ tale che:
  - ad ogni simbolo di costante $c \in \mathcal{F}$, con $ar(c) = 0$, associa un elemento $(c)^{\mathcal{A}} \in A$.
  - Ad ogni simbolo di funzione $f \in \mathcal{F}$ con $ar(f) = n > 0$, associa una funzione n-aria $(f)^{\mathcal{A}} : A^n \rightarrow A$.
  - Ad ogni simbolo di predicato $p \in \Pi$, $ar(p) = n$, associa un relazione n-aria $(p)^{\mathcal{A}} \subseteq A^n$.

Esempio 2.5. Consideriamo il linguaggio dell'Esempio 2.2. Una possibile interpretazione $\mathcal{A}_1$ per tale linguaggio può essere ottenuta ponendo:

- l'insieme dei numeri naturali come dominio;
- $(z)^{\mathcal{A}_1} = 0$;
- associando la funzione successore al simbolo di funzione s;
- associando la relazione $\{\langle 0,0\rangle, \langle 1,2\rangle, \langle 2,4\rangle, \ldots\}$ al simbolo di predicato d.

Una altra possibile interpretazione $\mathcal{A}_2$ è la seguente:

- l'insieme $\{0,1\}$ come dominio;
- $(z)^{\mathcal{A}_2} = 0$;
- la funzione identità per il simbolo di funzione s;
- la relazione $\{\langle 0,1\rangle, \langle 1,0\rangle\}$ per simbolo di predicato d.

Abbiamo visto come assegnare un "significato" ai termini ground di un linguaggio tramite la nozione di interpretazione. Per gestire il caso generale di termini in cui occorrono delle variabili abbiamo bisogno dell'ulteriore concetto di *assegnamento*, introdotto dalla seguente definizione.

Definizione 2.8. Sia $\mathcal{A} = \langle A, (\cdot)^{\mathcal{A}} \rangle$ una struttura, t un termine, B un insieme di variabili tale che $B \supseteq vars(t)$. Sia inoltre $\sigma : B \rightarrow A$ un *assegnamento* di elementi del dominio A alle variabili in B.

Si definisce la *valutazione* del termine t dato l'assegnamento σ e la struttura $\mathcal{A}$ nel seguente modo:

$$(t\sigma)^{\mathcal{A}} = \begin{cases} \sigma(X) & \text{se } t \equiv X \text{ è una variabile in } \mathcal{V} \\ (c)^{\mathcal{A}} & \text{se } t \equiv c \text{ è un simbolo in } \mathcal{F}, \text{ con } ar(c) = 0 \\ (f)^{\mathcal{A}}((t_1\sigma)^{\mathcal{A}}, \ldots, (t_n\sigma)^{\mathcal{A}}) & \text{se } t \text{ è della forma } f(t_1, \ldots, t_n) \end{cases}$$

Definizione 2.9. Siano $\mathcal{A}$ una struttura e $p(t_1, \ldots, t_n)$ una formula atomica. Sia $B \supseteq vars(p(t_1, \ldots, t_n))$ un insieme di variabili. Sia inoltre $\sigma : B \to A$ un assegnamento di elementi del dominio A alle variabili in B. Allora il *valore di verità* della formula atomica è definito come segue:

$$V^{\mathcal{A}}(p(t_1, \ldots, t_n)) = \begin{cases} \mathsf{true} & \text{se } \langle (t_1\sigma)^{\mathcal{A}}, \ldots, (t_n\sigma)^{\mathcal{A}} \rangle \in (p)^{\mathcal{A}} \\ \mathsf{false} & \text{altrimenti} \end{cases}$$

Data una formula φ ed un assegnamento σ il fatto che la formula abbia valore di verità true rispetto alla interpretazione $\mathcal{A}$ viene denotato come $\mathcal{A} \models \varphi\sigma$ (letto "$\mathcal{A}$ modella $\varphi\sigma$"). Il fatto che invece la formula abbia valore false è denotato scrivendo $\mathcal{A} \not\models \varphi\sigma$.

Diamo di seguito la regola ricorsiva per stabilire se $\mathcal{A} \models \varphi\sigma$ nel caso in cui φ sia una generica formula.

Definizione 2.10. Sia $\mathcal{A}$ una struttura e σ un assegnamento.

- Se $p(t_1, \ldots, t_n)$ è una formula atomica, allora $\mathcal{A} \models p(t_1, \ldots, t_n)\sigma$ se e solo se vale $V^{\mathcal{A}}(p(t_1, \ldots, t_n)) = \mathsf{true}$;
- $\mathcal{A} \models \neg\varphi\sigma$ se e solo se $\mathcal{A} \not\models \varphi\sigma$;
- $\mathcal{A} \models (\varphi \vee \psi)\sigma$ se e solo se $\mathcal{A} \models \varphi\sigma$ oppure $\mathcal{A} \models \psi\sigma$;
- $\mathcal{A} \models (\exists X\varphi)\sigma$ se e solo se esiste un elemento $a \in A$ tale che $\mathcal{A} \models \varphi\sigma'$, dove l'assegnamento σ' é tale che $\sigma'(X) = a$ e $\sigma'(Y) = \sigma(Y)$ per ogni $Y \not\equiv X$.

Si osservi che nessun assegnamento di variabili ha effetto sul valore di verità degli enunciati. Questo perché gli assegnamenti operano solo sulle variabili libere. Conseguentemente, il significato della notazione $\mathcal{A} \models \varphi$, dato un generico enunciato φ, è univocamente determinato. Tale notazione viene in genere indicata con le tre equivalenti diciture:

- l'enunciato φ è vero nella interpretazione $\mathcal{A}$;
- l'enunciato φ è soddisfatto dalla interpretazione $\mathcal{A}$;
- l'interpretazione $\mathcal{A}$ è modello dell'enunciato φ.

La seguente definizione introduce i concetti analoghi nel caso delle formule generiche.

Definizione 2.11. Sia $\mathcal{A}$ una interpretazione. Sia inoltre φ una formula tale che $vars(\varphi) = \{X_1, \ldots, X_n\}$. Allora si scrive $\mathcal{A} \models \varphi$ se e solo se esistono $a_1, \ldots, a_n \in A$ tale che $\mathcal{A} \models \varphi\sigma$, dove l'assegnamento σ é tale che $\sigma(X_i) = a_i$ per ogni $i = 1, \ldots, n$.

Si osservi che, in base alla precedente definizione, abbiamo che $\mathcal{A} \models \varphi$ se e solo se $\mathcal{A} \models \exists X_1 \cdots \exists X_n \varphi$ (si noti che $\exists X_1 \cdots \exists X_n \varphi$ è un enunciato).

Nel caso in cui φ contenga variabili libere, si dice che φ è soddisfacibile in $\mathcal{A}$.

La seguente definizione introduce ulteriori nozioni connesse a quella di soddisfacibilità.

Definizione 2.12. Dato un enunciato φ si dice che:

- φ è *soddisfacibile* se esiste un'interpretazione $\mathcal{A}$ tale che $\mathcal{A} \models \varphi$;
- φ è *insoddisfacibile* (o *contraddittorio*) se non esiste alcuna interpretazione $\mathcal{A}$ tale che $\mathcal{A} \models \varphi$;
- φ è *valido* se per ogni interpretazione $\mathcal{A}$ vale che $\mathcal{A} \models \varphi$.

Definizione 2.13. Dato un insieme di enunciati Γ (anche detto *teoria*) si dice che:

- Γ è *soddisfacibile* se esiste un'interpretazione $\mathcal{A}$ tale che $\mathcal{A} \models \varphi$ per ogni $\varphi \in \Gamma$;

- Altrimenti Γ è detto *insoddisfacibile* (o *contraddittorio*).
- Una formula φ è una *conseguenza logica* di Γ se per ogni modello $\mathcal{A}$ di Γ si ha che se $\mathcal{A} \models \varphi$. Con abuso di notazione ciò si indica con $\Gamma \models \varphi$.
- Se $\Gamma = \emptyset$, in luogo di $\Gamma \models \varphi$ si scrive semplicemente $\models \varphi$. La notazione $\models \varphi$ esprime quindi la validità della formula φ.

Riportiamo il seguente risultato la cui dimostrazione si può trovare in qualunque testo di logica matematica (si veda ad esempio [**End72**], [**Men79**] o [**Sho67**]).

LEMMA 2.1. *Sia Γ un insieme di enunciati e φ un enunciato allora vale che:*

$$\Gamma \models \varphi \textit{ se e solo se } \Gamma \cup \{\neg\varphi\} \textit{ è insoddisfacibile.}$$

ESEMPIO 2.6. Sia Σ un alfabeto in cui $\mathcal{F} = \{0, s, +\}$ e $\Pi = \{=\}$, e sia $\mathcal{N} = \langle \mathbb{N}, (\cdot)^{\mathcal{N}} \rangle$ la struttura con funzione di interpretazione definita come segue:

$$\begin{array}{rcl} (0)^{\mathcal{N}} & = & 0 \\ (s)^{\mathcal{N}} & = & \lambda x.x+1 \quad \text{(la funzione successore)} \\ (+)^{\mathcal{N}} & = & \lambda x, y.x+y \quad \text{(la funzione somma)} \end{array}$$

Interpretiamo il simbolo predicativo $=$ con l'uguaglianza tra numeri naturali. Si ha che:

- $\mathcal{N} \models \forall Y \exists X (X = s(Y))$;
- $\mathcal{N} \not\models \forall Y \exists X (Y = s(X))$.

Per verificare che l'enunciato $\forall Y \exists X (Y = s(X))$ non è soddisfatto dalla interpretazione $\mathcal{N}$ si istanzi la variabile Y con la costante 0 (che, per quanto detto, viene interpretata in $\mathcal{N}$ con il numero naturale 0). Questo enunciato sarà invece soddisfatto da un'interpretazione $\mathcal{Z} = \langle \mathbb{Z}, (\cdot)^{\mathcal{Z}} \rangle$ che differisca da $\mathcal{N}$ solo per la scelta del dominio: scegliendo come dominio l'insieme dei numeri interi e $(\cdot)^{\mathcal{Z}}$ coincidente a $(\cdot)^{\mathcal{N}}$. In tal caso infatti si otterrebbe che $\mathcal{Z} \models \forall Y \exists X (Y = s(X))$. È interessante notare che in questo caso la formula è vera pur non essendo possibile esprimere, con termini di $T(\Sigma, \mathcal{V})$, gli elementi del dominio che la soddisfano (ad esempio al numero -1, che è l'oggetto del dominio che rende soddisfacibile $\exists X (0 = s(X))$, non corrisponde alcun termine del linguaggio).

2.1. Interpretazioni di Herbrand. Dato un alfabeto $\Sigma = (\mathcal{F}, \Pi, \mathcal{V})$, l'insieme di tutti i termini ground $T(\mathcal{F})$ è detto *Universo di Herbrand* di Σ. Un particolare genere di interpretazioni per Σ sono quelle in cui si sceglie come dominio proprio l'insieme $T(\mathcal{F})$. Tali interpretazioni, pur costituendo una ristretta classe tra tutte le interpretazioni possibili, godono di importanti proprietà.

DEFINIZIONE 2.14. Dato un alfabeto $\Sigma = (\mathcal{F}, \Pi, \mathcal{V})$, la *(pre)-interpretazione di Herbrand* $\mathcal{H} = \langle T(\mathcal{F}), (\cdot)^{\mathcal{H}} \rangle$ si definisce nel modo seguente:

- il dominio di $\mathcal{H}$ è l'Universo di Herbrand $T(\mathcal{F})$;
- la funzione di interpretazione di Herbrand mappa ogni termine ground in se stesso ed è così definita:

$$\left\{ \begin{array}{rcl} c^{\mathcal{H}} & = & c \\ f^{\mathcal{H}}(t_1, \cdots, t_n) & = & f(t_1^{\mathcal{H}}, \cdots, t_n^{\mathcal{H}}) \end{array} \right.$$

L'uso del prefisso "pre" è legato al fatto che non si fissa nessuna interpretazione dei simboli predicativi.

Si noti che nel caso in cui l'alfabeto sia privo di simboli di funzione (ovvero $\mathcal{F} = \emptyset$) allora anche l'universo di Herbrand ad esso associato è vuoto ($T(\mathcal{F}) = \emptyset$). Se $\mathcal{F} \neq \emptyset$ ma $\mathcal{F}$ contiene solo simboli di costante (ovvero di arità uguale a 0), allora $T(\mathcal{F})$ coincide, in pratica, con $\mathcal{F}$. Se, invece, esistono nell'alfabeto almeno un simbolo di costante ed un simbolo di funzione, allora $T(\mathcal{F})$ ha cardinalità infinita.

Esempio 2.7. Proviamo a descrivere l'universo di Herbrand dell'alfabeto dell'Esempio 2.6. $T(\mathcal{F})$ sarà il seguente insieme infinito:

$$T(\mathcal{F}) = \left\{ \begin{array}{cccc} 0 & s(0) & s(s(0)) & \cdots \\ 0+0 & 0+s(0) & 0+s(s(0) & \cdots \\ s(0+s(0)) & s(0)+s(s(0)) & \ddots & \\ \vdots & \vdots & & \ddots \end{array} \right\}$$

Completiamo questa (pre-)interpretazione di Herbrand interpretando il simbolo predicativo $=$ con l'uguaglianza sintattica tra termini ground. Così facendo si ha che:

$$\begin{array}{l} \mathcal{H} \models \forall Y \exists X (X = s(Y)), \\ \mathcal{H} \not\models \forall Y \exists X (Y = s(X)), \\ \mathcal{H} \not\models 0 + s(0) = s(0) + 0. \end{array}$$

Osserviamo che per la interpretazione (non di Herbrand) scelta nell'Esempio 2.6, avevamo invece che $\mathcal{N} \models 0 + s(0) = s(0) + 0$.

Come abbiamo visto, nella definizione di pre-interpretazione di Herbrand il dominio e l'interpretazione dei termini sono fissati univocamente. Nessuna posizione viene invece presa riguardo i simboli predicativi.

Definizione 2.15. Una *interpretazione* di Herbrand è una struttura che estende una pre-interpretazione di Herbrand con l'interpretazione dei simboli predicativi $p \in \Pi$, ovvero una funzione che dice, per ogni $p \in \Pi$ e per ogni $t_1, \ldots, t_n \in T(\mathcal{F})$ se $p(t_1, \ldots, t_n)$ ha valore di verità false o true.

Per completare la descrizione di una interpretazione di Herbrand basta pertanto aggiungere alla pre-interpretazione H_Σ un insieme di atomi, tali saranno gli atomi intesi veri:

$$I = \{p(t_1, \ldots, t_n) \,:\, p \in \Pi,\ t_1, \ldots, t_n \in T(\mathcal{F}),\ p(t_1, \ldots, t_n) \text{ è vero}\}$$

Inoltre, poiché, fissato l'alfabeto, per qualsiasi interpretazione di Herbrand sia il dominio che l'interpretazione dei termini sono prefissate, l'insieme I da solo identifica univocamente una interpretazione. Sarà quindi consuetudine indicare una particolare interpretazione di Herbrand descrivendo semplicemente l'insieme degli atomi veri.

Definizione 2.16. L'insieme di tutti gli atomi (ground) di un linguaggio con alfabeto $\Sigma = (\mathcal{F}, \Pi, \mathcal{V})$,

$$\mathcal{B}_{\Pi,\mathcal{F}} = \{p(t_1, \ldots, t_n) \,:\, p \in \Pi,\ t_1, \ldots, t_n \in T(\mathcal{F})\}$$

è detto *base di Herbrand*.

Come ulteriore convenzione assumiamo che dato un insieme di formule di interesse P, deduciamo dai simboli occorrenti in P l'alfabeto sottinteso. Qualora i due insiemi di simboli Π e $\mathcal{F}$ siano desunti in questo modo dalle formule di P, indicheremo la base di Herbrand semplicemente con $\mathcal{B}_P$.

DEFINIZIONE 2.17. Una interpretazione di Herbrand I è un *modello di Herbrand* per una teoria T se $I \models T$.

ESEMPIO 2.8. Dato il seguente insieme di enunciati P

$$\begin{array}{l} p(a)\,, \\ q(b)\,, \\ \forall X\,(p(f(X)) \leftarrow r(X)) \end{array}$$

avremo che $\mathcal{F}_P = \{a, b, f\}$ e $\Pi_P = \{p, q, r\}$.

Data la teoria T costituita dal solo enunciato:

$$\forall x(x = 0 \vee \exists y(y < x))$$

avremo che $\mathcal{F}_T = \{0\}$ e $\Pi_T = \{<, =\}$.

Supponiamo che sia data una teoria T costituita da enunciati in cui non occorrono simboli di costante. In questo caso desumere Π e $\mathcal{F}$ porterebbe a descrivere un universo di Herbrand vuoto. In casi come questo, qualora non vi sia nessun simbolo di costante esplicitamente presente in T, ne inseriremo esplicitamente uno "d'ufficio".

ESEMPIO 2.9. Dato il seguente insieme di enunciati T

$$\{\ \exists X\,(q(X) \rightarrow q(h(X))), \quad \forall X \forall Y\,(p(g(X,Y)) \vee r(Y, h(Y)))\ \},$$

desumendo $\mathcal{F}_T$ dai simboli esplicitamente menzionati in T avremmo $\mathcal{F}_T = \emptyset$. Inseriamo quindi nell'universo di Herbrand una costante, diciamo c. Così facendo otteniamo

$$T(\mathcal{F}_T) = \{h(c), g(c,c), h(h(c)), h(g(c,c)), g(h(c), c), \ldots\}.$$

ESEMPIO 2.10. A partire dal seguente insieme di enunciati P

$$\begin{array}{l} p(a)\,, \\ r(b)\,, \\ \forall X\,(q(X) \leftarrow p(X)) \end{array}$$

Desumiamo che $\Pi_P = \{p, q, r\}$ e $\mathcal{F}_P = \{a, b\}$. Conseguentemente avremo $T(\mathcal{F}_P) = \{a, b\}$ e $\mathcal{B}_P = \{p(a), p(b), q(a), q(b), r(a), r(b)\}$.

Una possibile interpretazione di Herbrand è $I = \{p(a), r(b), q(b)\}$. Si noti che I non è un modello di P in quanto l'ultimo enunciato non è vero in I.

L'interpretazione di Herbrand $I_2 = B_P$ è invece un modello di P.

L'insieme $I_3 = \{p(a), r(b), q(a), q(b)\}$, sottoinsieme proprio di I_2, è ancora di un modello per P.

L'interpretazione $I_4 = \{p(a), r(b), q(a)\}$ (sottoinsieme proprio di I_3) è ancora modello di P. Considerando i sottoinsiemi propri di I_4 si osserva che I_4 sembra essere il "più piccolo" modello possibile. È infatti un modello *minimo*. Torneremo su queste questioni e sulla importanza dei modelli minimi nel Capitolo 6 (in particolare nella Sezione 2).

Esempio 2.11. Dato $T = \{p(a), \exists X \neg p(X)\}$, da cui possiamo ricavare $F = \{a\}$ e $B_T = \{p(a)\}$, abbiamo due possibili interpretazioni di Herbrand:

(1) $I_1 = \emptyset$;
(2) $I_2 = B_T = \{p(a)\}$.

Tuttavia I_1 non è un modello di T perché soddisfa l'enunciato $\exists X \neg p(X)$ ma non soddisfa l'enunciato $p(a)$. Poiché però l'unico elemento del dominio è a, nemmeno I_2 è un modello di T. Quindi T non ammette modelli di Herbrand.

Esistono però modelli per T. Questi si ottengono considerando una interpretazione di Herbrand per un alfabeto (e quindi un linguaggio) più ampio di quello che si desume dall'insieme T. Più in generale si può costruire un modello per T cambiando il dominio della interpretazione. Possiamo ad esempio porre il dominio pari a $\{a, b\}$ e $M = \{p(a)\}$. Si noti che questo modello M non è un modello di Herbrand.

3. Sostituzioni

Definizione 2.18. Una *sostituzione* è una funzione $\sigma : \mathcal{V} \longrightarrow T(\mathcal{F}, \mathcal{V})$ tale che il suo dominio $dom(\sigma) = \{X \in \mathcal{V} : \sigma(X) \not\equiv X\}$ è un insieme finito.

Data una sostituzione σ, se $dom(\sigma) = \{X_1, \ldots, X_n\}$ allora σ può essere rappresentata come $[X_1/t_1, \ldots, X_n/t_n]$ dove $X_i \neq X_j$ per $i \neq j$ e $t_i \equiv \sigma(X_i)$ per $i = 1, \ldots, n$. Ogni scrittura della forma "X/t" è detta *binding*.

L'insieme $vars(t_1, \ldots, t_n) = ran(\sigma)$, è detto *rango* di σ. Si definisce $vars(\sigma) = dom(\sigma) \cup ran(\sigma)$ (è l'insieme di tutte le variabili "trattate" dalla sostituzione). Se $dom(\sigma) = \emptyset$, la sostituzione è detta sostituzione vuota ed è denotata con ϵ.

Definizione 2.19. Sia data una sostituzione $\sigma = [X_1/t_1, \ldots, X_n/t_n]$.

- Se $t_1, \ldots, t_n$ sono tutte variabili allora σ è detta *sostituzione di variabili.*
- Se $t_1, \ldots, t_n$ sono tutte variabili distinte, allora la sostituzione è detta *rinomina* (renaming).
- Se inoltre σ è una rinomina tale che $dom(\sigma) = ran(\sigma)$, ovvero $\{X_1, \ldots, X_n\} = \{t_1, \ldots, t_n\}$, allora σ è detta *variante* (o permutazione di variabili).
- Se $t_1, \ldots, t_n$ sono tutti ground allora σ è detta sostituzione *ground.*

Esempio 2.12. $[X/a, Y/b]$, e $[X/f(X, Y), Y/g(Z, X)]$ sono sostituzioni. In particolare, $[X/Z, Y/Z]$ è una sostituzione di variabili (ma non una rinomina); $[X/Z, Y/W]$ è un rinomina; $[X/Y, Y/X]$ è una variante.

Il seguente lemma enuncia una proprietà delle sostituzioni che risulterà utile in seguito.

Lemma 2.2. *Sia σ una rinomina allora esiste una variante $\sigma' \supseteq \sigma$ (σ' estende σ) tale che $dom(\sigma') = ran(\sigma') = vars(\sigma)$.*

Dim. Siano $A = dom(\sigma) \setminus ran(\sigma)$ e $B = ran(\sigma) \setminus dom(\sigma)$.

Essendo σ rinomina, è iniettiva. La finitezza del dominio e l'iniettività garantiscono che la cardinalità di A sia uguale alla cardinalità di B; di conseguenza anche la cardinalità di $(A \setminus B)$ sarà uguale alla cardinalità di $(B \setminus A)$.

A questo punto, possiamo estendere il renaming σ con una qualsiasi funzione biiettiva $g : (B \setminus A) \longrightarrow (A \setminus B)$ ottenendo σ' che è una variante per costruzione. □

ESEMPIO 2.13. Riferendosi alla dimostrazione del lemma precedente, si ragioni pensando ad un caso specifico, per esempio con $\sigma = [X_3/X_1, X_4/X_2, X_5/X_3, X_6/X_4, X_7/X_5]$. In tale condizione possiamo porre $g = [X_1/X_6, X_2/X_7]$ per ottenere $\sigma' = \sigma \cup [X_1/X_6, X_2/X_7]$.

La definizione seguente introduce la nozione di applicazione di una sostituzione ad un termine.

DEFINIZIONE 2.20. L'*applicazione di una sostituzione* σ ad un termine t, $t\sigma$ è definita nel modo seguente:

$$t\sigma = \begin{cases} \sigma(X) & \text{se } t \equiv X \\ f(t_1\sigma, \dots, t_n\sigma) & \text{se } t \equiv f(t_1, \dots, t_n) \end{cases}$$

Si osservi che se $t \equiv c$ allora $c\sigma = c$ per ogni sostituzione σ. Più in generale, si osservi che se $vars(t) \cap dom(\sigma) = \emptyset$ allora $t\sigma = t$.

ESEMPIO 2.14. Se $\sigma = [X/f(X, Y), Y/g(X)]$ allora

(1) $X\sigma = f(X, Y)$

(2) $$\begin{aligned} h(f(X, Y), g(Y))\sigma &= h(f(X, Y)\sigma, g(Y)\sigma) \\ &= h(f(X\sigma, Y\sigma), g(Y\sigma)) \\ &= h(f(f(X, Y), g(X)), g(g(X))) \end{aligned}$$

DEFINIZIONE 2.21. Un termine s è un *istanza* di un termine t se esiste una sostituzione σ tale che $s = t\sigma$. Si può, senza perdere di generalità assumere che $dom(\sigma) \subseteq vars(t)$.

ESEMPIO 2.15. Il termine $f(a, a)$ è istanza di $f(X, Y)$, poiché $f(a, a) = f(X, Y)[X/a, Y/a]$. Si noti che avremmo anche potuto scrivere $f(a, a) = f(X, Y)[X/a, Y/a, Z/b]$. Quindi la stessa istanza può essere ottenuta utilizzando differenti sostituzioni.

Si noti anche che la relazione "essere istanza di" non è in generale simmetrica. Ad esempio il termine $f(Z, Z)$ è istanza di $f(X, Y)$ ma non viceversa.

DEFINIZIONE 2.22. Un termine s è una *variante* del termine t se esiste σ, variante, tale che $s = t\sigma$.

Una analoga definizione può essere data per la nozione di *rinomina* di un termine.

ESEMPIO 2.16. Il termine $f(X, Y)$ è una variante del termine $f(Y, X)$, poiché $f(X, Y) = f(Y, X)[X/Y, Y/X]$ e la sostituzione $[X/Y, Y/X]$ é una variante. Inoltre $f(X, Y)$ è anche una variante di $f(X, Z)$, poiché $f(X, Y) = f(X, Z)[Z/Y, Y/Z]$.

Invece $f(X, X)$ non è una variante di $f(X, Y)$. Infatti si ha che $f(X, X) = f(X, Y)[Y/X]$ ma la sostituzione $\sigma = [Y/X]$ non è una variante. Inoltre per ottenere una variante a partire da σ bisognerebbe aggiungervi un binding del tipo $[X/?]$, cioé che sostituisca un termine ad X. Tuttavia, così facendo, non si otterrebbe più $f(X, X)$ istanziando $f(X, Y)$.

Si osservi che se s è variante di t, allora t è variante di s (e viceversa). Si può infatti dimostrare (si veda Esercizio 2.1) che per ogni sostituzione variante θ esiste esattamente una sostituzione γ tale che $\theta\gamma = \gamma\theta = \varepsilon$. Denoteremo questa γ con θ^{-1}. Pertanto, se $s = t\theta$, con t variante, allora vale che $s\theta^{-1} = t\theta\theta^{-1} = t$.

DEFINIZIONE 2.23. Dato un insieme di variabili $V \subseteq \mathcal{V}$ ed una sostituzione σ definiamo la sostituzione $\sigma\mid_V$ (letto, σ *ristretto a* V) come segue:

$$\sigma\mid_V(X) = \begin{cases} \sigma(X) & \text{se } X \in V \\ X & \text{altrimenti.} \end{cases}$$

Date due sostituzioni θ e η definiamo la loro *composizione*, indicata con $\theta\eta$, come la sostituzione tale che: $(\theta\eta)(X) = \eta(X\theta)$. In notazione postfissa, scriviamo semplicemente $X(\theta\eta) = (X\theta)\eta$.

Il seguente lemma supporta la introduzione della operazione di composizione.

LEMMA 2.3. *Siano date due sostituzioni*

$$\begin{array}{rcl} \theta & = & [X_1/r_1, \ldots, X_m/r_m, Y_1/s_1, \ldots, Y_n/s_n] \quad e \\ \eta & = & [Z_1/t_1, \ldots, Z_p/t_p, Y_1/v_1, \ldots, Y_n/v_n] \end{array}$$

ove per ogni i e j, $X_i \neq Z_j$. Allora si ha che la loro composizione $\theta\eta$ si ottiene da

$$[X_1/r_1\eta, \ldots, X_m/r_m\eta, Y_1/s_1\eta, \ldots, Y_n/s_n\eta, Z_1/t_1, \ldots, Z_p/t_p]$$

rimuovendo i bindings del tipo $[V/V]$.

DIM. Esercizio. □

L'Esercizio 2.1 e il successivo Lemma 2.4 illustrano alcune utili proprietà delle sostituzioni.

ESERCIZIO 2.1. Dimostrare le seguenti proprietà delle sostituzioni:

(1) Per θ sostituzione qualsiasi si ha $\theta\varepsilon = \varepsilon\theta = \theta$, dove ε è la sostituzione vuota.
(2) Per ogni variante θ esiste esattamente una sostituzione γ tale che $\theta\gamma = \gamma\theta = \varepsilon$. Questa γ la chiamiamo θ^{-1}. Inoltre γ è una variante.
(3) Sia t un termine. Se $t = t\theta$ allora $\theta|_{vars(t)} = \varepsilon$.
(4) Date θ e η sostituzioni, se $\theta\eta = \varepsilon$ allora θ e η sono varianti.
(5) La relazione su termini "s è variante di t" è una relazione di equivalenza (ovvero è riflessiva, simmetrica e transitiva).

LEMMA 2.4. *Siano θ, η e γ sostituzioni e s un termine. Allora valgono le seguenti condizioni:*

(1) $(s\theta)\eta = s(\theta\eta)$;
(2) $(\theta\eta)\gamma = \theta(\eta\gamma)$.

DIM. La seconda proprietà è immediata ed è lasciata come esercizio. Dimostriamo la prima procedendo per induzione sulla struttura del termine s.
Caso base: Se s è una variabile, $s \equiv X$, allora la tesi discende per la definizione di composizione.
Passo induttivo: Se s è un termine composto, $s \equiv f(t_1, \ldots, t_n)$ allora si ha:

$$\begin{array}{rcll} (f(t_1, \ldots, t_n)\theta)\eta & = & f(t_1\theta, \ldots, t_n\theta)\eta & \text{per definizione di applicazione} \\ & = & f((t_1\theta)\eta, \ldots, (t_n\theta)\eta) & \text{per definizione di applicazione} \\ & = & f(t_1(\theta\eta), \ldots, t_n(\theta\eta)) & \text{per ipotesi ind. e per l'assioma dell'eguaglianza} \\ & = & f(t_1, \ldots, t_n)(\theta\eta) & \text{per definizione di applicazione} \end{array}$$

□

LEMMA 2.5 (Varianti). *Siano s e t due termini.*

(1) *s è variante di t se e solo se s è istanza di t e t è istanza di s,*
(2) *se s è variante di t allora $s = t\theta$ per almeno una variante θ tale che $vars(\theta) \subseteq vars(s) \cup vars(t)$.*

DIM. Dimostrando il punto (1), viene automaticamente dimostrato anche l'altro.
($\Rightarrow$) Se s è variante di t, allora per definizione esiste θ variante tale che $s = t\theta$.
Sappiamo (punto (2) dell'Esercizio 2.1) che data una variante esiste un'unica sostituzione che è la sua inversa, cioè tale che $\theta\theta^{-1} = \theta^{-1}\theta = \varepsilon$. Nel nostro caso abbiamo $s = t\theta$ dove θ è una biiezione da variabili in variabili. Se consideriamo θ^{-1} possiamo dimostrare per induzione sulla struttura dei termini che $s = t\theta$ implica che $s\theta^{-1} = t$.

($\Leftarrow$) Sappiamo che esistono θ e γ tali che $s = t\theta$ e $t = s\gamma$. Possiamo assumere, senza perdita di generalità, che

- $dom(\theta) \subseteq vars(t)$,
- $ran(\theta) \subseteq vars(s)$,
- $dom(\gamma) \subseteq vars(s)$,
- $ran(\gamma) \subseteq vars(t)$.

Sostituendo la s nella definizione di t, ottengo $t = t\theta\gamma$. Ciò significa che (si veda anche l'Esercizio 2.1):

$$\theta\gamma|_{vars(t)} = \varepsilon \tag{3.1}$$

Mostriamo che θ è una sostituzione di variabili iniettiva.

- Per assurdo sia $\theta(X) = f(t_1, \ldots, t_n)$. Allora, per definizione di sostituzione $X\theta\gamma$ avrà la forma $f(t_1\gamma, \ldots, t_n\gamma)$. Si avrebbe dunque che $X\theta\gamma \not\equiv X$ contraddicendo (3.1), poiché $X \in dom(\theta) \subseteq vars(t)$. Pertanto θ è una sostituzione di variabili.
- Mostriamo ora che è iniettiva. Siano $X, Y \in vars(t)$, $X \neq Y$. Per assurdo, sia $X\theta = Y\theta$. Allora si avrebbe $X\theta\gamma = Y\theta\gamma$. Ma ciò, per (3.1), significherebbe che $X = Y$. Assurdo.

Per il Lemma 2.2 una sostituzione di variabili iniettiva, ovvero una rinomina, può essere estesa ad una variante $\theta' = \theta \cup [V_1/W_1, \ldots, V_m/W_m]$. Per come viene costruita θ', $V_1, \ldots, V_m$ sono tutte variabili di s che non sono in t e sono presenti in $ran(\theta)$. Potrebbero infatti esserci anche variabili di t presenti in $ran(\theta)$ e non in $dom(\theta)$. Ma questo significherebbe che due variabili diverse di t vengono rese uguali da θ. Ma ciò porterebbe ad un assurdo per l'iniettività di θ su $vars(t)$. Pertanto, da $s = t\theta$ consegue che $s = t\theta'$ in quanto nuovi binding V_i/W_i che riguardano le variabili in s e non in t non producono effetto in quanto la sostituzione è applicata a t. Per costruzione si ha quindi che $vars(\theta') \subseteq vars(s) \cup vars(t)$ (come nell'enunciato del punto (2)). □

Sappiamo che la stessa istanza può essere ottenuta applicando differenti sostituzioni (vedi Esempio 2.15). Ci si può chiedere se tra esse vi sia una sostituzione "preferibile" alle altre. Più in generale in quanto segue vedremo che è possibile introdurre una qualche nozione di ordine tra le sostituzioni in dipendenza di quanto specifici siano i loro binding. Più formalmente abbiamo la seguente definizione:

DEFINIZIONE 2.24. Date θ e τ sostituzioni, θ è *più generale* di τ (ciò si indica con $\theta \leq \tau$) se esiste una sostituzione η tale che $\tau = \theta\eta$.

ESEMPIO 2.17. Alcune proprietà delle sostituzioni:

(1) Consideriamo $[X/Y]$ e $[X/a, Y/a]$. La seconda sostituzione si può ottenere applicando $[Y/a]$ alla prima: $[X/Y][Y/a] = [X/a, Y/a]$, quindi $[X/Y] \leq [X/a, Y/a]$.
(2) $[X/Y] \leq [X/Z, Y/Z]$, infatti $[X/Y][Y/Z] = [X/Z, Y/Z]$. Proviamo a chiederci se valga anche il viceversa. Supponiamo che esista una sostituzione $\eta = [Z/t]$ tale che $[X/Z, Y/Z]\eta = [X/Y]$, ovviamente con $X \not\equiv Y$, $X \not\equiv Z$ e $Y \not\equiv Z$. Applicando il Lemma 2.3 otteniamo che $[X/Z, Y/Z][Z/t] = [X/t, Y/t, Z/t]$. Quindi dovrebbe valere $[X/t, Y/t, Z/t] = [X/Y]$. Ma ciò sarebbe possibile solamente se $t \equiv Y$ e $t \equiv Z$, e quindi $Y \equiv Z$. Assurdo. Il punto chiave è che non è possibile trovare una sostituzione η tale che Z non compaia nel dominio della sostituzione $[X/Z, Y/Z]\eta$.
(3) La sostituzione vuota ε è più generale di ogni sostituzione. Prendendo una generica sostituzione θ si ha infatti $\theta = \varepsilon\theta$.
    Si può pensare a una forma di ordinamento (pre-ordine) tra sostituzioni in base alla relazione $\leq$, dove nel bottom c'è ε.
(4) La sostituzione $[X/f(Y, Z)]$ è più generale di $[X/f(a, a)]$? Sembrerebbe che applicando $[Y/a, Z/a]$ alla prima fossimo sulla buona strada. Tuttavia la composizione che otteniamo è $[X/f(a, a), Y/a, Z/a] \neq [X/f(a, a)]$. La risposta alla domanda è quindi negativa e possiamo dedurre che le due sostituzioni non sono confrontabili rispetto a $\leq$.
(5) Date le sostituzioni $[X/Y, Y/Z, Z/X]$ e $[X/Z, Y/X, Z/Y]$ abbiamo che
    $$[X/Y, Y/Z, Z/X] \leq [X/Z, Y/X, Z/Y].$$
    Infatti esiste una sostituzione che composta con la prima produce la seconda:
    $$[X/Y, Y/Z, Z/X][Y/Z, Z/X, X/Y] = [X/Z, Y/X, Z/Y].$$
    Esiste anche una sostituzione che composta con la seconda produce la prima:
    $$[X/Z, Y/X, Z/Y][Z/Y, X/Z, Y/X] = [X/Y, Y/Z, Z/X].$$
    Esistono quindi sostituzioni che sono reciprocamente una più generale dell'altra.

Si osservi che $\leq$ è una relazione riflessiva, transitiva, ma non antisimmetrica. Quindi non è una relazione d'ordine parziale, più semplicemente una relazione di *preordine*. Il seguente lemma lega la nozione di istanza con quella di preordine appena descritta.

LEMMA 2.6. *Se $\theta \leq \tau$, allora, per ogni termine t, $t\tau$ è istanza di $t\theta$.*

DIM. Consideriamo $\theta \leq \tau$. Allora, per definizione, esiste η tale che $\tau = \theta\eta$. Prendendo un termine t qualsiasi abbiamo: $t\tau = t(\theta\eta) = (t\theta)\eta$. Questo dimostra che $t\tau$ è istanza di $t\theta$. □

Dati due termini t_1 e t_2 ci si può chiedere se esista un terzo termine t che abbia sia t_1 che t_2 come istanze (in altre parole, un termine più generale di entrambi i termini dati). La risposta a questo quesito è sempre positiva. Per rendersene conto basta infatti osservare che una variabile può essere istanziata a qualsiasi termine. Tuttavia ci si può chiedere se sia possibile determinare un termine t che sia il più specifico tra tutti i possibili candidati. Anche in questo caso la risposta è affermativa. Vediamo un esempio.

Esempio 2.18. Dati i due termini $t_1 = h(f(a), X, f(g(g(X))))$ e $t_2 = h(b, b, f(g(b)))$, si determini (a meno di rinomine delle variabili) il termine più specifico tra quelli più generali sia di t_1 che di t_2.

Il termine cercato è $t = h(A, X, f(g(B)))$ (o una sua qualsiasi rinomina).

Lemma 2.7. *Siano θ e τ due sostituzioni. Allora $\theta \leq \tau$ e $\tau \leq \theta$ se e solo se esiste una sostituzione γ tale che γ è una variante e $vars(\gamma) \subseteq vars(\theta) \cup vars(\tau)$ e $\tau = \theta\gamma$.*

Dim. Il verso ($\leftarrow$) è banale. Per l'altro verso siano η e γ tali che:

$$\theta = \tau\eta \text{ e } \tau = \theta\gamma.$$

Allora si ha che $\theta = \theta\gamma\eta$. Pertanto γ e η sono varianti e $\gamma = \eta^{-1}$ (e valgono i vincoli sulle variabili). □

4. Esercizi

Alcuni dei seguenti esercizi sono stati tratti da vari libri di testo ai quali si rimanda per ulteriore materiale di studio (tra le varie fonti: [**Llo87, End72, Men79, Sho67**])

Esercizio 2.2. Supponiamo di scegliere un alfabeto in cui $\Pi = \{n, p, r\}$ e $\mathcal{F} = \{z\}$. Interpretiamo ora i termini del linguaggio assumendo che il predicato monadico $n(\cdot)$ rappresenti la proprietà "è un numero"; il predicato monadico $p(\cdot)$ rappresenti la proprietà "è interessante"; il predicato binario $r(\cdot, \cdot)$ rappresenti la proprietà "è più piccolo di"; il simbolo di costante z rappresenti lo zero.
Scrivere delle formule logiche che rendano le seguenti frasi del linguaggio naturale:

(1) Zero è più piccolo di ogni numero
(2) Se un numero è interessante allora zero è interessante
(3) Se un numero è interessante allora tutti i numeri sono interessanti
(4) Nessun numero è più piccolo di zero
(5) Ogni numero non interessante e tale che tutti i numeri più piccoli di lui sono interessanti è certamente interessante
(6) C'è almeno un numero tale che tutti i numeri siano più piccoli di lui
(7) C'è almeno un numero tale che nessun numero sia più piccolo di lui
(8) C'è esattamente un numero tale che tutti i numeri siano più piccoli di lui
(9) C'è esattamente un numero tale che nessun numero sia più piccolo di lui
(10) Non c'è nessun numero tale che tutti i numeri siano più piccoli di lui
(11) Non c'è nessun numero tale che nessun numero sia più piccolo di lui

Esercizio 2.3. Si considerino le seguenti tre formule:

(1) $\forall X \forall Y \forall Z\, (p(X, Y) \rightarrow (p(Y, Z) \rightarrow p(X, Z)))$
(2) $\forall X \forall Y\, (p(X, Y) \rightarrow (p(Y, X) \rightarrow X = Y))$
(3) $\forall X \exists Y\, p(X, Y) \rightarrow \exists Y \forall X\, p(X, Y)$

(dove $=$ si intende sempre interpretato con la relazione identità nel dominio della interpretazione). Si dimostri che nessuna di esse è logica conseguenza delle altre due. [SUGGERIMENTO: per ognuna delle tre formule, si trovi una interpretazione che non la renda vera ma che sia modello per le altre due]

Esercizio 2.4. Si consideri l'alfabeto in cui $\Pi = \{p, q, r\}$ e $\mathcal{F} = \{a, b, s\}$. Inoltre si assuma fissata la seguente interpretazione:

- Il dominio è l'insieme dei numeri naturali $\mathbb{N}$;
- le costanti a e b sono rispettivamente interpretate con i numeri 0 e 1;
- al simbolo funzionale s con arità 1 si associa la funzione dai naturali ai naturali $n \mapsto n + 1$;
- al simbolo di predicato binario p si associa la relazione $\{\langle n, m\rangle : n > m\} \subset \mathbb{N} \times \mathbb{N}$;
- al simbolo di predicato monadico q si associa la relazione $\{n : n > 0\} \subset \mathbb{N}$;
- al simbolo di predicato binario r si associa la relazione $\{\langle n, m\rangle : n \text{ divide } m\} \subset \mathbb{N} \times \mathbb{N}$;

Per ognuna delle seguenti formule si determini il valore di verità rispetto a tale interpretazione (si giustifichi la risposta).

(1) $\forall X \exists Y\, p(X, Y)$
(2) $\exists X \forall Y\, p(X, Y)$
(3) $p(s(a), b)$
(4) $\forall X\, (q(X) \rightarrow p(X, a))$
(5) $\forall X\, p(s(X), X)$
(6) $\forall X \forall Y\, (r(X, Y) \rightarrow \neg p(X, Y))$
(7) $\forall X (\exists Y\, p(X, Y) \vee r(s(b), s(X)) \rightarrow q(X))$

Esercizio 2.5. Dire (giustificando le risposte) se le seguenti formule sono o meno valide.

(1) $\forall X \exists Y\, q(X, Y) \rightarrow \exists Y \forall X\, q(X, Y)$
(2) $\exists Y \forall X\, q(X, Y) \rightarrow \forall X \exists Y\, q(X, Y)$

Esercizio 2.6. Considerando la formula

$$\Big(\forall X\, p_1(X, X) \wedge \forall X \forall Y \forall Z\, \big((p_1(X, Y) \wedge p_1(Y, Z)) \rightarrow p_1(X, Z)\big) \wedge \forall X \forall Y\, \big(p_1(X, Y) \vee p_1(Y, X)\big)\Big) \rightarrow \exists Y \forall X\, p_1(Y, X)$$

(1) Mostrare che ogni interpretazione in cui il dominio è un insieme finito, è un modello della formula;
(2) Trovare una interpretazione che non sia un modello della formula.

Esercizio 2.7. Per ognuna delle seguenti coppie di termini t_1 e t_2 si fornisca, se esiste, una sostituzione θ tale che $t_1\theta$ e $t_2\theta$ siano sintatticamente uguali.

(1) $t_1 :\ h(f(Y), W, g(Z))$ e $t_2 :\ h(X, X, V)$
(2) $t_1 :\ h(f(Y), W, g(Z))$ e $t_2 :\ h(X_1, X_2, X_1)$
(3) $t_1 :\ h(f(Y), W, g(Z))$ e $t_2 :\ h(X_1, X_2, X_3)$
(4) $t_1 :\ k(a, X, f(g(Y)))$ e $t_2 :\ k(Z, h(Z, W), f(W))$
(5) $t_1 :\ f(a, X, h(g(Z)))$ e $t_2 :\ f(Z, h(Y), h(Y))$
(6) $t_1 :\ h(X, X)$ e $t_2 :\ h(Y, f(Y))$
(7) $t_1 :\ g(f(X), h(Y), b)$ e $t_2 :\ g(f(g(b, b)), h(b), Y)$
(8) $t_1 :\ f(g(X, a), Z)$ e $t_2 :\ f(Z, Y)$
(9) $t_1 :\ h(f(X), f(a), g(X, Y))$ e $t_2 :\ h(f(g(W, Y)), W, Z)$
(10) $t_1 :\ h(f(X), f(a), g(X, Y))$ e $t_2 :\ h(f(g(W, Y)), W, a)$
(11) $t_1 :\ g(f(a, Y), Z)$ e $t_2 :\ g(Z, Z)$
(12) $t_1 :\ h(g(f(a, Y), Z))$ e $t_2 :\ h(g(Z, b))$

(13) $t_1 :\ h(g(f(a,Y),Z))$ e $t_2 :\ h(g(Z,W))$
(14) $t_1 :\ g(f(a,b),h(X,Y))$ e $t_2 :\ g(f(Z,b),h(b,b))$
(15) $t_1 :\ g(f(a,X),h(X,b))$ e $t_2 :\ g(f(a,b),h(a,b))$
(16) $t_1 :\ g(f(a,X),h(X,b))$ e $t_2 :\ g(f(a,b),h(b,b))$
(17) $t_1 :\ g(f(a,X),h(Y,b))$ e $t_2 :\ g(Z,Y)$
(18) $t_1 :\ g(f(a,X),h(Y,b))$ e $t_2 :\ g(Z,X)$

Esercizio 2.8. Dire quale sostituzione $\rho = \theta\eta$ si ottiene componendo le seguenti due sostituzioni $\theta = \big[X/f(Y), Y/Z\big]$ e $\eta = \big[X/a, Y/b, Z/Y\big]$.

Esercizio 2.9. Dire quale sostituzione $\rho = \theta\eta$ si ottiene componendo le seguenti due sostituzioni $\theta = \big[A/f(B), B/C\big]$ e $\eta = \big[A/a, C/b, D/B\big]$. Applicare quindi la sostituzione ρ al termine $t = h(f(A), g(B), D)$.

Esercizio 2.10 (Forma normale congiuntiva). Una formula è detta in *forma normale congiuntiva prenessa* se è della forma $Q_1\, X_1 \dots Q_n\, X_n\, \phi$, dove ogni Q_i è un quantificatore ($\forall$ oppure $\exists$) e ϕ è una congiunzione di disgiunzioni di letterali.
Si supponga che φ sia una formula in cui tutte le variabili legate sono distinte tra loro e nessuna variabile occorre sia libera che legata. Dimostrare che una tale formula può essere trasformata in una formula in forma normale congiuntiva prenessa logicamente equivalente, utilizzando le seguenti regole di riscrittura.

(1) Rimpiazzare le occorrenze della forma $\chi \rightarrow \psi$ con $\neg\chi \vee \psi$;
rimpiazzare le occorrenze della forma $\chi \leftrightarrow \psi$ con $(\neg\chi \vee \psi) \wedge (\chi \vee \neg\psi)$.
(2) Rimpiazzare $\neg\forall X\, \chi$ con $\exists X\, \neg\chi$;
rimpiazzare $\neg\exists X\, \chi$ con $\forall X\, \neg\chi$;
rimpiazzare $\neg(\chi \vee \psi)$ con $\neg\chi \wedge \neg\psi$;
rimpiazzare $\neg(\chi \wedge \psi)$ con $\neg\chi \vee \neg\psi$;
rimpiazzare $\neg\neg\chi$ con χ;
procedendo fino a che tutti i connettivi $\neg$ precedono gli atomi.
(3) Rimpiazzare $\exists X\, \chi \vee \psi$ con $\exists X\, (\chi \vee \psi)$;
rimpiazzare $\chi \vee \exists X\, \psi$ con $\exists X\, (\chi \vee \psi)$;
rimpiazzare $\forall X\, \chi \vee \psi$ con $\forall X\, (\chi \vee \psi)$;
rimpiazzare $\chi \vee \forall X\, \psi$ con $\forall X\, (\chi \vee \psi)$;
rimpiazzare $\exists X\, \chi \wedge \psi$ con $\exists X\, (\chi \wedge \psi)$;
rimpiazzare $\chi \wedge \exists X\, \psi$ con $\exists X\, (\chi \wedge \psi)$;
rimpiazzare $\forall X\, \chi \wedge \psi$ con $\forall X\, (\chi \wedge \psi)$;
rimpiazzare $\chi \wedge \forall X\, \psi$ con $\forall X\, (\chi \wedge \psi)$;
procedendo fino a che tutti i quantificatori si trovano nel prefisso della formula.
(4) Rimpiazzare $(\psi_1 \wedge \psi_2) \vee \chi$ con $(\psi_1 \vee \chi) \wedge (\psi_2 \vee \chi)$;
rimpiazzare $\psi \vee (\chi_1 \wedge \chi_2)$ con $(\psi \vee \chi_1) \wedge (\psi \vee \chi_2)$;
fino a che non si raggiunge la forma normale congiuntiva prenessa.

Esercizio 2.11. Dire quale è nel caso pessimo la complessità della trasformazione descritta nell'esercizio precedente (valutare la complessità di una formula come il numero di occorrenze di simboli funzionali, predicativi e di variabile).

CAPITOLO 3

Programmazione con clausole definite

Questo capitolo vuol essere una introduzione alla metodologia di programmazione dichiarativa basata su programmi logici. Nel resto del testo, quando avremo studiato dettagliatamente la semantica di tali linguaggi, sarà possibile una comprensione più profonda sia dei concetti qui introdotti sia del significato preciso che viene attribuito ai programmi logici.

Abbiamo visto nel capitolo precedente che una teoria è un insieme di formule. Analizzeremo ora il caso di particolari insiemi in cui le formule sono disgiunzioni di letterali. In questo caso è usuale utilizzare il termine *programma* in luogo del generico "teoria".

DEFINIZIONE 3.1. Sia dato un alfabeto $\Sigma = \langle \Pi, \mathcal{F}, \mathcal{V} \rangle$, allora:

- se $H, A_1, \ldots, A_n$ sono atomi, $n \geq 0$, allora $H \leftarrow A_1, \ldots, A_n$ è una *regola.* L'atomo H è detto *testa* della regola. Gli atomi $A_1, \ldots, A_n$ invece costituiscono il *corpo* della regola.
- Se $n = 0$, la regola $H \leftarrow$ è detto anche *fatto.*
- Se $A_1, \ldots, A_n$ sono atomi, allora $\leftarrow A_1, \ldots, A_n$ è un *goal* (o *query*). Se inoltre $n = 0$, indichiamo con $\leftarrow \Box$ il goal vuoto.
- Un *programma* è un insieme finito di regole.

Diamo una lettura intuitiva della regola

$$H \leftarrow A_1, \ldots, A_n.$$

La virgola è da intendersi come la congiunzione logica ($\wedge$). Pertanto la regola asserisce che quando $A_1, \ldots, A_n$ sono vere, deve essere vero anche H. Poiché $P \rightarrow Q$ è equivalente a $\neg P \vee Q$, allora la regola $H \leftarrow A_1, \ldots, A_n$ è equivalente ad una disgiunzione di letterali con esattamente un letterale positivo (ovvero un atomo):

$$H \vee \neg A_1 \vee \cdots \vee \neg A_n \, .$$

In generale, una disgiunzione di letterali viene chiamata *clausola.* Le clausole con al piú un letterale positivo sono dette di *clausole di Horn.* Una disgiunzione di letterali con esattamente un letterale positivo è detta *clausola definita.* Un programma di clausole definite è detto *programma definito.*

In quanto definito sopra abbiamo lasciato implicita la quantificazione delle variabili che occorrono negli atomi. In realtà le clausole/regole sono da intendersi come enunciati universalmente quantificati. In particolare, se $\{X_1, \ldots, X_n\} = \mathit{vars}(H, A_1, \ldots, A_n)$, allora la clausola $H \leftarrow A_1, \ldots, A_n$ è una scrittura abbreviata dell'enunciato

$$\forall X_1 \cdots \forall X_n (H \leftarrow A_1, \ldots, A_n).$$

Supponiamo che $vars(A_1, \ldots, A_n) \setminus vars(H) = \{V_1, \ldots, V_m\}$ e $vars(H) = \{W_1, \ldots, W_p\}$, allora la formula:

$$\forall V_1 \cdots \forall V_m \forall W_1 \cdots \forall W_p (H \leftarrow A_1, \ldots, A_n)$$

equivale a

$$\forall W_1 \cdots \forall W_p (H \leftarrow \exists V_1 \cdots \exists V_m (A_1, \ldots, A_n))$$

Si osservi nel seguente esempio un risvolto pratico della quantificazione indicata:

ESEMPIO 3.1. Si consideri la clausola definita:

$$\texttt{nonno}(\texttt{X}, \texttt{Y}) \leftarrow \texttt{padre}(\texttt{X}, \texttt{Z}), \texttt{padre}(\texttt{Z}, \texttt{Y})$$

In base a quanto detto, possiamo interpretarla in due modi equivalenti:

(1) Per ogni valore di `X`, `Y`, e `Z`, se `X` e `Z` soddisfano il predicato `padre` e `Z` ed `Y` a loro volta soddisfano il predicato `padre`, allora `X` e `Y` devono soddisfare il predicato `nonno`, ovvero:

$$\forall \texttt{X} \forall \texttt{Y} \forall \texttt{Z} (\texttt{nonno}(\texttt{X}, \texttt{Y}) \leftarrow \texttt{padre}(\texttt{X}, \texttt{Z}) \wedge \texttt{padre}(\texttt{Z}, \texttt{Y}))$$

Si tenga presente che in questa clausola (così come nelle successive) il programmatore ha semplicemente introdotto due simboli di predicato, `padre` e `nonno`, e dichiarato che essi soddisfano una determinata proprietà. La lettura intuitiva di questa clausola, connessa alla relazione tra i gradi di parentela, ovvero *"Per ogni valore di* `X`, `Y`*, e* `Z`*, se* `X` *è il padre di* `Z` *e* `Z` *è il padre di* `Y` *allora* `X` *è il nonno paterno di* `Y`*"*, va oltre ciò che viene asserito con la clausola stessa. Tuttavia, per semplicità di esposizione, in quanto segue faremo uso di questa lettura "intesa" delle clausole; sempre tenendo presente che così facendo commettiamo una sorta di abuso di notazione.

(2) Per ogni valore di `X`, `Y`, se esiste uno `Z` tale che se `X` è padre di `Z` e `Z` è padre di `Y` allora `X` è nonno di `Y`, ovvero:

$$\forall \texttt{X} \forall \texttt{Y} (\texttt{nonno}(\texttt{X}, \texttt{Y}) \leftarrow \exists \texttt{Z} (\texttt{padre}(\texttt{X}, \texttt{Z}) \wedge \texttt{padre}(\texttt{Z}, \texttt{Y}))).$$

Cercheremo di classificare i programmi definiti in alcune famiglie che permettono di codificare diverse classi di problemi.

1. Programmi con dominio vuoto—proposizionali

Si consideri il programma P costituito dalle seguenti clausole:

$$\begin{array}{rcl} estate & \leftarrow & \\ caldo & \leftarrow & estate \\ caldo & \leftarrow & sole \\ sudato & \leftarrow & estate, caldo \end{array}$$

La prima clausola è un fatto mentre le altre tre sono regole. In questo esempio tutti i simboli di predicato hanno arità zero, dunque: $\mathcal{F}_P = \emptyset$ e $\Pi_P = \{estate, caldo, sole, sudato\}$.

Nella sintassi Prolog queste clausole vengono scritte così:

```
estate.
caldo :- estate.
caldo :- sole.
sudato :- estate, caldo.
```

Una volta scritto il programma in un file di testo, si può farlo leggere all'interprete Prolog (tramite il comando `consult`) e chiedere all'interprete stesso se un goal è vero o falso.[1] Ad esempio:

```
?- estate.
yes
?- inverno.
no
?- caldo.
yes
?- sudato.
yes
```

Nello standard Prolog, i simboli di predicato, di funzione e di costante devono iniziare con una lettera minuscola, mentre le variabili con una lettera maiuscola (oppure con il trattino di sottolineatura _).

2. Programmi con dominio finito - database programming

In questa sezione illustriamo programmi in cui l'insieme $\mathcal{F}_P$ è costituito da soli simboli di costante.

Per rappresentare in Prolog un albero genealogico come quello di Figura 3.1, è naturale usare il seguente insieme di fatti:

```
padre(antonio,bruno).
padre(antonio,carlo).
padre(bruno,davide).
padre(bruno,ettore).
```

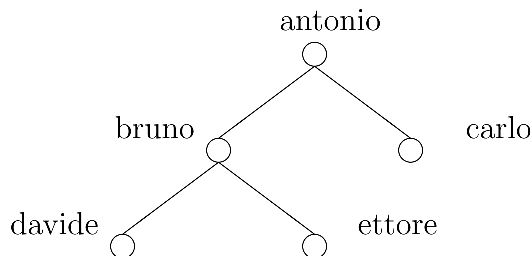


FIGURA 3.1. Albero genealogico

Come prima, si fa consultare il programma all'interprete Prolog e successivamente si può chiedere se un goal è vero o falso:

[1]La istruzione o direttiva `:-consult(NomeFile).` provoca il caricamento di tutte le clausole presenti nel file identificato dall'atomo `NomeFile`. Si noti che molte implementazioni di Prolog offrono differenti istruzioni per caricare il contenuto di un file. Una altra istruzione frequentemente disponibile è `:-compile(NomeFile)`. Spesso accade che `:-consult/1` non sia altro che un sinonimo per `:-compile/1`. Si veda il manuale degli specifici Prolog per individuare l'esistenza di altre istruzioni analoghe a `:-consult(NomeFile)` e le differenze di funzionamento che queste prevedono.

```
?- padre(antonio,bruno).
yes
?- padre(antonio,ettore).
no
?- padre(antonio,Y).
yes Y=bruno ?
```

Quest'ultimo goal contiene la variabile `Y`. Sottoponendo tale goal non si chiede se questo sia vero o falso, ma più precisamente si chiede se esista un modo di istanziare la variabile `Y` in modo che l'istanza del goal così ottenuta sia vera. In questo particolare caso ciò significa chiedere all'interprete Prolog se esista qualche figlio il cui padre è `antonio`. Il primo figlio trovato dall'interprete è `bruno`. Poi un punto di domanda chiede se si vogliono altre risposte, per far sì che l'interprete le cerchi si deve digitare il carattere ";".

Consideriamo un altro esempio:

```
?- padre(X,carlo).
yes X=antonio
```

Questo goal chiede chi sia il padre di `carlo`. La risposta dell'interprete Prolog è, come ci si aspettava, `antonio`.
Per far sì che l'interprete Prolog fornisca in risposta (una ad una) tutte le coppie padri-figli si può sottomettere il goal:

```
?- padre(X,Y).
yes X=antonio,Y=bruno ?  ;
yes X=antonio,Y=carlo ?
⋮
```

Negli esempi visti fino a questo punto abbiamo solo utilizzato clausole unitarie, ovvero fatti. La parte di un programma Prolog costituita dai fatti è solitamente denominata parte *estensionale* del programma. La parte *intensionale* invece è costituita dalle clausole con corpo non vuoto. Queste clausole caratterizzano degli oggetti del dominio in modo implicito, specificando cioé delle proprietà che li relazionano agli oggetti descritti nella parte estensionale del programma. Inseriamo quindi qualche regola nel programma sopra riportato, definendo un nuovo predicato in maniera intensionale (ovvero utilizzando le variabili e non specificando direttamente delle istanze ground):

```
figlio(X,Y) :- padre(Y,X).
```

Il significato inteso di questa clausola è chiaro. Sottomettendo il goal

```
?- figlio(Y,bruno).
```

otteniamo come risposte:

```
yes Y = davide ;
Y = ettore
```

Consideriamo ora la regola

```
nonno(X,Y) :- padre(X,Z),padre(Z,Y).
```

Essa si caratterizza per il fatto che il corpo contiene una variabile che non occorre nella testa. (Si veda l'Esempio 3.1 per un chiarimento sull'interpretazione da dare alle variabili.) Sottoponendo alcuni goal si ottiene:

```
?- nonno(bruno,davide).
no
?- nonno(antonio,ettore).
yes
```

Cerchiamo ora i nipoti di `antonio` sottoponendo il goal:

```
?- nonno(antonio,Y).
yes Y = davide ?  ;
yes Y = ettore ?  ;
no
```

Si noti che la risposta all'ultimo ";" è `no`, ciò significa che non vi sono ulteriori soluzioni.

Nei semplici programmi visti finora non si è sfruttata in nessun modo la ricorsione. Essa tuttavia costituisce una tecnica fondamentale della programmazione con clausole. Vediamo un esempio di come possa essere utilizzata nel definire predicati intensionali:

```
antenato(X,Y) :- padre(X,Y).
antenato(X,Y) :- antenato(X,Z), padre(Z,Y).
```

Anche in questo caso il significato inteso è chiaro. Avremo quindi le seguenti risposte al goal:

```
?-antenato(antonio,Y)
yes Y=bruno ?  ;
yes Y=carlo ?  ;
yes Y=davide ?  ;
yes Y=ettore ?  ;
no
```

3. Programmi con dominio infinito

Consideriamo ora programmi P in cui $\mathcal{F}_P$, ovvero l'insieme dei simboli di funzione usati in P, contiene simboli di funzione con $ar \geq 1$.

In questo modo possiamo, ad esempio, definire il concetto di numerale descrivendo implicitamente un insieme infinito di termini (e quindi denotando un insieme infinito di oggetti del dominio):

```
num(0).
num(s(X)) :- num(X).
```

Assumendo che il significato inteso del predicato `num` sia "essere un numero naturale" abbiamo che il fatto definisce che il simbolo `0` del linguaggio denota un numero. La regola

invece asserisce che dato un numero denotato da `X`, ne esiste un altro denotato da `s(X)`. Risulta intuitivo interpretare `s` come la funzione successore. Con questa lettura, la regola asserisce che il successore di qualcosa è un numero se quel qualcosa è un numero.

Si ricordi che la interpretazione che il lettore/programmatore intuitivamente conferisce al predicato `num`, ovvero di caratterizzare i numeri naturali, non è in alcun modo comunicata/nota all'interprete Prolog. Come vedremo quando nel Capitolo 6 studieremo la semantica dei programmi definiti, l'interprete Prolog agisce in un modo coerente con qualsiasi possibile interpretazione dei simboli del linguaggio.

Chiediamo all'interprete Prolog se `s(s(0))` è un numero:

```
?- num(s(s(0))).
```

La domanda naturalmente deve essere posta dopo aver fatto consultare il programma all'interprete. La risposta sarà ovviamente positiva.
Un goal del tipo:

```
?- num(Z).
```

ha come risposte:

```
yes Z=0 ?  ;
yes Z=s(0) ?  ;
yes Z=s(s(0)) ?  ;
⋮
```

Nota 3.1. La scrittura dei numerali in questo modo può risultare piuttosto prolissa. Un modo per semplificare la notazione è quello di *dichiarare* che `s` è un operatore infisso. In tal modo, anzichè scrivere `s(s(s(s(0))))` sarà sufficiente scrivere `s s s s 0` (con gli spazi!!!). Per far ciò bisogna inserire nel file contenente il programma la direttiva:

```
:- op( 100, fy, s).
```

Per maggiori ragguagli sull'uso di direttive e dichiarazioni di operatori prefissi, infissi e postfissi si rimanda ad un manuale di Prolog (per esempio, il manuale in linea del SICStus Prolog).

Definiamo ora alcuni predicati sui numeri naturali:

```
leq(0, Y).
leq(s(X), s(Y)) :- leq(X, Y).
lt(0, s(Y)).
lt(s(X), s(Y)) :- lt(X, Y).
```

Osservazione: una definizione più corretta del predicato `leq` dovrebbe prevedere che zero sia minore od uguale di un numero naturale e non di una qualunque costante. Dovrebbe dunque essere:

```
leq(0, Y) :- num(Y).
```

4. Definizione di funzioni

Ricordiamo che una funzione $f : A_1 \times \cdots \times A_n \longrightarrow B$ non è altro che un insieme di ennuple $\langle x_1, \ldots, x_n, y\rangle \in A_1 \times \cdots \times A_n \times B$. Poiché siamo in un contesto predicativo, nel descrivere una funzione tramite dei predicati, procederemo caratterizzando l'insieme di ennuple che definiscono la funzione. In altri termini, per definire una funzione n-aria, definiremo un predicato $(n+1)$-ario.

Facendo riferimento alla descrizione precedente dei numeri naturali, descriviamo alcune funzioni sui naturali definendo opportuni predicati. Per comodità indichiamo con $\texttt{s}^i\texttt{(0)}$ il termine $\underbrace{\texttt{s(}\cdots\texttt{(s(}}_{i}\texttt{0)}\cdots\texttt{))}$.

- Successore
  `succ(X, s(X)).`
- Somma
  ```
  plus(X, 0, X).
  plus(X, s(Y), s(Z)) :- plus(X, Y, Z).
  ```
  Queste due clausole asseriscono quali siano le proprietà che la tripla `X,Y,Z` deve soddisfare affinché `Z` denoti la somma dei numeri denotati da `X` e `Y`. Si osservi che, attraverso un uso opportuno delle variabili nei goal con predicato `plus` si ottiene l'operazione inversa della somma. Ciò proprio perché descrivendo dichiarativamente le proprietà della somma, implicitamente descriviamo anche quelle della differenza. Ad esempio il goal
  $$\texttt{:- plus(s(0), X, s}^3\texttt{(0))}$$
  calcola $X = 3 - 1$.
- Prodotto
  ```
  times(X, 0, 0).
  times(X, s(Y), Z) :- times(X, Y, V), plus(V, X, Z).
  ```
- Esponenziale
  ```
  exp(X, 0, s(0)).
  exp(X, s(Y), Z) :- exp(X, Y, V), times(V, X, Z).
  ```
  Si noti che, similmente a quanto accadeva per la somma, caratterizzando l'esponenziale abbiamo implicitamente descritto anche altre funzioni sui numeri interi. Ad esempio, attraverso le interrogazioni:
  (1) :-$\texttt{exp(X, s}^3\texttt{(0), s}^8\texttt{(0))}$ si ottiene la radice cubica di 8,
  (2) :-$\texttt{exp(s}^2\texttt{(0), Y, s}^8\texttt{(0))}$ si ottiene il logaritmo in base 2 di 8.
- Fattoriale
  ```
  fatt(0, s(0)).
  fatt(s(X), Y) :- fatt(X, V), times(s(X), V, Y).
  ```

5. Turing completezza

Dimostreremo in questa sezione che mediante programmi di clausole definite (e disponendo di un interprete per le stesse—lo descriveremo nei prossimi capitoli), si dispone di un

formalismo equivalente a quello della Macchina di Turing. Per far ciò, iniziamo dimostrando che nel formalismo si possono costruire funzioni ricorsive primitive e le funzioni parziali ricorsive.

TEOREMA 3.1. *Se una funzione $f : \mathbb{N}^n \longrightarrow \mathbb{N}$ è ricorsiva (parziale), allora f è definibile da un programma definito.*

DIM. Dobbiamo saper definire le funzioni di base:

- Funzione costante zero:

```
zero(X1, ...,Xn,0).
```

- Funzione successore:

```
succ(X,s(X)).
```

- Funzione di proiezione. Per ogni n e $i \leq n$ possiamo definire:

```
pi_n_i(X1, ...,Xn,Xi).
```

  Per questo punto parrebbe ci fosse il bisogno di un numero infinito di fatti. Tuttavia, dato un certo algoritmo/programma, avremo bisogno solo di un numero finito e predeterminato di funzioni di proiezione.
  Passiamo ora al trattamento delle operazioni di composizione, ricorsione primitiva e minimalizzazione.
- Composizione. Supponiamo di aver definito le funzioni $g_1, \ldots, g_n$ mediante i predicati:

```
p_g1(X1,...,Xm,Y1),
...,
p_gn(X1,...,Xm,Ym)
```

  e la funzione f mediante il predicato:

```
p_f(Y1,...,Yn,Y)
```

  Definiamo la funzione h che le compone:

```
p_h(X1, ...,Xm,Y)  :-  p_g1(X1,...,Xm,Y1),
                       ⋮
                       p_gn(X1,...,Xm,Yn),
                       p_f(Y1,...,Yn,Y).
```

- Ricorsione primitiva. Supponiamo di aver definito le funzioni g e f mediante i predicati:

```
p_g(X1,...,Xn,Y)
p_f(X1,...,Xn,Y1,Y)
```

  Definiamo, per ricorsione primitiva, la funzione h:

```
   p_h(X1,...,Xn,0,Y)  :-  p_g(X1,...,Xn,Y).
p_h(X1,...,Xn,s(X),Y)  :-  p_h(X1,...,Xn, X, Y1),
                           p_f(X1,...,Xn, Y1,Y).
```

- μ-operatore di minimalizzazione. Supponiamo di aver definito la funzione f mediante il predicato

```
p_f(X1,...,Xn,Y,Z)
```

Dobbiamo definire la funzione h tale che $h(X_1, \ldots, X_n, Y) = \mu Y\, (f(X_1, \ldots, X_n, Y) = 0)$, ovvero valga $\mu Y\, (\texttt{p_f}(X_1, \ldots, X_n, Y, 0))$.

```
          p_h(X1,...,Xn,0)  :-  p_f(X1,...,Xn,0,0).
       p_h(X1,...,Xn,s(Y))  :-  p_f(X1,...,Xn,s(Y),0),
                                mai_prima(X1,...,Xn,Y).
     mai_prima(X1,...,Xn,0)  :-  p_f(X1,...,Xn,0,s(_))
  mai_prima(X1,...,Xn,s(Y))  :-  p_f(X1,...,Xn,s(Y),s(_)),
                                mai_prima(X1,...,Xn,Y).
```

□

Dunque alla luce del precedente teorema, possiamo asserire che i programmi definiti sono almeno tanto espressivi quanto lo è la Macchina di Turing.

6. Esercizi

Esercizio 3.1. Si fornisca una dimostrazione alternativa del Teorema 3.1 sfruttando il fatto che una Macchina di Turing si può descrivere fornendo un insieme di fatti che esprimano la sua funzione di transizione δ, ad esempio:

```
delta(q0,0,q1,1,L).
delta(q0,1,q2,1,R).
delta(q0,$,q0,$,L).
...
```

Il nastro della Macchina di Turing, nella usuale notazione:

$$\cdots \$\$\$\ell_n \ell_{n-1} \cdots \ell_2 \ell_1\, c\, r_1 r_2 r_3 \cdots r_{m-1} r_m \$\$\$ \cdots$$

può essere rappresentato da due liste (che sono finite, in ogni istante della computazione) che ne caratterizzano la parte significativa a sinistra $[\ell_1, \ell_2, \ldots, \ell_n]$ e a destra $[c, r_1, r_2, \ldots, r_m]$ della testina (includendo in questa anche il simbolo corrente c).
Per il trattamento delle liste in Prolog si suggerisce di ricorrere al materiale presentato nel Capitolo 7. Tuttavia per il momento si osservi che una lista può essere descritta utilizzando un simbolo di funzione binario `f` e un simbolo di costante di fine lista, diciamo `nil`. Le liste

$$[\,] \qquad [a, b] \qquad [1, 2, 3, 4]$$

si possono quindi rappresentare rispettivamente come:

```
nil    f(a,f(b,nil))    f(1,f(2,f(3,f(4,nil)))).
```

Esercizio 3.2. Si scriva un interprete di Macchine di Turing usando un programma definito.

CAPITOLO 4

Unificazione

In questo capitolo saranno presentati gli aspetti logici ed algoritmici del problema dell'unificazione e le sue soluzioni.

1. Preliminari

Nella prossima definizione introduciamo la nozione di unificatore e, ricorrendo alla relazione di pre-ordine $\leq$ introdotta dalla Definizione 2.24, stabiliamo un criterio di confronto tra diversi unificatori.

DEFINIZIONE 4.1. Dati due termini s e t ed una sostituzione θ diremo che:

- θ è un *unificatore* di s e t se $s\theta \equiv t\theta$ (se esiste un unificatore di s e t, sono detti *unificabili*).
- θ è un *m.g.u.* (*most general unifier*) di s e t se θ è un unificatore e per ogni unificatore σ di s e t vale che $\theta \leq \sigma$.

Si noti che la condizione per la quale nella precedente definizione due termini si considerano unificabili è prettamente sintattica. Per questo motivo la nozione di unificazione qui introdotta viene sovente denominata *unificazione sintattica*, in contrapposizione a nozioni più sofisticate di unificazione in cui, in virtù di opportune assunzioni, si "dichiara" che due termini unificano anche se non sono sintatticamente identici (vedremo un esempio di ciò nella Sezione 5).

ESEMPIO 4.1.

(1) Consideriamo i termini $f(g(X,a),Z)$ e $f(Y,b)$. La sostituzione $[Y/g(c,a), X/c, Z/b]$ è un loro unificatore. Un altro unificatore è $[Y/g(h(W),a), X/h(W), Z/b]$. Questi due unificatori sono tra loro non confrontabili rispetto a $\leq$. L'm.g.u. (in questo caso unico) è invece $[Y/g(X,a), Z/b]$.
(2) I due termini $f(a,b)$ e $f(X,b)$ hanno $[X/a]$ come unificatore.
(3) Per i termini $f(a,b)$ e $f(X,a)$ invece non esiste alcun unificatore. Ciò perché comunque si sostituisca X, si avrà sempre $b \neq a$. Questo esempio si può visualizzare meglio rappresentando i termini con la struttura ad albero di Figura 4.1. L'unificabilità si può infatti ridurre ad un problema di ricerca di isomorfismo di alberi etichettati.
(4) Consideriamo i termini X e $g(X)$, sono unificabili? La sostituzione $[X/g(X)]$ sembra una buona candidata. Ciò che si ottiene con la sua applicazione è però: $g(X)$ e $g(g(X))$. Se la stessa sostituzione venisse applicata infinite volte otterremmo due termini $g(g(\cdots(X)\cdots))$ di lunghezza infinita che differiscono per un $g(X)$ (e

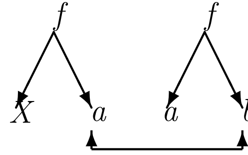


FIGURA 4.1.

possono quindi, in un certo senso, essere considerati uguali). La soluzione è comunque inaccettabile perché nei linguaggi del prim'ordine che ci interessa trattare, consideriamo solo termini di lunghezza finita.

NOTA 4.1. Si osservi, anche alla luce dell'ultimo esempio, che se s è sottotermine proprio di t allora s e t non saranno unificabili. In particolare, il problema di determinare se dati due termini X (variabile) e t, il primo sia sottotermine (proprio) del secondo ha particolare rilevanza in ogni algoritmo di unificazione. Tale test è detto *occur-check*.

Introduciamo ora il concetto di sistema di equazioni (si noti che un sistema di equazioni può essere visto indifferentemente come insieme di equazioni o come congiunzione di equazioni; in quanto segue adotteremo questa seconda convenzione).

DEFINIZIONE 4.2. $C \equiv (s_1 = t_1 \wedge \cdots \wedge s_n = t_n)$, dove ogni s_i e t_i è un termine, si dice *sistema di equazioni*. Se θ è una sostituzione, allora

- θ è unificatore di C se per ogni $i \in \{1, \ldots, n\}$ si ha che $s_i\theta \equiv t_i\theta$.
- θ è *m.g.u.* di C se θ è unificatore di C e per ogni unificatore σ di C si ha che $\theta \leq \sigma$.

Una ulteriore definizione a cui ricorreremo nel capitolo successivo:

DEFINIZIONE 4.3. Date due formule atomiche A_1 e A_2 ed una sostituzione θ, diremo che θ è unificatore di A_1 e A_2 se

- A_1 e A_2 hanno stesso simbolo predicativo e stessa arità, ovvero $A_1 \equiv p(s_1, \ldots, s_n)$ e $A_2 \equiv p(t_1, \ldots, t_n)$ ove ogni s_i e t_i è un termine;
- la sostituzione θ è unificatore del sistema di equazioni $s_1 = t_1 \wedge \cdots \wedge s_n = t_n$.

Nel caso in cui θ sia anche m.g.u., scriveremo $\theta = mgu(A_1, A_2)$.

2. Il problema dell'unificazione

Dato un sistema C, si possono porre almeno tre problemi:

(1) Problema di decisione: esiste un unificatore θ di C?
(2) Verificare che non esiste un unificatore, altrimenti fornirne uno.
(3) Verificare che non esiste un unificatore, altrimenti fornire l'm.g.u..

Si noti che saper risolvere il problema (2) implica poter risolvere anche il problema (1). Inoltre, saper risolvere (3) implica saper risolvere (2). L'algoritmo di unificazione risolve direttamente il punto (3).

L'algoritmo che vedremo deriva da quello presente nella tesi di laurea di Jacques Herbrand [**Her30**].[1]

[1]Il primo algoritmo di unificazione che fu inserito in una procedura risolutiva è quello di Robinson [**Rob65, Rob68**]. Martelli e Montanari in [**MM82**] svilupparono un'ottimizzazione dell'algoritmo

DEFINIZIONE 4.4. Un sistema di equazioni $C \equiv s_1 = t_1 \wedge \cdots \wedge s_n = t_n$ è in *forma risolta* se e solo se $s_1, \ldots, s_n$ sono variabili tutte diverse tra loro e tali da non occorrere nei termini $t_1, \ldots, t_n$. In altre parole, C ha la forma $X_1 = t_1 \wedge \cdots \wedge X_n = t_n$ con $\forall i \neq j$, $X_i \not\equiv X_j$ e $\forall i, j$, $X_i \notin vars(t_j)$.

Dato un sistema in forma risolta $C \equiv X_1 = t_1 \wedge \cdots \wedge X_n = t_n$, ad esso viene associata naturalmente una sostituzione $\theta = [X_1/t_1, \ldots, X_n/t_n]$. Questa è una sostituzione particolare per cui si ha che $dom(\theta) \cap ran(\theta) = \emptyset$. Tale condizione assicura l'idempotenza, ovvero la proprietà che $t\theta = t\theta\theta$ per ogni termine t. Vale inoltre il seguente lemma.

LEMMA 4.1. *Se $C \equiv X_1 = t_1 \wedge \cdots \wedge X_n = t_n$ è in forma risolta allora la sostituzione $\theta = [X_1/t_1, \ldots, X_n/t_n]$ è m.g.u. di C.*

DIM. Dobbiamo innanzitutto dimostrare che θ è un unificatore. Per fare ciò dobbiamo dimostrare che per ogni $i = 1, \ldots, n$ vale $X_i\theta = t_i\theta$. Ma questo segue dal fatto che $X_i\theta = t_i$, per definizione di θ, e $t_i\theta = t_i$ poiché per ipotesi (forma risolta) per ogni j, $X_j \notin vars(t_i)$.

Ci resta da dimostrare che θ è m.g.u., ovvero che se σ è un unificatore di C allora esiste η t.c. $\sigma = \theta\eta$. Mostriamo una proprietà più forte, cioè che se σ è un unificatore allora $\sigma = \theta\sigma$. Dobbiamo procedere distinguendo due casi, le variabili $X_1, \ldots, X_n$ e tutte le altre.

- Per $i = 1, \ldots, n$ si ha $X_i\sigma = t_i\sigma$ (poiché σ è unificatore), ma $t_i = X_i\theta$ quindi $X_i\sigma = (X_i\theta)\sigma = X_i(\theta\sigma)$.
- Per $V \neq X_1 \wedge \cdots \wedge V \neq X_n$ risulta immediatamente che $V_i\sigma = V_i\theta\sigma = V_i\sigma$ in quanto $V_i\theta = V_i$.

□

Un risultato che ci verrà utile in seguito:

LEMMA 4.2 (Binding). *Sia θ una sostituzione. Se $X\theta = t\theta$ allora $\theta = [X/t]\theta$.*

DIM. Dimostriamo la tesi trattando i seguenti casi:

- Per la variabile X: $X\theta = t\theta$ (per l'enunciato del lemma) $X[X/t]\theta = t\theta$
- Per variabili $Y \in dom(\theta), Y \neq X$: $Y[X/t]\theta = Y\theta$
- Per $Y \notin dom(\theta), Y \neq X$: $Y[X/t]\theta = Y\theta = Y$.

□

3. Algoritmo di unificazione

In questa sezione affronteremo il problema (3) enunciato nella sezione precedente. Ovvero: dato un sistema di equazioni determinare se esso ammetta o meno un m.g.u.. Preannunciamo che, per quanto concerne l'unificazione sintattica, l'm.g.u. è unico a meno di varianti (dimostreremo questo fatto nel Teorema 4.2). Vedremo in seguito (Sezione 5) che esistono nozioni più "raffinate" di unificazione per le quali questa proprietà non è garantita.

Questo algoritmo, dovuto ad Herbrand, è composto dalle sei regole di riscrittura sotto riportate. L'algoritmo opera applicando ripetutamente queste regole in modo non deterministico finché è possibile. Ogni applicazione di una regola di riscrittura trasforma quindi il sistema di equazioni. Questo processo termina producendo un sistema in forma risolta se e

di unificazione di Herbrand in grado di eseguire l'unificazione in tempo quasi lineare. Paterson e Wegman in [**PW78**] presentarono un nuovo algoritmo che permette di effettuare l'unificazione in tempo lineare.

solo se il sistema iniziale ammette m.g.u.. Le regole dell'algoritmo $\mathsf{Unify}(\cdot)$ sono riportate in Figura 4.2.

(1)	$f(s_1,\ldots,s_n) = f(t_1,\ldots,t_n) \wedge C$	$\mapsto$	$s_1 = t_1 \wedge \cdots \wedge s_n = t_n \wedge C$
(2)	$f(s_1,\ldots,s_m) = g(t_1,\ldots,t_n) \wedge C$ $f \neq g$ oppure $m \neq n$	$\mapsto$	false
(3)	$X = X \wedge C$	$\mapsto$	C
(4)	$t = X \wedge C$ $t \notin \mathcal{V}$	$\mapsto$	$X = t \wedge C$
(5)	$X = t \wedge C$ $X \notin vars(t)$ e $X \in vars(C)$	$\mapsto$	$C[X/t] \wedge X = t$
(6)	$X = t \wedge C$ $X \in vars(t), X \not\equiv t$	$\mapsto$	false

Figura 4.2. Regole dell'algoritmo di unificazione

La regola (1) include implicitamente anche la trasformazione $a = a \wedge C \mapsto C$ per ogni termine costituito da una qualsiasi costante a. Ciò permette di eliminare dal sistema tutte le equazioni ovvie. Si osservi che la applicazione di qualsiasi regola di questo algoritmo non introduce nuove variabili nel sistema.

Il risultato principale di questa sezione sarà il Teorema 4.1. Esso asserisce le proprietà di terminazione, correttezza e completezza dell'algoritmo $\mathsf{Unify}(\cdot)$. Per poterlo dimostrare peró dobbiamo introdurre alcuni concetti ausiliari. Intuitivamente, l'idea base della dimostrazione consisterà prima nell'introdurre un modo per misurare quanto è complesso un sistema (ovvero quanto è "lontano" da un sistema in forma risolta). Poi la dimostrazione procederà facendo vedere che ogni applicazione di una regola di trasformazione produce un sistema in un certo senso "più semplice" del precedente.

Introduciamo formalmente questa tecnica di prova. La prima nozione che ci serve è:

Definizione 4.5. Una relazione d'ordine $\prec \subseteq A \times A$ è un *buon ordine* quando non esistono infiniti elementi $a_1, a_2, \cdots \in A$ tali che $\cdots \prec a_4 \prec a_3 \prec a_2 \prec a_1$ (ovvero non esiste alcuna catena discendente infinita).

Esempio 4.2. Se $\prec$ è l'usuale relazione di minore sugli interi, allora $\langle \mathbb{N}, \prec \rangle$ è un buon ordine. Invece $\langle \mathbb{Z}, \prec \rangle$ non è un buon ordine.

La seguente definizione dice come sia possibile combinare due buoni ordini tramite il prodotto cartesiano. Questo ci permetterà di combinare buoni ordini per ottenere buoni ordini su domini più complessi.

Definizione 4.6. Se $\langle A, \prec_A \rangle$ e $\langle B, \prec_B \rangle$ sono ordini, allora l'ordine lessicografico ottenuto dai due è l'ordine $\langle A \times B, \prec \rangle$ definito come:

$$\langle X, Y \rangle \prec \langle X', Y' \rangle \quad \text{se e solo se} \quad \begin{array}{l} X \prec_A X' \vee \\ X =_A X' \wedge Y \prec_B Y' \end{array}$$

L'idea può facilmente estendere a terne, quaterne e cosí via. Intuitivamente, l'ordine lessicografico altro non è che l'ordinamento del vocabolario: si confronta prima il primo carattere, poi il secondo, il terzo, e cosí via.

Sussiste la seguente proprietà:

PROPOSIZIONE 4.1. *Se* $\langle A, \prec_A \rangle$ *e* $\langle B, \prec_B \rangle$ *sono buoni ordini, allora l'ordine lessicografico ottenuto dai due è un buon ordine.*

DIM. Esercizio. □

Si può intuitivamente dimostrare che l'ordine lessicografico è un buon ordine immaginando lo spazio generato dagli insiemi A e B, rappresentati su assi cartesiani. Preso un punto a caso nello spazio di coordinate $\langle X, Y \rangle$ con $X \in A$ e $Y \in B$ si nota che tutti i punti minori di questo sono tutti quelli con ascissa minore di X (che sono infiniti) e quelli con ordinata minore di Y (che invece sono finiti). Si potrebbe pensare che il fatto che esistano infiniti punti minori di $\langle X, Y \rangle$ sia in contraddizione con la definizione di buon ordine. In realtà si vede facilmente che, se $\langle A, \prec_A \rangle$ e $\langle B, \prec_B \rangle$ sono buoni ordini, allora non si può costruire una catena discendente infinita per $A \times B$. Questa infatti indurrebbe una catena discendente infinita per uno tra A o B, contraddicendo l'ipotesi.

Una altra nozione che ci sarà utile è quella di *multiinsieme*. Possiamo definire un multiinsieme come un insieme in cui le ripetizioni "contano", ovvero un multiinsieme può avere due (o più) elementi uguali. Ad esempio, $\{\!\![\, 5, 4, 5, 5, 3, 2 \,]\!\!\}$ e $\{\!\![\, 5, 4, 3, 2 \,]\!\!\}$ sono due multiinsiemi diversi. Si noti che un insieme è un multiinsieme particolare in cui ogni elemento compare una sola volta. Questo tipo di oggetti sono utili per modellare contenitori di entità che "si consumano".

I multiinsiemi a cui siamo interessati in questo capitolo sono *finiti* e *piatti*, ovvero contengono solo un numero finito (e un numero finito di volte) di elementi presi da un insieme A (non contengono quindi altri multiinsiemi come elementi).

Estendiamo la nozione di buon ordine ai multiinsiemi. Se $\langle A, \prec_A \rangle$ è un buon ordine, allora un buon ordine per i multiinsiemi, con elementi presi da A, è dato dalla chiusura transitiva della regola:

$$\{\!\![\, s_1, \ldots, s_{i-1}, t_1, \ldots, t_n, s_{i+1}, \ldots, s_m \,]\!\!\} \prec \{\!\![\, s_1, \ldots, s_i, \ldots, s_m \,]\!\!\}$$

se $t_1 \prec_A s_i \wedge \cdots \wedge t_n \prec_A s_i$ e $n \geq 0$ (il caso $n = 0$ significa che il multiinsieme di sinistra si ottiene da quello di destra togliendo s_i ma non aggiungendo alcun t_i).

In altre parole, un multiinsieme X è più piccolo di un'altro multiinsieme Y se X è ottenibile da Y eliminando un elemento o sostituendolo con uno o più elementi più piccoli (secondo l'ordine su A).

ESEMPIO 4.3. Se $\langle A, \prec \rangle$ è l'insiene dei numeri naturali con l'ordinamento usuale, allora $\{\!\![\, 5, 1, 1, 1, 2, 2, 2, 0, 0, 0, 2, 1, 2 \,]\!\!\} \prec \{\!\![\, 5, 3, 2, 1, 2 \,]\!\!\}$: in questo caso il primo è stato ottenuto dal secondo sostituendo a 3 gli elementi 1,1,1,2,2,2,0,0,0. Vale anche $\{\!\![\, 5, 3, 1, 2 \,]\!\!\} \prec \{\!\![\, 5, 3, 2, 1, 2 \,]\!\!\}$ perchè in questo caso il primo è ottenibile dal secondo eliminando il suo terzo elemento.

Introduciamo ora una misura per valutare la complessità sintattica dei termini:

DEFINIZIONE 4.7. Sia t un termine, allora

$$size(t) = \begin{cases} 0 & \text{Se } t \text{ è una variabile} \\ 1 + \sum_{i=1}^{n} size(s_i) & \text{Se } t \text{ è della forma } f(s_1, \ldots, s_n) \end{cases}$$

Per ogni termine t la funzione *size* conta il numero di occorrenze di simboli di $\mathcal{F}$ in t. Infatti, $size(X) = 0$ se X è una variabile, mentre ad esempio $size(f(a, b, f(X, c, Y))) = 5$ in quanto vi occorrono 5 simboli di funzione (o costante).

Abbiamo ora quasi tutti gli ingredienti per definire una misura di complessità per i sistemi di equazioni tale da essere ridotta ad ogni passo di $\mathsf{Unify}(\cdot)$. Considerando infatti la somma delle *size* di tutti i termini presenti in un sistema, si osserva che alcune azioni dell'algoritmo fanno diminuire tale valore. Ciò peró non basta perchè questo non accade tutte le regole di trasformazione. Infatti, se ad esempio tramite la regola (5) sostituisco ad X (la cui *size* vale 0) il termine $f(f(f(a)))$ (la cui *size* vale 4), allora la *size* totale del sistema aumenta (in particolare aumenta di una quantità proporzionale al numero di occorrenze di X nel sistema).

Dobbiamo quindi raffinare la misura basata su *size* in modo da definirne una che funzioni per tutte le regole di trasformazione. Un altro ingrediente utile:

DEFINIZIONE 4.8. Sia C un sistema di equazioni. Una variabile $X \in vars(C)$ è detta *risolta* se X occorre in C solo una volta e nella forma $X = t \wedge C'$ con $X \notin vars(t)$ e $X \notin vars(C')$. Dato un sistema C, definiamo l'insieme $Uns(C)$ nel modo seguente:

$$Uns(C) = \{X \in vars(C) \ : \ X \text{ non è risolta}\}$$

Notiamo che la applicazione della regola (5) ha come effetto di diminuire il numero di variabili che non sono risolte.

Dato che abbiamo introdotto un criterio per dire se un multiinsieme è o meno più piccolo di un altro. Trasportiamo questo criterio all'insieme delle equazioni di un sistema. Ciò, unitamente alla funzione *size*, ci fornirà la misura cercata:

DEFINIZIONE 4.9. Sia C un sistema di equazioni. Definiamo la sua misura di complessità come:

$$Compl(C) = \langle |Uns(C)|, \{\!\!\{\, size(\ell) : \ell = r \text{ in } C \,\}\!\!\} \rangle$$

La definizione precedente assegna come misura di complessità di un sistema un oggetto dell'insieme $\mathbb{N} \times \mathbb{M}$, dove $\mathbb{M}$ è l'insieme dei miltiinsiemi (finiti) di naturali. Ogni elemento di $\mathbb{N} \times \mathbb{M}$ è quindi una coppia con prima componente un naturale e seconda componente un multiinsieme finito di naturali. Possiamo ora introdurre un buon ordine su $\mathbb{N} \times \mathbb{M}$ come l'ordine lessicografico ottenuto dai due buoni ordini che abbiamo a disposizione su $\mathbb{N}$ (l'usuale $<$) e su $\mathbb{M}$ (vedi pag. 43). Poiché questo è un buon ordine, non esistono catene discendenti infinite, quindi se dimostriamo che ogni passo di $\mathsf{Unify}(\cdot)$ fa diminuire la complessità, ne seguirà che l'algoritmo deve terminare (altrimenti avremmo un assurdo perchè infiniti passi dell'algoritmo individuerebbero una catena discendente infinita).

Il teorema successivo garantisce le proprietà di terminazione, correttezza e completezza dell'algoritmo $\mathsf{Unify}(\cdot)$.

TEOREMA 4.1. *Valgono le seguenti proprietà:*

(1) $\mathsf{Unify}(C)$ *termina per qualsiasi sistema C.*
(2) $\mathsf{Unify}(C)$ *restituisce* false *oppure un sistema C' in forma risolta.*
(3) *Una sostituzione θ è unificatore di C se e solo se θ è unificatore di C'.*

DIM.

(1): Dimostriamo che $\mathsf{Unify}(C)$ termina per qualsiasi sistema C facendo vedere che qualunque passo di $\mathsf{Unify}(\cdot)$ fa diminuire la misura complessità $Compl(\cdot)$ del sistema. Per farlo analizziamo il comportamento di tutte le regole di riscrittura:

Regola (1): $f(s_1, \ldots, s_n) = f(t_1, \ldots, t_n) \wedge C \mapsto s_1 = t_1 \wedge \cdots \wedge s_n = t_n \wedge C$.
Una di queste equazioni $s_1 = t_1, \ldots, s_n = t_n$ potrebbe essere del tipo $X = t_i$ con X non occorrente altrove, quindi $|Uns(\cdot)|$ diminuisce o resta costante, ma sicuramente non aumenta. Inoltre la seconda componente della misura (il multiinsieme) diminuisce (secondo l'ordine su $\mathbb{M}$) perchè $size(f(s_1, \ldots, s_n)) = 1 + \sum_{i=1}^{n} size(s_i)$ viene eliminato e sostituito con $size(s_1), \ldots, size(s_n)$. Quindi la misura di complessità del sistema diminuisce.

Regole (2) e (6): Queste regole portano ad immediata terminazione con false.

Regola (3): $X = X \wedge C \mapsto C$.
Questa regola elimina l'equazione $X = X$. Può accadere che $|Uns(\cdot)|$ diminuisca o rimanga invariata, ma anche questo caso diminuisce la seconda componente della misura di complessità (il multiinsieme) perchè la occorrenza di 0 ($size(X = X) = 0$) viene eliminata dal multiinsieme.

Regola (4): $t = X \wedge C \mapsto X = t \wedge C$.
Anche in questo caso $|Uns(\cdot)|$ può diminuire o rimanere costante ma il multiinsieme diminuisce perchè dato che t non è una variabile si ha certamente $size(t) > 0$, quindi nel multiinsieme il numero $size(t)$ verrà rimpiazzato da 0.

Regola (5): $X = t \wedge C \mapsto C[X/t] \wedge X = t$.
La variabile X non occorre in t ma occorre in C. Quindi applicando la regola (5) X diventa risolta. Diminuisce perció il valore di $|Uns(\cdot)|$.

(2): Sappiamo dal punto precedente che $\mathsf{Unify}(\cdot)$ termina sempre. Ciò avviene perche viene prodotto false o perchè non è possibile applicare alcuna regola di trasformazione. Se non è stato prodotto false e il sistema non è ancora in forma risolta, allora si vede osservando le regole di riscrittura, che almeno una di esse risulta ancora applicabile, quindi l'algoritmo avrebbe potuto procedere, ma ciò contraddice l'ipotesi.

(3): Per mostrare che θ è unificatore di C se e solo se lo è di C', mostriamo che tale proprietà si tramanda passo per passo a tutti i sistemi intermedi prodotti dalle applicazioni delle regole dell'algoritmo.

(1) Per definizione di applicazione di sostituzione, $f(s_1, \ldots, s_n)\theta = f(t_1, \ldots, t_n)\theta$ vale se e solo se vale $s_i\theta = t_i\theta$ per ogni $i \in \{1, \ldots, n\}$.
(2) Per definizione di unificatore, non esiste nessun unificatore di $f(\ldots)$ e $g(\ldots)$ con f diverso da g.
(3) Banale perchè ogni sostituzione è unificatore di $X = X$.
(4) Anche in questo caso la proprietà vale banalmente.
(5) Consideriamo i due sistemi $C \wedge X = t$ e $C[X/t] \wedge X = t$.
Sia θ unificatore di $C \wedge X = t$. Per il Lemma 4.2, poiché $X\theta = t\theta$, si ha $\theta = [X/t]\theta$. Pertanto $C[X/t]\theta = C\theta$ e dunque θ è unificatore $C[X/t]$.
Sia ora θ unificatore di $C[X/t] \wedge X = t$. Sempre per il (Lemma 4.2) $\theta = [X/t]\theta$. Pertanto $C\theta = C[X/t]\theta$ che per ipotesi è composto da una congiunzione di identità sintattiche.
(6) X è sottotermine proprio di t. Essi non saranno mai unificabili.

□

Esercizio 4.1. Definire una misura della complessità di un sistema di equazioni che non utilizzi i multiinsiemi. In base a questa nuova nozione di misura sviluppare una dimostrazione alternativa della terminazione dell'algoritmo $\mathsf{Unify}(\cdot)$.

Corollario 4.1. Sia C un sistema di equazioni.

- Se $\mathsf{Unify}(C) = \mathsf{false}$ allora C non ammette unificatori.
- Se $\mathsf{Unify}(C) \equiv X_1 = t_1 \land \cdots \land X_n = t_n$ è in forma risolta, allora $[X_1/t_1, \ldots, X_n/t_n]$ è m.g.u. di C.

Nota 4.2. Per ragioni di efficienza e semplicità implementativa, in SICStus Prolog (ma lo stesso dicasi per la maggior parte delle implementazioni di Prolog esistenti) l'unificazione è implementata senza il controllo "occur-check" (senza cioé la regola (6) dell'algoritmo $\mathsf{Unify}(\cdot)$). Si può tuttavia richiedere esplicitamente la unificabilità con occur-check utilizzando il predicato built-in unify_with_occurs_check fornito nella libreria terms. In dettaglio, va invocato il comando:

```
use_module( library( terms )).
```

(anche come direttiva) e poi, se S e T sono i due termini da unificare, si esegue:

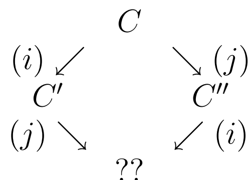
```
unify_with_occurs_check( S , T ).
```

Per applicare tale predicato a sistemi di equazioni sarà sufficiente definire ricorsivamente un predicato che lo richiama.

4. Osservazioni circa l'algoritmo $\mathsf{Unify}(C)$

4.1. Effetti del non determinismo. Nell'esecuzione dell'algoritmo di unificazione, può accadere che a seconda della diversa scelta di regole applicate, si ottengano risultati, sistemi intermedi e finali, diversi. Vediamo ora invece di chiarire questo concetto con degli esempi.

Ammettiamo di avere un sistema C, supponiamo che si possano applicare a C due regole (i) e (j). Supponiamo che applicando a C la prima regola otteniamo un sistema C' mentre applicando la seconda (sempre a C) otteniamo un sistema C''. Supponiamo ora di applicare la regola (j) a C' e la regola (i) a C''. Ci chiediamo: il risultato che si trova, sarà lo stesso? In altri termini, le strade non deterministiche che si aprono possono confluire nello stesso risultato?



In generale, come vedremo, la risposta è negativa.

Esempio 4.4. In questo esempio verrà risolta un'equazione seguendo due strade distinte. Mostreremo cosí come sia possibile giungere a m.g.u. distinti. Consideriamo il seguente sistema C costituito da una sola equazione:

$$C \equiv f(X, Y, Z) = f(Y, Z, X)$$

Applicando la regola (1) otteniamo un sistema di tre equazioni:

$$f(X,Y,Z) = f(Y,Z,X) \overset{(1)}{\mapsto} X = Y \wedge Y = Z \wedge Z = X$$

Scegliamo la prima delle 3 equazioni: $X = Y$. Se si applica la regola (5) dell'algoritmo di unificazione, ovvero si applica la sostituzione $[X/Y]$ si giunge al sistema:

$$X = Y \wedge Y = Z \wedge Z = Y$$

Selezioniamo ora la seconda equazione del sistema ottenuto e applichiamo nuovamente la regola (5). Ciò causa la applicazione della sostituzione $[Y/Z]$, e produce il sistema:

$$X = Z \wedge Y = Z \wedge Z = Z$$

La equazione $Z = Z$ viene ora eliminata applicando la regola (3). Otteniamo così un sistema in forma risolta corrispondente all'unificatore $\theta = [X/Z, Y/Z]$.

Consideriamo ora una diversa scelta della sequenza di regole da applicare al sistema $X = Y \wedge Y = Z \wedge Z = X$. Scegliamo di applicare la regola (5) alla seconda equazione con sostituzione $[Y/Z]$. Otteniamo:

$$X = Z \wedge Y = Z \wedge Z = X$$

Ora selezioniamo la terza equazione e applichiamo nuovamente la regola (5) (sostituzione $[Z/X]$), ottenendo:

$$X = X \wedge Y = X \wedge Z = X.$$

Ora è possibile eliminare l'equazione $X = X$, con la regola (3), ottenendo un sistema in forma risolta corrispondente all'unificatore $\theta' = [Y/X, Z/X]$.

I due unificatori θ e θ' sono diversi. Ciò che possiamo però osservare è che i due m.g.u. calcolati sono varianti, infatti vale:

- $[X/Z, Y/Z][Z/X, X/Z] = [Y/X, Z/X]$ e
- $[Y/X, Z/X][X/Z, Z/X] = [X/Z, Y/Z]$.

Un altro esempio è il seguente: a partire da $f(X,Y) = f(a,Z)$ si possono ottenere:

- $[X/a, Y/Z]$ e
- $[Y/a, Z/Y]$.

I due m.g.u. non sono varianti ma equivalenti a meno di varianti (ovvero si ottengono l'uno con l'altro componendoli a varianti).

L'intuizione fornita dal precedente esempio è in realtà effetto di un più generale risultato:

TEOREMA 4.2. *Se C'_1 e C'_2 sono sistemi in forma risolta che si ottengono non deterministicamente da* Unify(C) *allora essi identificano due sostituzioni θ'_1 e θ'_2 che sono tra loro equivalenti a meno di varianti.*

DIM. Dal Teorema 4.1 discende che sia θ'_1 che θ'_2 sono unificatori di C. Inoltre θ'_1 e θ'_2 sono entrambi m.g.u.. Dunque varrà che: $\theta'_1 \leq \theta'_2$ e $\theta'_2 \leq \theta'_1$. Per il Lemma 2.7 θ'_1 e θ'_2 sono equivalenti a meno di varianti. □

Abbiamo quindi stabilito che tutte le alternative soluzioni che non deterministicamente possono essere ottenute dall'algoritmo Unify(C) sono equivalenti a meno di varianti le une delle altre. Ci si può chiedere però se anche il numero di passi che l'algoritmo compie per

calcolare un unificatore sia invariante rispetto alle scelte non deterministiche. Il seguente esempio fornisce una risposta negativa.

Esempio 4.5. Consideriamo il semplice sistema di due equazioni: $f(X) = f(X) \wedge X = f(a)$. Applicando prima la regola (5) (utilizzando la sostituzione $[X/f(a)]$) e successivamente tre volte la regola (1), otteniamo l'unificatore $[X/f(a)]$ in 4 passi. Tuttavia lo stesso unificatore si sarebbe potuto ottenere in due passi applicando inizialmente la regola (1) alla prima equazione e successivamente la regola (3).

4.2. Aspetti computazionali di $\mathsf{Unify}(C)$. Ci chiediamo ora quali sono le caratteristiche computazionali dell'algoritmo $\mathsf{Unify}(C)$.

Esempio 4.6. Consideriamo il seguente sistema di quattro equazioni:

$$X_1 = f(X_2, X_2) \wedge X_2 = f(X_3, X_3) \wedge X_3 = f(X_4, X_4) \wedge X_4 = f(X_5, X_5)$$

Applicando la sostituzione indotta (regola (5)) alla quarta equazione otteniamo:

$$X_1 = f(X_2, X_2) \wedge X_2 = f(X_3, X_3) \wedge X_3 = f(f(X_5, X_5), f(X_5, X_5)) \wedge X_4 = f(X_5, X_5)$$

Ora possiamo applicare la sostituzione per X_3:

$$\begin{array}{l} X_1 = f(X_2, X_2) \wedge X_2 = f(f(f(X_5, X_5), f(X_5, X_5)), f(f(X_5, X_5), f(X_5, X_5))) \wedge \\ X_3 = f(f(X_5, X_5), f(X_5, X_5)) \wedge X_4 = f(X_5, X_5) \end{array}$$

e così via.

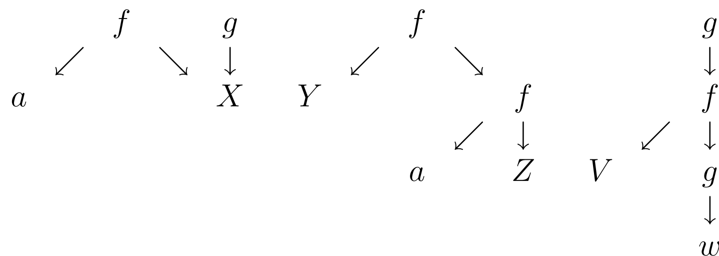
Appare chiaro che durante l'esecuzione dell'algoritmo di unificazione, la dimensione dei termini possa crescere molto velocemente: il termine relativo a X_4 ha *size* 1, quello relativo a X_3 ha *size* 3, quello relativo a X_2 ha *size* 7 e se avessimo proseguito avremmo verificato che il termine relativo a X_1 avrebbe avuto *size* 15.

L'intuizione che ne scaturisce è corretta: si dimostra infatti che l'applicazione di questo algoritmo conduce alla costruzione di termini la cui dimensione cresce in maniera esponenziale rispetto alle dimensioni del sistema iniziale. Tuttavia, la caratteristica del non-determinismo e la applicazione esplicita delle sostituzioni rende questo algoritmo adatto per descrivere semplicemente le dimostrazioni di correttezza, completezza e terminazione.

Vediamo brevemente che l'impiego di opportune strutture dati permette di ovviare al problema della crescita esponenziale della dimensione dei termini. Per semplicità consideriamo un esempio specifico:

$$f(a, X) = f(Y, f(a, Z)) \wedge g(X) = g(f(V, g(w)))$$

Costruiamo un grafo che avrà un nodo per ogni occorrenza di simbolo funzionale e un nodo per ogni variabile (indipendentemente da quante volte tale variabile occorre nel sistema). Gli archi diretti denotano la dipendenza indotta dai termini in gioco tra un'occorrenza di un simbolo funzionale e i sottotermini legati ai suoi argomenti:



L'algoritmo di unificazione può a questo punto essere eseguito utilizzando questa rappresentazione. Selezionata una equazione $\ell = r$ dal sistema si collassano i due nodi radice associati ai termini ℓ e r. Se i due nodi erano etichettati:

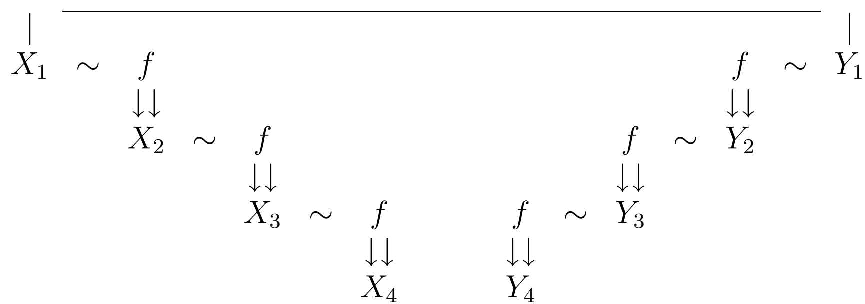
- con simboli funzionali diversi, allora si termina con fallimento;
- con lo stesso simbolo funzionale f, allora si aggiungono le equazioni relative ai figli senza ripetere l'equazione di partenza;
- uno con una variabile e l'altro con un simbolo di funzione, allora si effettua l'*occur-check* (si verifica l'aciclicità del grafo ottenuto).

A questo punto si seleziona una altra equazione e si ripete il procedimento.

Esempio 4.7. Consideriamo il sistema

$$X_1 = f(X_2, X_2) \wedge X_2 = f(X_3, X_3) \wedge X_3 = f(X_4, X_4) \wedge$$
$$Y_1 = f(Y_2, Y_2) \wedge Y_2 = f(Y_3, Y_3) \wedge X_3 = f(Y_4, Y_4) \wedge$$
$$X_1 = Y_1$$

Anche su sistemi di questo tipo l'algoritmo di Herbrand ha un comportamento esponenziale. Costruiamo il grafo associato:



Si dimostra che l'esecuzione dell'algoritmo utilizzando la struttura dati grafo, avviene in tempo polinomiale.

4.3. Significato dell'unificazione. Ci chiediamo ora cosa significhi per due termini "unificare". Supponiamo di scegliere un alfabeto, conseguentemente risulta determinato il relativo Universo di Herbrand $\mathcal{H}$. Consideriamo un generico termine t. Se $X_1, \ldots, X_n$ sono le variabili che vi occorrono allora possiamo idealmente pensare che t "rappresenti" l'insieme di tutti i termini ground che si possono ottenere da t istanziando ogni X_i con un certo elemento di $\mathcal{H}$ (chiaramente se t è ground allora "rappresenterà" solo se stesso).

Se due termini ground s_1 e s_2 sono sintatticamente uguali (ad esempio $s_1 \equiv s_2 \equiv f(1, 2)$), ovviamente unificano; inoltre qualunque sia l'interpretazione che sceglieremo per il sottostante linguaggio, gli oggetti denotati da s_1 e s_2 saranno uguali.

Supponiamo ora che due termini t_1 e t_2 unifichino tramite un m.g.u. θ (cioè $t_1\theta \equiv t_2\theta$). Vedendoli come "rappresentanti" di insiemi di termini ground, il fatto che essi unifichino significa che esiste un sottoinsieme di termini ground che sono rappresentati sia da t_1 che da t_2. Inoltre, esiste un termine che rappresenta proprio questo sottoinsieme: $t_1\theta$.

Possiamo concludere che la unificabilità di due termini certifica che, almeno in parte, essi denoteranno gli stessi oggetti in qualsiasi interpretazione possibile.

Come ulteriore esempio, consideriamo i termini $f(0, X)$ e $f(X, f(0, 0))$. Essi non unificano, tuttavia, se f venisse interpretato come una funzione che gode della proprietà commutativa (ad esempio come l'unione insiemistica ($\cup$) mentre 0 denota l'insieme vuoto, oppure

come la congiunzione logica ($\wedge$) mentre 0 denota il valore false), allora sarebbe del tutto lecito assumere che questa equazione abbia degli unificatori (ad esempio, nel caso della disgiunzione, si vorrebbe che $0 \wedge X$ abbia lo stesso significato di $X \wedge 0 \wedge 0$).

Quindi se l'unificabilità garantisce che comunque si interpretino i simboli, due termini denoteranno oggetti "in qualche modo" uguali, la non unificabilità non garantisce il contrario: ciò dipende anche dalla scelta dell'interpretazione.

L'Esempio 2.7 mostra come le interpretazioni di Herbrand agiscano in modo più restrittivo. Tale intuizione è confermata dal seguente teorema. Esso dimostra che se un insieme di equazioni è soddisfacibile nell'universo di Herbrand allora lo è in qualsiasi interpretazione. Inoltre, stabilisce che l'esistenza di un unificatore per un insieme di equazioni ne implica la soddisfacibilità in qualsiasi interpretazione.

Teorema 4.3. *Sia C un sistema di equazioni tale che $vars(C) = \{X_1, \dots, X_n\}$. Se $\mathcal{F}$ contiene almeno un simbolo di costante, posto $\mathcal{H}$ l'universo di Herbrand, i tre seguenti enunciati sono sono equivalenti:*

(1) *C è unificabile;*
(2) $\models \exists X_1 \cdots X_n\, C$;
(3) $\mathcal{H} \models \exists X_1 \cdots X_n\, C$.

Dim. (1) $\Rightarrow$ (2): Se C è unificabile, allora esiste θ t.c. $s\theta \equiv t\theta$ per ogni $s = t$ in C. Sia $\mathcal{A} = \langle A, (\cdot)^{\mathcal{A}} \rangle$ una generica struttura. Siano inoltre $vars(C\theta) = \{Y_1, \cdots, Y_n\}$ e σ una qualunque funzione di assegnamento tale che $\sigma : \{Y_1, \cdots Y_n\} \to A$. Si può mostrare per induzione sulla struttura di $s\theta$ che $s\theta \equiv t\theta$ implica che $\mathcal{A} \models (C\theta)\sigma$. Ma $(C\theta)\sigma = C(\theta\sigma)$ pertanto abbiamo che $\mathcal{A} \models \exists X_1 \cdots X_n C$.

(2) $\Rightarrow$ (3): Immediato. Per definizione $\models \exists X_1 \cdots X_n(C)$ significa che la formula è soddisfatta per ogni interpretazione. Quindi anche in $\mathcal{H}$.

(3) $\Rightarrow$ (1): Per definizione, $\mathcal{H} \models \exists X_1 \cdots X_n(C)$ significa che esistono $t_1, \dots, t_n$ termini ground tali che $\mathcal{H} \models C[X_1/t_1, \cdots, X_n/t_n]$ e quindi si ha che per ogni equazione $s = t$ in C vale $s\theta \equiv t\theta$, con $\theta = [X_1/t_1, \cdots, X_n/t_n]$. Conseguentemente, θ è un unificatore di C. $\square$

Esempio 4.8. Poniamoci ora il problema di stabilire in che modo le seguenti formule siano logicamente connesse:

$$f(s_1, \cdots, s_n) = f(t_1, \cdots, t_n) \quad \text{e} \quad s_1 = t_1 \wedge \cdots \wedge s_n = t_n$$

L'implicazione

$$s_1 = t_1 \wedge \cdots \wedge s_n = t_n \;\to\; f(s_1, \cdots, s_n) = f(t_1, \cdots, t_n)$$

è sempre verificata grazie alle proprietà dell'identità.[2] L'implicazione

$$s_1 = t_1 \wedge \cdots \wedge s_n = t_n \;\leftarrow\; f(s_1, \cdots, s_n) = f(t_1, \cdots, t_n)$$

invece non è sempre verificata. Questo si può verificare, ad esempio, osservando che

$$1 + 3 = 3 + 1 \qquad \text{ma} \qquad 1 \neq 3 \text{ e } 3 \neq 1$$

[2]Usualmente si assumono rispettate le seguenti proprietà dell'uguaglianza: Riflessività: $\forall x\,(x = x)$. Sostitutività: $\forall x\,\forall y\,(x = y \to (\varphi \to \varphi'))$, dove la formula φ' è ottenuta da φ sostituendo alcune delle occorrenze di x con y.

Ciò che questo esempio suggerisce discende in realtà da una proprietà generale. Vale infatti che:

$$\models \overrightarrow{\forall}(\mathsf{Unify}(C) \to C). \tag{4.1}$$

Questa proprietà tuttavia non è del tutto soddisfacente. Ciò che si vorrebbe garantire infatti è che qualora l'algoritmo $\mathsf{Unify}(\cdot)$ termini esso produca un sistema che sia equivalente al sistema di partenza. Per ottenere questo risultato Clark nel '78, riprendendo un lavoro di Mal'cev propose la seguente una teoria equazionale

Definizione 4.10 (Clark's Equality Theory). La teoria nota come CET è definita dal seguente schema di assiomi:

- **(F1):** $f(t_1, \cdots, t_n) = f(s_1, \cdots, s_n) \to t_1 = s_1 \wedge \cdots \wedge t_n = s_n$
- **(F2):** $f(t_1, \cdots, t_n) \neq g(s_1, \cdots, s_m)$ se $f \neq g$
- **(F3):** $X \neq t[X]$: ogni termine X è diverso da ogni termine t che contenga X come sottotermine proprio.

Assumendo questi tre assiomi si dimostra il seguente risultato che rafforza (4.1):

Teorema 4.4. *Sia C un sistema di equazioni. Allora $CET \models \overrightarrow{\forall}(C \leftrightarrow \mathsf{Unify}(C))$.*

5. E-unificazione

Abbiamo visto che due termini sono unificabili se esiste una sostituzione che li rende sintatticamente identici. Abbiamo anche detto che in alcuni casi sarebbe necessario disporre di un criterio meno rigido della pura identità sintattica, per dichiarare che due termini debbano essere interpretati nello stesso modo. Vedremo ora un raffinamento della nozione di unificabilità che risponde a questa esigenza.

In presenza di una teoria E che coinvolga i termini, potremmo pensare ad un concetto più generale di unificabilità, che chiameremo E-unificabilità: due termini saranno E-unificabili quando esiste una sostituzione che li renda equivalenti nella teoria. Si tratta pertanto di stabilire cosa significhi per due termini s e t essere "equivalenti nella teoria":

(1) un primo approccio potrebbe essere quello di verificare se esista una sequenza finita di applicazioni di regole deducibili da E tali che $s \longrightarrow_* t$;
(2) un altro approccio invece prevede di partizionare l'universo di Herbrand in classi di equivalenza (in accordo con la relazione di equivalenza indotta da E) e dichiarare due termini s e t equivalenti se appartengono alla stessa classe di equivalenza.

In quanto segue affronteremo tali problematiche relativamente ad una particolare classe di teorie dette *equazionali*. Concetti analoghi tuttavia possono essere studiati nel contesto più generale in cui E è una qualsiasi teoria del prim'ordine.

Esempio 4.9. Consideriamo la seguente teoria che esprime le proprietà del simbolo funzionale binario $\cup$:

$$\begin{array}{lrcl} (A) & (X \cup Y) \cup Z & \approx & X \cup (Y \cup Z) \\ (C) & X \cup Y & \approx & Y \cup X \\ (I) & X \cup X & \approx & X \end{array}$$

Ci chiediamo se $a \cup b$ debba essere considerato equivalente al termine $b \cup a$ o meno. Se adottassimo come criterio l'unificazione sintattica ($\mathsf{Unify}(\cdot)$), allora dovremmo dare una risposta

negativa. Tuttavia, rispetto alla teoria ACI si ha che i due termini sono equivalenti. Infatti i due termini $a \cup b$ e $b \cup a$ saranno interpretati con lo stesso oggetto in qualsiasi interpretazione che soddisffi l'assioma di commutatività C. Si noti anche che l'assioma stesso suggerisce una regola di trasformazione che applicata al primo termine produce il secondo.
Similmente abbiamo $(a \cup b) \cup b \overset{ACI}{=} b \cup a$, infatti sfruttando gli assiomi ACI possiamo effettuare la seguente successione di trasformazioni:

$$(a \cup b) \cup b \xrightarrow{A} a \cup (b \cup b) \xrightarrow{I} a \cup b \xrightarrow{C} b \cup a$$

La nozione di E-unificabilità estende pertanto quella di unificabilità sintattica vista nei paragrafi precedenti. La sua generalità riapre però le domande a cui avevamo dato risposta nel caso sintattico. Ad esempio, esiste sempre un E-unificatore? se esiste è unico?

Esempio 4.10. Cerchiamo gli ACI-unificatori della seguente equazione:

$$X \cup Y = a \cup b$$

Tale equazione ammette gli ACI-unificatori

$$[X/a, Y/b] \quad \text{e} \quad [X/b, Y/a]$$

(in realtà ne ammette anche molti altri, come vedremo nell'Esempio 4.12). Si noti come i due ACI-unificatori siano tra loro indipendenti. Inoltre non appare possibile avere un ACI-unificatore più generale di nessuno dei due (e dunque nessuno più generale di entrambi). Parrebbe qiondi che, per questa teoria, descrivere tutte le soluzioni con un unico m.g.u., a meno di varianti, non sia possibile. Inoltre, le seguenti sostituzioni:

$$[X/a, Y/b],\ [X/a \cup a, Y/b],\ [X/a \cup a \cup a, Y/b],\ [X/a \cup a \cup a \cup a, Y/b],\ \ldots$$

sembrano essere infiniti ACI-unificatori del problema di partenza.

Dal precedente esempio scopriamo quindi che il problema della E-unificabilità è in generale molto più complesso della unificazione sintattica: anche per esempi semplici come il precedente, possono esistere un numero infinito E-unificatori e potrebbe non esistere un unico E-unificatore più generale. Allora ci possiamo chiedere se sia possibile generalizzare la nozione di m.g.u., ovvero se esista un insieme, se non unitario per lo meno finito, di E-unificatori in grado di descrivere tutti gli E-unificatori?
Prima di rispondere a questa domanda formalizziamo alcuni concetti relativi alle teorie equazionali.

Definizione 4.11. Una *teoria equazionale* è un insieme di identità del tipo $s \approx t$.

Le identità di una teoria equazionale vanno intese come assiomi del tipo $\vec{\forall}(s = t)$. Ad esempio $X \cup Y \approx Y \cup X$ va inteso come $\forall X \forall Y(X \cup Y = Y \cup X)$.

Definizione 4.12. Indichiamo con $=_E$ la più piccola relazione di congruenza chiusa per sostituzione su $T(\mathcal{F}, \mathcal{V})$ (e dunque su $T(\mathcal{F})$).

La relazione $=_E$ ci permette di partizionare $T(\mathcal{F}, \mathcal{V})$ e $T(\mathcal{F})$ in classi di equivalenza. La scelta di prendere la più piccola congruenza permette di evitare di mettere tutto nella stessa classe senza contraddire gli assiomi (Si osservi infatti che la scelta banale in cui ogni cosa é equivalente a ogni altra cosa non viola nessun assioma di alcuna teoria equazionale).

Raffiniamo ora la nozione di interpretazione cercando un omologo alla interpretazione di Herbrand. Le precedenti definizioni suggeriscono la seguente interpretazione

$$\mathcal{H}_E = \langle T(\Sigma)/ =_E, (\cdot)^{\mathcal{H}_E} \rangle.$$

Essa mappa ogni termine nella propria classe di equivalenza secondo la teoria E. Tale interpretazione prende il nome di *algebra iniziale* e viene anche indicata con $T(\Sigma)/ =_E$. Si può dimostrare che, dati s e t termini ground, le seguenti nozioni sono equivalenti:

- $E \models s = t$
- $s =_E t$
- $\mathcal{H}_E \models s = t$

Definiamo ora formalmente il concetto di E-unificatore e di E-unificatore più generale.

Definizione 4.13. Un *E-unificatore* di un sistema C è una sostituzione θ tale che

$$s\theta =_E t\theta \text{ per ogni equazione } s = t \text{ in } C$$

Diremo che C è *E-unificabile* se esiste un E-unificatore per C. L'insieme di tutti gli E-unificatori di C viene indicato con $\mathcal{U}_E(C)$.

Definizione 4.14. Sia E una teoria equazionale e $\mathcal{X}$ un insieme di variabili, allora σ è *più generale di θ modulo E su $\mathcal{X}$* (e scriveremo: $\sigma \leq_E^{\mathcal{X}} \theta$) se esiste una sostituzione η tale che $X\theta =_E X\sigma\eta$ per ogni $X \in \mathcal{X}$.
Qualora si abbia $\sigma \leq_E^{\mathcal{X}} \theta$, si dice anche che θ è una E-istanza di σ.

Un esempio chiarirà la precedente definizione:

Esempio 4.11. Sia E la teoria ACI. Consideriamo il sistema di una sola equazione:

$$X \cup a = X \cup b.$$

Abbiamo:

(1) $[X/a \cup b]$ è un ACI-unificatore;
(2) $[X/a \cup b \cup b]$ è un ACI-unificatore ma, poiché $a \cup b \cup b \stackrel{ACI}{=} a \cup b$ si ha che l'ACI-unificatore (1) è istanza di (2) e viceversa;
(3) $[X/a \cup b \cup c]$ è un altro ACI-unificatore, ma (1) e (3) sono indipendenti.

Esisterà nel caso dell'esempio precedente un ACI-unificatore più generale di (1) e (3)? Consideriamo $\theta = [X/a \cup b \cup N]$ con N *nuova variabile*. Allora avremo che:

- $[X/a \cup b \cup N][N/a] = [X/a \cup b \cup a, N/a]$ che è una estensione di (1).
- $[X/a \cup b \cup N][N/c] = [X/a \cup b \cup c, N/c]$ che è una estensione di (3).

Quindi abbiamo trovato una sostituzione θ che "in qualche modo" è in grado di surrogare sia (1) che (3). Notiamo tuttavia che θ, tramite istanziazione, permette solo di ottenere delle estensioni di (1) o di (3). Ciò è causato dalla presenza della variabile N, che non è una delle variabili occorrenti nel sistema iniziale.

Tuttavia in pratica è molto più accettabile ammettere in θ la presenza di nuove variabili, adottando l'accortezza di restringere successivamente le sue istanze all'insieme delle variabili del sistema, piuttosto che necessitare di insiemi infiniti di E-unificatori per collezionare tutte le risposte. Ciò giustifica la presenza dell'insieme $\mathcal{X}$ nella Definizione 4.14.

Definizione 4.15. Dato un sistema di equazioni C ed una teoria equazionale E, un *insieme completo* di E-unificatori per C, è un insieme S di sostituzioni tali che

(1) $S \subseteq \mathcal{U}_E(C)$
(2) per ogni $\theta \in \mathcal{U}_E(C)$ esiste $\mu \in S$ t.c. $\mu \leq_E^{vars(C)} \theta$,

Un insieme completo è *minimale* e si denota con $\mu\mathcal{U}_E(C)$ se vale inoltre:

(3) ogni coppia σ, μ di elementi distinti di S è incomparabile, ovvero
$$\sigma \not\leq_E^{vars(C)} \mu \quad \text{e} \quad \mu \not\leq_E^{vars(C)} \sigma.$$

Esempio 4.12. Sia $X \cup Y = a \cup b$ come nell'Esempio 4.12. Gli E-unificatori sono:

(1) $[X/a, Y/b], [X/a \cup a, Y/b, [X/a, Y/b \cup b], \ldots$
(2) $[X/b, Y/a], [X/b \cup b, Y/a], [X/b, Y/a \cup a], \ldots$
(3) $[X/a \cup b, Y/a], [X/a \cup b, Y/a \cup a], \ldots$
(4) $[X/a \cup b, Y/b], \ldots$
(5) $[X/a, Y/a \cup b], \ldots$
(6) $[X/b, Y/a \cup b], \ldots$
(7) $[X/a \cup b, Y/a \cup b], [X/a \cup b \cup a, Y/a \cup b], \ldots$

L'insieme $\mathcal{U}_E(C)$ è infinito. Se prendiamo l'insieme costituito dalle prime sostituzioni di ogni riga, si ottiene un insieme completo e minimale di E-unificatori.

Definizione 4.16. Data una teoria equazionale E ed un sistema C, il *Problema di E-unificazione* è articolato in tre punti:

(1) dire se esiste un E-unificatore (problema decisionale).
(2) fornire un insieme completo di E-unificatori.
(3) fornire un insieme completo minimale $\mu\mathcal{U}_E(C)$.

Data una teoria equazionale E, il problema di E-unificazione si dice di tipo:

1: (unitario) se per ogni C t.c. $\mathcal{U}_E(C) \neq \emptyset$ (ovvero il sistema C ammette almeno un E-unificatore), allora esiste $\mu\mathcal{U}_E(C)$ tale che $|\mu\mathcal{U}_E(C)| = 1$ (ovvero, è rappresentabile con un solo E-unificatore).
ω: (finitario) se per ogni C t.c. $\mathcal{U}_E(C) \neq \emptyset$, esiste $\mu\mathcal{U}_E(C)$ di cardinalità finita (ed inoltre non è semplicemente di tipo 1).
∞: (infinitario) se per ogni C t.c. $\mathcal{U}_E(C) \neq \emptyset$, esiste $\mu\mathcal{U}_E(C)$ di cardinalità infinita (ed esiste almeno un sistema C per cui non esiste $\mu\mathcal{U}_E(C)$ finito).
0: se esiste almeno un sistema C per cui $\mathcal{U}_E(C) \neq \emptyset$ e $\mu\mathcal{U}_E(C)$ non esiste.

Esercizio 4.2. Per ognuno dei quattro *tipi* si fornisca almeno un esempio di teoria. Nell'ultimo caso (tipo 0) si consiglia la lettura della letteratura (ad esempio [**BS98**]).

5.1. *ACI*-unificazione. Per concludere questo capitolo, analizziamo il problema decisionale relativo alla teoria ACI. Consideriamo il problema di decisione prima nel caso in cui il lingiaggio della teoria contenga solo simboli di costante e poi nel caso in cui contenga uno o più simboli di funzione con arità non nulla.

Assumiamo pertanto che la teoria contenga solo *simboli di costante.* Si consideri il problema di unificazione:

$$X \cup a \cup b \cup Y = c \cup X \cup b$$

Costruiamo una sostituzione nel modo seguente: ad ogni variabile si assegni il termine costituito dall'unione di tutte le costanti che occorrono nel problema:

$$[X/a \cup b \cup c, Y/a \cup b \cup c]$$

È immediato verificare che tale sostituzione è un ACI-unificatore. Ciò che si dimostra è che in presenza di variabili occorrenti in entrambi i termini della equazione, questa "ricetta" per costruire ACI-unificatori può essere facilmente generalizzata. Diamo qui di seguito un criterio per risolvere il problema di decisione per un sistema costituito da un'unica equazione $\ell = r$:

(1) $vars(\ell) \neq \emptyset$ e $vars(r) \neq \emptyset$: esiste sempre un ACI-unificatore (costruito come sopra).
(2) $vars(\ell) \neq \emptyset$ e $vars(r) = \emptyset$: esiste un ACI-unificatore se e solo se ogni costante che occorre in ℓ occorre anche in r.
(3) $vars(\ell) = \emptyset$ e $vars(r) \neq \emptyset$: analogo al punto precedente (con i ruoli di ℓ e r scambiati).
(4) $vars(\ell) = \emptyset$ e $vars(r) = \emptyset$: esiste un ACI-unificatore se i termini ℓ e r contengono esattamente le stesse costanti.

Pertanto possiamo concludere che il problema di decisione dell'ACI-unificazione nel caso di segnatura con costanti ammette soluzione in tempo polinomiale.

Vediamo come cambia la situazione quando nel linguaggio della teoria ammettiamo la presenza di simboli di funzione (ne basta uno solo, come vedremo). Si consideri il problema NP-completo 3-SAT: *Data una formula logica proposizionale in forma normale congiuntiva in cui ogni congiunto è una disgiunzione di tre letterali, determinare se esiste un assegnamento di verità per le variabili proposizionali che soddisfi la formula.*

Mostriamo come il problema 3-SAT si possa ridurre ad un problema di ACI-unificazione. Consideriamo la formula

$$\varphi = (X_1 \vee \neg X_2 \vee X_3) \wedge (\neg X_1 \vee X_2 \vee X_3) \wedge (X_1 \vee \neg X_2 \vee \neg X_3).$$

Osserviamo che nel problema di ACI-unificazione:

$$\{\{X_i\} \cup \{Y_i\}\} = \{\{0\} \cup \{1\}\}$$

si vincola una variabile ad assumere valore 0 e l'altra ad assumere valore 1 (si pensi all'interpretazione insiemistica dei termini). Possiamo sfruttare ciò per modellare con dei termini i valori di verità e variabili proposizionali del problema 3-SAT. Possiamo infatti pensare che i termini (costanti del linguaggio) 0 e 1 corrispondano ai valori di verità false e true, e inoltre mettere in corrispondenza le variabili del linguaggio con i valori assunti dalle variabili proposizionali nel problema 3-SAT. Per l'equazione sopra si avrà quindi che il valore di verità (corrispondente a) Y_i è vincolato ad essere la negazione del valore di verità (corrispondente a) X_i.

Adottando questo approccio otteniamo che il seguente problema di ACI-unificazione modella esattamente 3-SAT. L'annidamento di insiemi è stato ottenuto mediante l'utilizzo di un simbolo di funzione unario $\{\cdot\}$ che permette di descrivere un insieme contenente un solo elemento (singoletto).

$$\begin{array}{l} \{\{\{X_1\} \cup \{Y_1\}\} \cup \{\{X_2\} \cup \{Y_2\}\} \cup \{\{X_3\} \cup \{Y_3\}\} \cup \\ \{X_1\} \cup \{Y_2\} \cup \{X_3\} \cup \{0\}\} \cup \{\{Y_1\} \cup \{X_2\} \cup \{X_3\} \cup \{0\}\} \cup \{\{X_1\} \cup \{Y_2\} \cup \{Y_3\} \cup \{0\}\} = \\ \{\{0\} \cup \{1\}\} \end{array}$$

Si dimostra che ogni soluzione del problema di ACI-unificazione è infatti un assegnamento di successo per φ e viceversa.

Pertanto il problema di decisione per l'ACI-unificazione con simboli di funzione (ne basta uno e di arità unitaria) è NP-hard. Per approfondimenti su queste tematiche si veda ad esempio [**DPPR00**].

6. Esercizi

Esercizio 4.3. Per ogni coppia di termini dell'Esercizio 2.7 si determini, qualora esista, un most general unifier.

Esercizio 4.4. Scrivere un termine t e due sostituzioni σ_1 e σ_2 tali che $\sigma_1\sigma_2 \neq \sigma_2\sigma_1$ e $t\sigma_1\sigma_2 = t\sigma_2\sigma_1$.

Esercizio 4.5. Scrivere tre letterali L_1, L_2, e L_3 tali che:

- a due a due unifichino: per ogni $i, j \in \{1, 2, 3\}$ esista l'm.g.u. $\sigma_{i,j}$ di L_i, e L_j.
- non esista alcuna sostituzione che unifichi simultaneamente tutti tre i letterali.

Scrivere $\sigma_{1,2}$, $\sigma_{2,3}$, e $\sigma_{1,3}$.

CAPITOLO 5

SLD-risoluzione

Nel Capitolo 3 abbiamo visto alcuni esempi elementari di programmazione con clausole definite. Riprendiamo uno di quegli esempi, ovvero il programma definito costituito dalle clausole

```
(1)  padre(antonio,bruno).
(2)  padre(antonio,carlo).
(3)  padre(bruno,davide).
(4)  padre(bruno,ettore).
(5)  antenato(X,Y) :- padre(X,Y).
(6)  antenato(X,Y) :- antenato(X,Z), padre(Z,Y).
```

ed il goal `?-antenato(antonio,Y)`. Abbiamo illustrato che l'interprete Prolog risponde a questo goal con la risposta calcolata:

```
yes Y=bruno
```

In questo capitolo illustreremo come si proceda per inferire tale risposta a partire dalla congiunzione di programma e goal.

Prima di fornire una trattazione formale della semantica operazionale del Logic Programming ne daremo una descrizione intuitiva, cercando così di suggerire le idee cardine del metodo di risoluzione impiegato dall'interprete Prolog, una delle implementazioni più rappresentative del Logic Programming.

Per rispondere ad un goal, Prolog procede costruendo (o cercando di farlo) una successione di passi di inferenza, ovvero una successione di trasformazioni del goal fornito. Tale processo continua fino a che non sia più possibile effettuare alcuna trasformazione.

Ogni passo avviene:

(1) selezionando uno dei letterali del goal;
(2) scegliendo opportunamente una delle clausole del programma;
(3) combinando i letterali del goal e quelli della clausola per ottenere un nuovo goal.

Tale processo può terminare solo se al punto (1) non ci sono letterali selezionabili (cioè se il goal è vuoto: $\leftarrow \Box$); oppure se al punto (2) nessuna delle clausole del programma permette di effettuare il passo (3).

Questa forma di ragionamento, in cui a partire dal goal si procede "all'indietro", è detta *backward chaining*.

Anticipiamo qui che al punto (1) l'interprete Prolog seleziona sempre l'atomo più a sinistra, mentre al punto (2) la ricerca di una clausola "adeguata" procede seguendo l'ordine in cui le clausole sono elencate nel programma (vedremo successivamente le ragioni che giustificano la scelta di questa particolare strategia, rispetto a tutte le possibili alternative).

Non abbiamo specificato cosa significhi che una clausola è "adeguata" ad essere selezionata. Una volta selezionato l'atomo nel goal attuale, una clausola si ritiene adeguata se la sua testa unifica con questo atomo (sia μ il corrispondente m.g.u.).

Un altro punto ancora non chiarito nella nostra descrizione informale del processo risolutivo, è come si produca il nuovo goal. Quest'ultimo viene ottenuto dal goal attuale sostituendo al posto dell'atomo selezionato il corpo della clausola scelta, ed istanziando tramite μ la congiunzione così ottenuta.

Relativamente all'esempio sopra menzionato, vediamo come viene processato il goal

```
?-antenato(antonio,Y).
```

Questo goal ha un solo letterale che quindi viene selezionato. Si cerca nel programma una clausola la cui testa unifichi con l'atomo `antenato(antonio,Y)`. Tale clausola (la prima con questa proprietà ad essere individuata procedendo dall'alto verso il basso) è (5). Dato che nei passi successivi della derivazione potremmo dover riutilizzare questa clausola, per evitare conflitti tra nomi di variabili effettuiamo una rinomina di tutte le variabili della clausola (5). Consideriamo, così facendo, una sua rinomina, in cui occorrano solo variabili "nuove", ad esempio:

(5′) `antenato(X1,Y1) :- padre(X1,Y1).`

La sostituzione $[X_1/antonio, Y_1/Y]$ è un m.g.u. di $\{antenato(antonio, Y), antenato(X_1, Y_1)\}$ Il nuovo goal sarà quindi:

```
?- padre(antonio,Y).
```

Ora il procedimento continua selezionando l'unico atomo, `padre(antonio,Y)`, del nuovo goal. Si cerca quindi una clausola la cui testa unifichi con questo atomo. La prima clausola con questa proprietà è il fatto (1). Essendo un fatto ground non è necessario effettuare la rinomina delle variabili e l'unificatore in questo caso è $[Y/bruno]$. La costruzione del nuovo goal genera il goal vuoto $\leftarrow \Box$ che mette fine alla ricerca della prima risposta al goal. Tale risposta può essere infatti desunta restringendo, alle variabili del goal iniziale, la composizione di tutti gli m.g.u. intervenuti nel processo di derivazione. Ciò che otteniamo è: $[Y/bruno]$.

Questa sequenza di (due) passi di derivazione può essere visualizzata come segue:

$$\cfrac{\leftarrow antenato(antonio, Y) \qquad antenato(X_1, Y_1) \leftarrow padre(X_1, Y_1)}{\cfrac{\leftarrow padre(antonio, Y) \qquad padre(antonio, bruno)}{\leftarrow \Box}\ [Y/bruno]}\ [X_1/antonio, Y_1/Y]$$

In questo caso il processo risolutivo è terminato perchè il goal $\leftarrow \Box$ non contiene ulteriori atomi selezionabili. In tale situazione l'interprete Prolog ha modo di ottenere, dalle sostituzioni impiegate nella derivazione, una risposta al goal. Come abbiamo detto in precedenza, c'è un altro caso in cui non è possibile procedere con ulteriori passi di derivazione. Ciò accade quando, pur essendoci atomi nel goal attuale, non vi sono clausole utilizzabili per effettuare ulteriori passi di inferenza. Questo è un caso in cui l'interprete fallisce (temporaneamente) nella ricerca di soluzioni. Vedremo nelle prossime pagine come l'interprete

Prolog possa comunque continuare la ricerca. L'idea base sarà quella di "disfare" uno o più degli ultimi passi di inferenza effettuati e riprendere il processo modificando la scelta delle clausole selezionate.

1. SLD-derivazioni

Passiamo ora a descrivere formalmente la semantica operazionale del Logic Programming. Questa è basata sul concetto di SLD-risoluzione (SLD-derivazione) dove

S: sta per *Selection* function,
L: sta per *Linear* resolution e
D: sta per *Definite* clauses.

DEFINIZIONE 5.1. Dato un atomo A_i ed una clausola $C \equiv H \leftarrow B_1, B_2, \ldots, B_n$ in un programma definito P, si dice che C è *applicabile* ad A_i se presa comunque una rinomina $H' \leftarrow B'_1, B'_2, \ldots, B'_n$ di C con variabili non occorrenti in A_i, $\mathsf{Unify}(A_i = H') \neq \mathsf{false}$.

La notazione $\mathsf{Unify}(A_i = H')$ non è del tutto corretta. Infatti abbiamo introdotto l'algoritmo $\mathsf{Unify}(\cdot)$ in relazione alla unificazione di sistemi di equazioni, mentre A_i e H' sono atomi e non termini. Possiamo tuttavia, anche considerando Definizione 4.3, permetterci questo abuso di notazione.

Definiamo il passo di SLD-derivazione:

DEFINIZIONE 5.2. Sia G un goal $\leftarrow A_1, \ldots, \ldots, A_n$ in cui occorre l'atomo A_i, e sia $C \equiv H \leftarrow B_1, \ldots, B_m$ una clausola applicabile ad A_i. Allora scriveremo

$$\frac{\leftarrow A_1, \ldots, A_{i-1}, A_i, A_{i+1}, \ldots, A_n \qquad H \leftarrow B_1, \ldots, B_m}{\leftarrow (A_1, \ldots, A_{i-1}, B_1, \ldots, B_m, A_{i+1}, \ldots, A_n)\theta} \; \theta = mgu(A_i, H)$$

per indicare che il goal $G' \equiv \leftarrow (A_1, \ldots, A_{i-1}, B_1, \ldots, B_m, A_{i+1}, \ldots, A_n)\theta$ è ottenuto per SLD-risoluzione da G e C. Il goal G' è detto SLD-risolvente di G e C.

Il passo di inferenza procede quindi selezionando un'atomo A_i del goal G ed una clausola C applicabile ad A_i. La clausola C viene rinominata con variabili nuove Si calcola a questo punto un m.g.u. θ di A_i con la testa della clausola rinominata e si costruisce il risolvente G' di G e C rispetto ad A_i, con m.g.u. θ. Si osservi che per semplicità nella precedente definizione il fatto che in luogo di C venga utilizzata una rinomina di $C\sigma$ in cui occorrono solo variabili nuove è stato lasciato implicito. In quanto segue ricorreremo spesso a questa convenzione per semplificare la notazione.

Nell'effettuare un passo di SLD-risoluzione vi sono diversi punti in cui possono essere fatte, in modo arbitrario, delle scelte:

(1) Scelta del letterale A_i del goal.
(2) Scelta della clausola C di input.
(3) Scelta della rinomina $C\sigma$ di C.
(4) Scelta dell'm.g.u. θ.

Una volta effettuate queste scelte il passo si compie rimpiazzando al posto di A_i il corpo della rinomina $C\sigma$ e istanziando il tutto tramite θ. Come vedremo, il punto più complesso risulterà essere è il calcolo dell'unificatore (secondo il metodo che abbiamo studiato nel Capitolo 4).

Un passo di SLD-risoluzione può essere rappresentato sinteticamente con la seguente notazione:

$$G \overset{\theta}{\Rightarrow}_C G',$$

con G' goal ottenuto a partire dal goal G utilizzando una (rinomina della) clausola C e la sostituzione θ. Anche in questo caso, solitamente si indica la clausola C come appare nel programma e non la sua rinomina.

ESEMPIO 5.1. Consideriamo il semplice programma:

```
num(s(X)) :- num(X).
```

ed il goal G:

```
?-num(s(0)).
```

Un primo passo di SLD-risoluzione sarà il seguente:

$$\frac{\leftarrow num(s(0)) \qquad num(s(X_1)) \leftarrow num(X_1)}{\leftarrow num(0)} \; \theta = [X_1/0]$$

DEFINIZIONE 5.3. Una *SLD-derivazione* per un programma P ed un goal G (in breve, di $P \cup \{G\}$) è una sequenza *massimale* di passi di derivazione nella forma:

$$G \overset{\theta_1}{\Rightarrow}_{C_1} G_1 \overset{\theta_2}{\Rightarrow}_{C_2} G_2 \overset{\theta_3}{\Rightarrow}_{C_3} \dots (\overset{\theta_n}{\Rightarrow}_{C_n} G_n)$$

che può essere finita o infinita e tale che:

(1) $G_1, G_2, \dots$ sono goal (anche $G = G_0$ è un goal),
(2) $\theta_1, \theta_2, \dots, \theta_{n-1}$ sono sostituzioni,
(3) le clausole scelte dal programma sono indicate con C_i: esse vengono di volta in volta rinominate opportunamente in C_i' in modo tale che valga quanto segue: $vars(C_i') \cap \Big(vars(G_{i-1}) \cup \bigcup_{j=1}^{i-1} \big(vars(\theta_j) \cup vars(C_j')\big)\Big) = \emptyset$.
(4) $G_i \overset{\theta_{i+1}}{\Rightarrow}_{C_{i+1}} G_{i+1}$ è un passo di SLD-derivazione.

Per "massimale" si deve intendere che qualora esista almeno una clausola in P che sia applicabile all'atomo correntemente selezionato, l'unificazione deve essere operata e il passo di SLD-derivazione eseguito.

DEFINIZIONE 5.4. La *lunghezza* di una SLD-derivazione è data dal numero di passi che la compongono. La lunghezza di una SLD-derivazione può quindi essere finita o infinita.

Si osservi che, per il requisito di massimalità, se una SLD-derivazione ha lunghezza finita n, allora vale una delle seguenti proprietà:

- G_n è il goal vuoto $\leftarrow \Box$, oppure
- per ogni atomo di G_n non esistono clausole di P applicabili.

DEFINIZIONE 5.5. Sia data una SLD-derivazione finita ξ del tipo

$$\xi : \; G \overset{\theta_1}{\Rightarrow}_{C_1} G_1 \overset{\theta_2}{\Rightarrow}_{C_2} G_2 \overset{\theta_3}{\Rightarrow}_{C_3} \cdots \overset{\theta_n}{\Rightarrow}_{C_n} G_n.$$

Si dice che ξ è di *successo* se G_n è $\leftarrow \Box$. In questo caso la sostituzione $(\theta_1\theta_2 \dots \theta_n) \mid_{vars(G)}$ si dice (sostituzione di) *risposta calcolata (cas)*.

Se $G_n \not\equiv \leftarrow \Box$ allora ξ è una derivazione di *fallimento*.

Si osservi che le derivazioni infinite non sono né di successo né di fallimento.

Esempio 5.2. Consideriamo ora alcune derivazioni a partire dal seguente programma P:

```
(1) padre(antonio, bruno).
(2) padre(antonio, carlo).
(3) padre(bruno, davide).
(4) padre(bruno, ettore).
(5) figlio(X, Y) ← padre(Y, X).
(6) nonno(X, Y) ← padre(X, Z), padre(Z, Y).
(7) antenato(X, Y) ← padre(X, Y).
(8) antenato(X, Y) ← antenato(X, Z), padre(Z, Y).
```

relativamente a differenti goal.

- Sia $G \equiv \leftarrow nonno(antonio, davide)$. Sfruttando la clausola (5) abbiamo:

$$\frac{\leftarrow nonno(antonio, davide) \qquad nonno(X_1, Y_1) \leftarrow padre(X_1, Z_1), padre(Z_1, Y_1)}{\leftarrow padre(antonio, Z_1), padre(Z_1, davide)} \theta_1$$

  dove $\theta_1 = [X_1/antonio, Y_1/davide]$ è m.g.u. di $nonno(antonio, davide)$ e $nonno(X_1, Y_1)$. Dal goal così ottenuto, selezionando la clausola (1) e la clausola (3) otteniamo:

$$\frac{\leftarrow padre(antonio, Z_1), padre(Z_1, davide) \qquad padre(antonio, bruno)}{\frac{\leftarrow padre(bruno, davide) \qquad padre(bruno, davide)}{\leftarrow \Box} \theta_3 = \varepsilon} \theta_2 = [Z_1/bruno]$$

  La derivazione è finita ed è di successo. La risposta calcolata è:

$$\theta_1\theta_2\theta_3 \mid_{vars(G)} = [X_1/antonio, Y_1/davide, Z_1/bruno] \mid_{\emptyset} = \varepsilon.$$

- Consideriamo ora il goal

$$G \equiv \leftarrow nonno(X, davide).$$

  e la seguente SLD-derivazione per G:

$$\frac{\leftarrow nonno(X, davide) \qquad nonno(X_1, Y_1) \leftarrow padre(X_1, Z_1), padre(Z_1, Y_1)}{\frac{\leftarrow padre(X_1, Z_1), padre(Z_1, davide) \qquad padre(antonio, bruno)}{\frac{\leftarrow padre(bruno, davide) \qquad padre(bruno, davide)}{\leftarrow \Box} \varepsilon} [X_1/antonio, Z_1/bruno]} [X/X_1, Y_1/davide]$$

  La derivazione è finita e di successo. La risposta calcolata è:

$$[X/antonio, X_1/antonio, Y_1/davide, Z_1/bruno] \mid_X = [X/antonio].$$

- Consideriamo ora lo stesso goal del caso precedente $\leftarrow nonno(X, davide)$, ma verifichiamo come la scelta di clausole diverse pregiudichi l'ottenimento di una

derivazione di successo. Il primo passo è simile:

$$\frac{\leftarrow nonno(X, davide) \qquad nonno(X_1, Y_1) \leftarrow padre(X_1, Z_1), padre(Z_1, Y_1)}{\leftarrow padre(X_1, Z_1), padre(Z_1, davide)} [X/X_1, Y_1/davide]$$

Ora selezionando la clausola (2) possiamo effettuare il passo:

$$\frac{\leftarrow padre(X_1, Z_1), padre(Z_1, davide) \qquad padre(antonio, carlo)}{\leftarrow padre(carlo, davide)} [X_1/antonio, Z_1/carlo]$$

A questo punto non esiste alcuna clausola del programma la cui testa unifichi con $\leftarrow padre(carlo, davide)$. Quindi si ottiene una derivazione finita ma di fallimento.

- consideriamo ora il goal $G \equiv \leftarrow antenato(X, Y)$. Ecco un primo passo di derivazione che impiega una rinomina della clausola (8):

$$\frac{\leftarrow antenato(X, Y) \qquad antenato(X_1, Y_1) \leftarrow antenato(Z_1, Y_1), padre(X_1, Z_1)}{\leftarrow antenato(Z_1, Y_1), padre(X_1, Z_1)} \theta_1$$

con $\theta_1 = [X/X_1, Y/Y_1]$. Supponiamo di selezionare l'atomo $antenato(Z_1, Y_1)$ del nuovo goal e di impiegare di nuovo una rinomina della clausola (8):

$$\frac{\leftarrow antenato(Z_1, Y_1), padre(X_1, Z_1) \qquad antenato(X_2, Y_2) \leftarrow antenato(Z_2, Y_2), padre(X_2, Z_2)}{\leftarrow antenato(Z_2, Y_2), padre(X_2, Z_2), padre(X_1, X_2)} \theta_2$$

con $\theta_2 = [Z_1/X_2, Y_1/Y_2]$. Come si può notare, è possibile, continuando a selezionare l'atomo più a sinistra del goal corrente e utilizzando sempre la clausola (8), generare goal sempre più complessi. La derivazione che si costruisce in questo modo è infinita.

ESERCIZIO 5.1. Si consideri il secondo goal dell'esempio precedente. Supponendo che al primo passo la sostituzione calcolata da $\mathsf{Unify}(\cdot)$ fosse $\theta_1 = [X_1/X, Y_1/davide]$, verificare che la risposta calcolata della SLD-derivazione sarebbe stata la stessa.

2. Indipendenza dal non-determinismo nell'SLD-risoluzione

Vedremo ora, anche tramite degli esempi, se le varie scelte non-deterministiche possano o meno influenzare le risposte calcolate tramite SLD-risoluzione.

2.1. Indipendenza dalla scelta della rinomina della clausola. Riprendiamo il programma dell'Esempio 5.2 e consideriamo il goal

$$\leftarrow nonno(A, B)$$

Ora costruiamo due SLD-derivazioni ξ e ξ' che si differenziano per la scelta della rinomina delle variabili presenti nelle clausole.

Derivazione ξ:

$$\frac{\leftarrow nonno(A, B) \qquad nonno(X_1, Y_1) \leftarrow padre(X_1, Z_1), padre(Z_1, Y_1)}{} [A/X_1, B/Y_1]$$

$$\frac{\leftarrow padre(X_1, Z_1), padre(Z_1, Y_1) \qquad padre(antonio, bruno)}{\leftarrow padre(bruno, Y_1) \qquad padre(bruno, davide)} [X_1/antonio, Z_1/bruno]$$

$$\frac{\leftarrow padre(bruno, Y_1) \qquad padre(bruno, davide)}{\leftarrow \Box} [Y_1/davide]$$

Questa è una derivazione di successo con risposta calcolata $[A/antonio, B/bruno]$.

Derivazione ξ':

$$\frac{\leftarrow nonno(A, B) \qquad nonno(X_2, Y_2) \leftarrow padre(X_2, Z_2), padre(Z_2, Y_2)}{\leftarrow padre(X_2, Z_2), padre(Z_2, Y_2) \qquad padre(antonio, bruno)} [A/X_2, B/Y_2]$$

$$\frac{\leftarrow padre(X_2, Z_2), padre(Z_2, Y_2) \qquad padre(antonio, bruno)}{\leftarrow padre(bruno, Y_2) \qquad padre(bruno, davide)} [X_2/antonio, Z_2/bruno]$$

$$\frac{\leftarrow padre(bruno, Y_2) \qquad padre(bruno, davide)}{\leftarrow \Box} [Y_2/davide]$$

Si noti come nonostante la diversa scelta delle rinomine, si ottenga stessa risposta calcolata.

Un altro esempio:

Esempio 5.3. Consideriamo il seguente programma ove si utilizza il simbolo di funzione binario $lista(\cdot, \cdot)$ ed il simbolo di predicato binario *member*:

```
member(X,lista(X,Z)).
member(X,lista(Y,Z)):- member(X,Z).
```

Possiamo costruire facilmente a partire dal goal $\leftarrow member(A, B)$, due derivazioni identiche salvo che per le rinomine utilizzate per ridenominare le variabili delle clausole. Si possono ottenere così due risposte calcolate, ad esempio:

(1) $[A/X_1, B/lista(Y_1, lista(X_1, Z_1))]$ e
(2) $[A/V_1, B/lista(W_2, lista(V_1, Y_1))]$

Notiamo che le due risposte sono tra loro varianti rispetto alle variabili presenti nel goal: si pensi infatti alla sostituzione variante

$$[X_1/V_1, Y_1/W_2, Z_1/Y_1, V_1/X_1, W_2/Y_1, Y_1/Z_1]$$

che applicata alla prima fornisce la seconda.

Quanto illustrato dall'esempio precedente è in realtà una proprietà generale: diverse selezioni delle rinomine portano a ottenere soluzioni identiche a meno di varianti.

2.2. Indipendenza dalla scelta dell'm.g.u. Consideriamo nuovamente il goal

$$\leftarrow nonno(A, B)$$

e la seconda tra le due derivazioni analizzate sopra. Calcoliamo un diverso m.g.u. al primo passo.

$$\frac{\leftarrow nonno(A, B) \qquad nonno(X_2, Y_2) \leftarrow padre(X_2, Z_2), padre(Z_2, Y_2)}{\leftarrow padre(A, Z_2), padre(Z_2, B) \qquad padre(antonio, bruno)} [X_2/A, Y_2/B]$$

$$\frac{\leftarrow padre(A, Z_2), padre(Z_2, B) \qquad padre(antonio, bruno)}{} [A/antonio, Z_2/bruno]$$

$$\frac{\leftarrow padre(bruno, B) \qquad padre(bruno, davide)}{\leftarrow \Box}\ [B/davide]$$

Si ottiene una risposta calcolata identica alla precedente.

In generale, si osserva che diverse scelte portano alle stesse risposte calcolate o quantomeno a rinomine di esse. Questa intuizione può essere giustificata dal seguente risultato di cui riportiamo solo l'enunciato.

TEOREMA 5.1. *Siano due derivazioni di successo per un goal G che differiscono solo nelle scelte degli m.g.u. (derivazioni* simili*) con risposta calcolata θ e θ'. Allora $G\theta$ e $G\theta'$ sono rinomine.*

2.3. Scelta dell'atomo nel goal. Innanzitutto definiamo formalmente il concetto di *regola di selezione*. Assumendo fissato un dato alfabeto.

DEFINIZIONE 5.6. Sia IF *(Initial Fragment)* l'insieme dei frammenti iniziali di tutte le SLD-derivazioni con ultimo goal non vuoto. R è una *regola di selezione* se è una funzione che, applicata ad un elemento di IF, individua esattamente un atomo dell'ultimo goal. La regola R dunque si basa su tutta la storia della derivazione. Diciamo che una SLD-derivazione ξ è *via* R se tutte le selezioni di atomi in essa effettuate sono in accordo con R.

Si osservi ovviamente che:

- Ogni SLD-derivazione è via R per una qualche R (ad ogni passo avviene una scelta di un qualche atomo. La regola R può essere individuata a posteriori).
- Se si dimostrasse che vi è indipendenza dalla scelta della regola R, ciò permette di adottare la regola più semplice. Per esempio, in Prolog viene sempre scelto l'atomo che si trova più a sinistra (regola *leftmost*).

Nei seguenti esempi indicheremo l'atomo che viene selezionato mediante sottolineatura.

ESEMPIO 5.4. Consideriamo il goal $\leftarrow padre(X, Z), padre(Z, Y)$. Una prima derivazione, con risposta $[X/antonio, Y/davide, Z/bruno]$, è:

$$\frac{\frac{\leftarrow \underline{padre(X, Z)}, padre(Z, Y) \qquad padre(antonio, bruno)}{\leftarrow padre(bruno, Y) \qquad padre(bruno, davide)}\ [X/antonio, Z/bruno]}{\leftarrow \Box}\ [Y/davide]$$

Una seconda SLD-derivazione, con la stessa risposta calcolata, è:

$$\frac{\frac{\leftarrow padre(X, Z), \underline{padre(Z, Y)} \qquad padre(bruno, davide)}{\leftarrow padre(X, bruno) \qquad padre(antonio, bruno)}\ [Z/bruno, Y/davide]}{\leftarrow \Box}\ [X/antonio]$$

Si osservi che sono state impiegate le medesime clausole ma nell'ordine inverso.

Un importante risultato è il seguente lemma. Intuitivamente esso stabilisce che se esiste una SLD-derivazione ξ in cui la regola di selezione dell'atomo sceglie A al passo i e B al

passo j allora esiste anche una SLD-derivazione ξ' che, a meno di varianti, differisce da ξ solo per il fatto di selezionare l'atomo B al passo i e l'atomo A al passo j.

LEMMA 5.1 (Switching lemma). *Sia ξ una SLD-derivazione tale che:*

$$\xi = G_0 \overset{\theta_1}{\Rightarrow}_{c_1} G_1 \overset{\theta_2}{\Rightarrow}_{c_2} \cdots \Rightarrow G_n \overset{\theta_{n+1}}{\Rightarrow}_{c_{n+1}} G_{n+1} \overset{\theta_{n+2}}{\Rightarrow}_{c_{n+2}} G_{n+2} \ldots$$

tale che:

- *G_n ha almeno 2 atomi.*
- *A_i è l'atomo selezionato in G_n.*
- *$A_j\theta_{n+1}$ è l'atomo selezionato in G_{n+1}.*

Allora esiste una SLD-derivazione ξ' tale che:

$$\xi' = G_0 \overset{\theta'_1}{\Rightarrow}_{c_1} G_1 \overset{\theta'_2}{\Rightarrow}_{c_2} \cdots \Rightarrow G_n \overset{\theta'_{n+1}}{\Rightarrow}_{c_{n+2}} G'_{n+1} \overset{\theta'_{n+2}}{\Rightarrow}_{c_{n+1}} G_{n+2} \cdots$$

- *ξ e ξ' coincidono fino al passo n e dal passo $n+2$,*
- *A_j è l'atomo selezionato in G_n in ξ'.*
- *$A_i\theta'_{n+1}$ è l'atomo selezionato in G'_{n+1}.*

DIM. Data una derivazione che includa i passi:

$$[\theta_{n+1} = mgu(A_i, A'_i)] \quad \underline{\leftarrow A_1, \ldots, A_i, \ldots, A_j, \ldots, A_n \qquad A'_i \leftarrow \overline{B_1}}$$

$$[\theta_{n+2} = mgu(A_j\theta_{n+1}, A'_j)] \quad \underline{(\leftarrow A_1, \ldots, \overline{B}_1, \ldots, \overline{A}_j, \ldots, A_n)\theta_{n+1} \qquad A'_j \leftarrow \overline{B}_2}$$

$$\leftarrow (A_1, \ldots, \overline{B}_1, \ldots, \overline{B}_2, \ldots, A_n)\theta_{n+1}\theta_{n+2} \qquad \ldots$$

Dobbiamo dimostrare che ne esiste una che includa i passi:

$$[\theta'_{n+1} = mgu(A_j, A'_j)] \quad \underline{\leftarrow A_1, \ldots, A_i, \ldots, A_j, \ldots, A_n \qquad A'_j \leftarrow \overline{B_2}}$$

$$[\theta'_{n+2} = mgu(A_i\theta'_{n+1}, A'_i)] \quad \underline{(\leftarrow A_1, \ldots, \overline{A}_i, \ldots, \overline{B}_2, \ldots, A_n)\theta'_{n+1} \qquad A'_i \leftarrow \overline{B}_1}$$

$$\leftarrow (A_1, \ldots, \overline{B}_1, \ldots, \overline{B}_2, \ldots, A_n)\theta'_{n+1}\theta'_{n+2} \qquad \ldots$$

tale che:

$$(A_1, \ldots, \overline{B}_1, \ldots, \overline{B}_2, \ldots, A_n)\theta_{n+1}\theta_{n+2} \;=\; (A_1, \ldots, \overline{B}_1, \ldots, \overline{B}_2, \ldots, A_n)\theta'_{n+1}\theta'_{n+2}$$

In altri termini, relativamente alle variabili coinvolte, deve valere che

$$\theta_{n+1}\theta_{n+2} \;=\; \theta'_{n+1}\theta'_{n+2}$$

Possiamo schematizzare così i passi di unificazione effettuati nei due casi:

$$\begin{array}{rcr} A_i = A'_i & \qquad & A_j = A'_j \\ \downarrow \theta_{n+1} & & \downarrow \theta'_{n+1} \\ A_j\theta_{n+1} = A'_j & & A_i\theta'_{n+1} = A'_i \\ \downarrow \theta_{n+2} & & \downarrow \theta'_{n+2} \\ \text{in } (\xi) & & \text{in } (\xi') \end{array}$$

Le due computazioni si possono vedere come due possibili strade non deterministiche nella risoluzione del sistema

$$A_i = A'_i \wedge A_j = A'_j$$

Sappiamo dal Teorema 4.2, che $\theta_{n+1}\theta_{n+2}$ e $\theta'_{n+1}\theta_{n+2}$ sono tra loro varianti e dunque esiste γ variante tale che:

$$\theta_{n+1}\theta_{n+2} \quad = \quad \theta'_{n+1}\theta'_{n+2}\gamma$$

Rimane da dimostrare che impiegando $\mathsf{Unify}(\cdot)$ due volte in ξ' si può ottenere esattamente $\theta_{n+1}\theta_{n+2}$. Possiamo spezzare γ in due parti γ' e γ'' distinguendo tra le variabili in $vars(A'_i)$ o in $vars(A_i) \setminus vars(A_j)\theta'_{n+1}$ e le altre. Le due sostituzioni γ' e γ'' saranno a loro volta due varianti. Inoltre avremo che

$$\theta'_{n+1}\theta'_{n+2}\gamma = \theta'_{n+1}\gamma'\theta'_{n+2}\gamma''$$

La sostituzione $\theta'_{n+1}\gamma'$ sarà variante di θ'_{n+1} mentre $= \theta'_{n+2}\gamma''$ lo sarà di θ'_{n+2}. La prima è una delle possibili soluzioni ottenibili da $\mathsf{Unify}(A_i = A'_i)$, mentre la seconda è una delle possibili soluzioni ottenibili da $\mathsf{Unify}(A_j\theta'_{n+1}\gamma' = A'_j)$. $\square$

Il seguente teorema dimostra che l'esistenza di una SLD-derivazione di successo non dipende dalla regola di selezione dell'atomo adottata.

Teorema 5.2 (Indipendenza dalla Regola di Selezione). *Per ogni SLD-derivazione di successo ξ di $P \cup \{G\}$ e per ogni regola di selezione R, esiste una SLD-derivazione di successo ξ' di $P \cup \{G\}$ via R tale che:*

(1) *Le risposte calcolate da ξ e ξ' sono uguali;*
(2) *ξ e ξ' hanno la stessa lunghezza.*

Dim. Consideriamo la derivazione ξ:

$$\xi : G \equiv G_0 \stackrel{\theta_1}{\Rightarrow}_{C_1} G_1 \Rightarrow \ldots \Rightarrow \underbrace{\stackrel{\stackrel{A_h}{\downarrow}}{G_i} \Rightarrow \ldots G_{i+j-1} \Rightarrow}_{j} \quad \stackrel{\stackrel{A_k\theta_{i+1}\cdots\theta_{i+j}}{\downarrow}}{G_{i+j}} \quad \Rightarrow \cdots \Rightarrow G_n \equiv \leftarrow \square$$

e costruiamo ξ' via R. Assumiamo induttivamente che fino al goal G_i la computazione sia via R. Supponiamo quindi che in G_i venga selezionato l'atomo A_h mentre R selezionerebbe A_k. Lo stesso A_k verrà poi selezionato ad un successivo passo G_{i+j}. Con un numero finito j di applicazioni del Lemma 5.1 possiamo costruire una SLD-derivazione in cui la selezione di A_k avviene al passo i-esimo, rispettando R. Questa procedura viene poi iterata per tutta la lunghezza della SLD-derivazione.
Per dimostrare che il procedimento termina si considera ad ogni suo passo la coppia $\langle n-i, j\rangle$, che codifica la lunghezza del tratto differente tra ξ e ξ' e la distanza fra i due passi che prevedono le selezioni dei due atomi A_h e A_k. La coppia rappresenta quindi una valutazione di quanto siano "differenti" la derivazione data e quella che rispetta R. Una coppia del tipo $\langle 0, \cdot\rangle$ rappresenterà quindi il raggiungimento dell'obbiettivo. Una successione di coppie via via lessicograficamente più prossime a $\langle 0, \cdot\rangle$ rappresenta una successione di SLD-derivazioni via via piú "vicine" a ξ. Si possono presentare le due seguenti situazioni:

(1) $\langle 0, \cdot\rangle$. In questo caso ξ è via R;
(2) $\langle i, j\rangle$. In questo caso, applicando lo *Switching Lemma* (Lemma 5.1) si ottiene una SLD-derivazione ξ'' alla quale può corrispondere una coppia della forma:
   (a) $\langle i, j-1\rangle$; o
   (b) $\langle i-1, \cdot\rangle$.

Ogni passo di questo procedimento porta ad una coppia lessicograficamente minore della precedente. Ciò dimostra che il procedimento termina dopo un numero finito di applicazioni dell'*Switching Lemma*. □

Il precedente teorema dimostra quindi che la scelta della regola di selezione è ininfluente per quanto riguarda le derivazioni di successo. Tuttavia, relativamente alle derivazioni di fallimento vi è un importante risvolto computazionale. Infatti il fallimento viene scoperto quando la regola seleziona un atomo che non unifica con alcuna testa di clausola del programma. È chiaro quindi che se tale atomo esiste, una regola che lo selezionasse subito porterebbe ad una derivazione di fallimento più corta.

Esempio 5.5. Sia dato il seguente goal:

$$\leftarrow q(1), \ldots, q(999), p(a).$$

ed il semplice programma

$$q(X) \leftarrow$$

È quindi chiaro che solo per l'ultimo atomo, $p(a)$, del goal non c'è alcuna clausola applicabile. Adottando la regola di selezione dell'atomo *leftmost* accade che prima di selezionare $p(a)$ devono essere processati tutti gli atomi alla sua sinistra. Solo a questo punto si scopre il fallimento. Se invece si adottasse una regola *rightmost* il fallimento avverrebbe immediatamente.

Il precedente esempio suggerisce una *regola di programmazione*: essendo Prolog implementato con la regola di scelta *leftmost* conviene scrivere più a sinistra gli atomi che sono più facilmente falsificabili.

3. SLD-alberi e regole di selezione

Assumiamo che la regola di selezione R sia fissata. Ad ogni passo di SLD-derivazione, una volta selezionato tramite R l'atomo del goal attuale, potrebbero esistere più clausole del programma applicabili a tale atomo. Quindi ad ogni passo una SLD-derivazione potrebbe potenzialmente essere estesa in differenti modi. Tutti i possibili modi di generare una SLD-derivazione per un dato goal vengono catturati dal concetto di SLD-albero.

Definizione 5.7. Un *SLD-albero* per $P \cup \{G\}$ via R è un albero tale che:

(1) i cammini da radice a foglia sono SLD-derivazioni di $P \cup \{G\}$ via R;
(2) ogni nodo G' con atomo selezionato A ha esattamente un figlio per ogni clausola C di P applicabile ad A. Inoltre il nodo figlio è l'SLD-risolvente di G e C rispetto ad A.

Le (eventuali) foglie dell'SLD-albero sono goal vuoti oppure goal di fallimento (ovvero goal in cui, per l'atomo selezionato, non esiste alcuna clausola applicabile).

Definizione 5.8. Un SLD-albero è:

- *di successo* se contiene almeno una SLD-derivazione di successo;
- *di fallimento finito* se è finito e non contiene SLD-derivazioni di successo.

Si noti che esistono SLD-alberi che non sono né di successo né di fallimento finito. Ad esempio si consideri P costituito dalle sole clausole $\{p(a) \leftarrow p(b),\ p(b) \leftarrow p(a)\}$ ed il goal $\leftarrow p(X)$.

Si noti che diverse regole di selezione possono generare differenti SLD-alberi per lo stesso goal. Un esempio di ciò:

Esempio 5.6. Consideriamo il seguente programma Prolog:

```
C1 :  antenato(X,Y) :- padre(X,Y).
C2 :  antenato(X,Y) :- antenato(X,Z),padre(Z,Y).
C3 :  padre(antonio,bruno).
```

Figura 5.1 mostra un SLD-albero di con regola di selezione leftmost, mentre un SLD-albero con regola di selezione rightmost è illustrato in Figura 5.2. Si può notare che i due SLD-alberi sono entrambi di successo, con risposta calcolata $[X/antonio, Y/bruno]$. Tuttavia, mentre il primo è infinito il secondo è finito.

Nota 5.1. Dato che la regola di selezione adottata in Prolog è *leftmost*, l'SLD-albero visitato dall'interprete è quello di Figura 5.1. Quindi sottoponendo il goal $\leftarrow antenato(X, Y)$ si otterrebbe subito la risposta $[X/antonio, Y/bruno]$. Tuttavia una eventuale richiesta di ulteriori soluzioni (digitando ";") porterebbe l'interprete a costruire un sottoalbero infinito

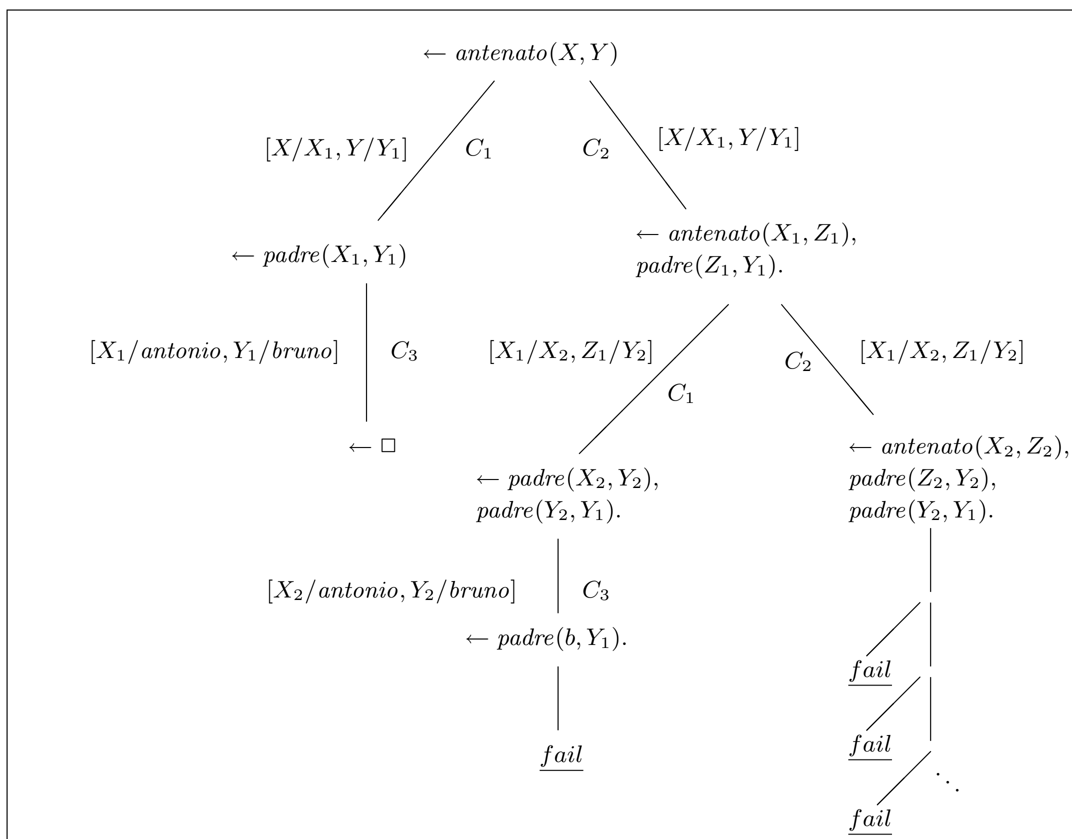


Figura 5.1. SLD-albero con regola di selezione leftmost

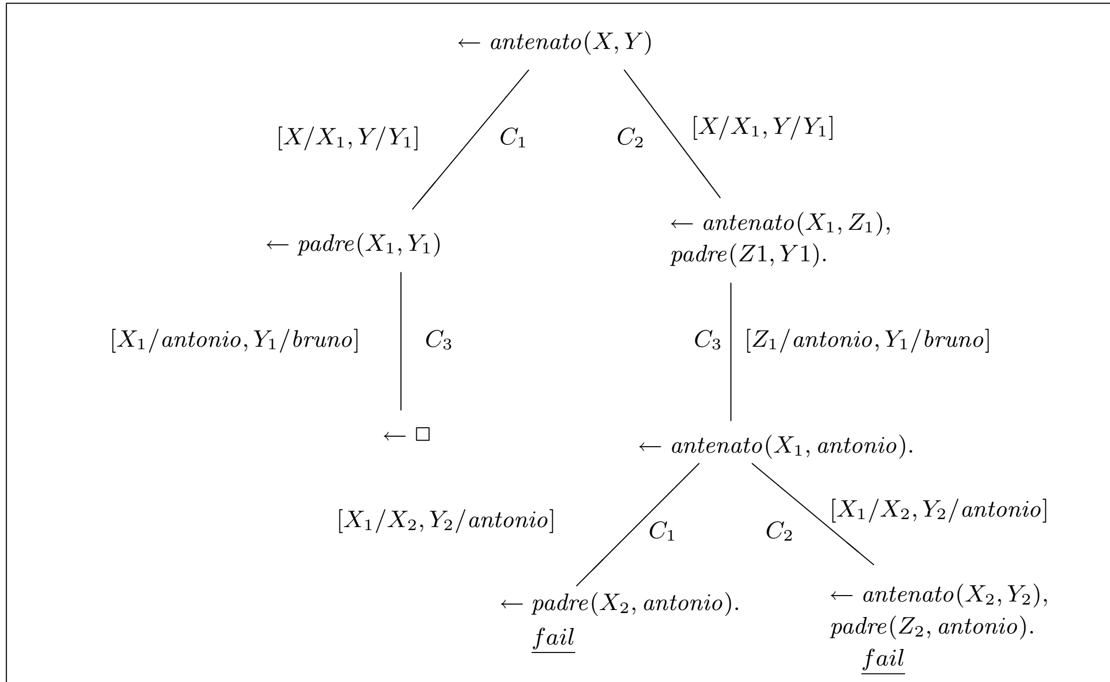


FIGURA 5.2. SLD-albero con regola di selezione rightmost

ma privo di soluzioni. Ciò dipende dall'ordine degli atomi nel corpo della clausola C_2 e dal fatto che questa clausola definisce ricorsivamente il predicato *antenato*.
L'esempio precedente suggerisce dunque un'altra regola pragmatica di programmazione:

> *"Specificando un predicato tramite una clausola ricorsiva, è buona norma che il predicato dichiarato nella testa non occorra quale predicato più a sinistra nel corpo."*

Si noti che rispetto ad una lettura completamente dichiarativa delle clausole e dei programmi definiti, questa regola di programmazione non ha rilevanza. È infatti originata solamente da motivazioni pragmatiche, causate delle particolari scelte algoritmiche effettuate nella implementazione dell'interprete Prolog.

Abbiamo visto quindi che gli SLD-alberi possono essere diversi per regole di selezione diverse. Tuttavia le SLD-derivazioni di successo presenti negli SLD-alberi sono sempre le stesse. Questa proprietà generale è sancita dal seguente teorema.

TEOREMA 5.3. *Fissata una regola di selezione R, se un SLD-albero per $P \cup \{G\}$ è di successo allora tutti gli SLD-alberi (ottenuti con diverse regole di selezione) saranno di successo.*

DIM. Traccia: ogni SLD-derivazione è presente nell'SLD-albero; si conclude osservando che se esiste una SLD-derivazione di successo, questa è presente indipendentemente dalla regola di selezione adottata. □

4. Search rule e costruzione dell'SLD-albero mediante backtracking

Abbiamo appena stabilito che la presenza di derivazioni di successo in un SLD-albero è indipendente dalla regola di selezione degli atomi. Tuttavia alcune regole di selezione portano a costruire SLD-alberi finiti mentre per altre l'SLD-albero può risultare infinito. Fissata una regola di selezione, vedremo in questa sezione come l'interprete Prolog costruisca l'SLD-albero. Più correttamente potremmo dire che l'SLD-albero non viene costruito interamente, ma viene solamente visitato in profondità, (costruendo cioè solo una SLD-derivazione alla volta) alla ricerca delle SLD-derivazioni di successo.

In linea di principio sarebbe possibile effettuare una costruzione in ampiezza (breadth-first) dell'SLD-albero. In tal caso tutte le derivazioni di successo sarebbero sempre (prima o poi) identificate e un interprete sarebbe potenzialmente in grado di fornire sempre tutte le soluzioni. Tuttavia una tale implementazione potrebbe comportare, in generale, inaccettabili problemi di efficienza e di risorse di calcolo.

Una *search rule* è una strategia di ricerca che determina il modo in cui viene costruito/visitato un SLD-albero. In altre parole, una search rule elimina il non-determinismo originato dalla arbitrarietà nella scelta della clausola applicabile.

Se, come accade in Prolog solo un branch dell'SLD-albero è presente in un dato istante del processo di inferenza, allora fornire una search rule significa stabilire, per ogni atomo selezionato, in che ordine debbano essere prese in considerazione le clausole applicabili.

Come menzionato la *search rule* adottata in Prolog, congiuntamente ad una strategia di visita depth-first, è:

> *"Scandendo il programma dalla prima alla ultima clausola, scegliere la prima clausola applicabile. Nel caso si costruisca una derivazione di fallimento o in caso di richiesta da parte dell'utente (;) di ulteriori soluzioni, si attui il backtracking (ovvero si torni all'ultimo punto in cui è stata scelta una clausola e si scelga la prossima clausola applicabile)."*

A titolo di esempio costruiamo l'SLD-albero della computazione relativa al programma precedente (adottiamo, come Prolog, la regola di selezione *leftmost*). Figura 5.3 illustra la prima SLD-derivazione di successo.

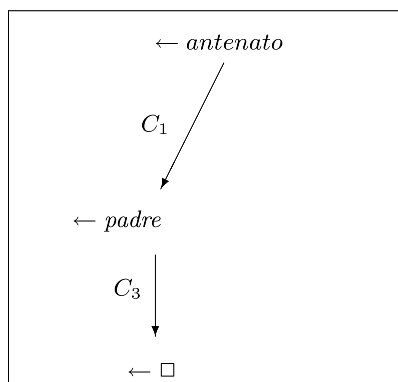


FIGURA 5.3. Costruzione di un SLD-albero con ottenimento di una prima soluzione.

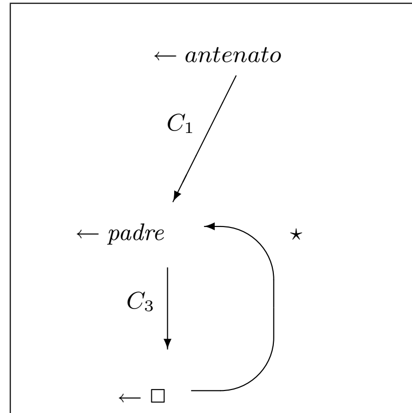


FIGURA 5.4. Backtracking al choice point precedente (vedi Figura 5.3)

Dopo aver fornito la prima soluzione, l'interprete Prolog attende una eventuale richiesta di ulteriori soluzioni. Qualora l'utente digiti ";" si effettua backtracking: cioè si individua il punto di scelta, *choice point*, più vicino al nodo foglia. Un choice point è un punto di branching dell'SLD-albero corrispondente ad un goal intermedio per il quale vi siano diverse clausole applicabili (fermo restando l'atomo selezionato). In Figura 5.4 il più vicino nodo candidato ad essere choice point è indicato dal carattere $\star$. Individuato il choice point (candidato) ci si chiede se vi sono altre clausole applicabili e non ancora utilizzate (ciò si verifica continuando a percorrere il programma nell'ordine in cui è scritto, considerando le clausole successive alla ultima clausola selezionata).

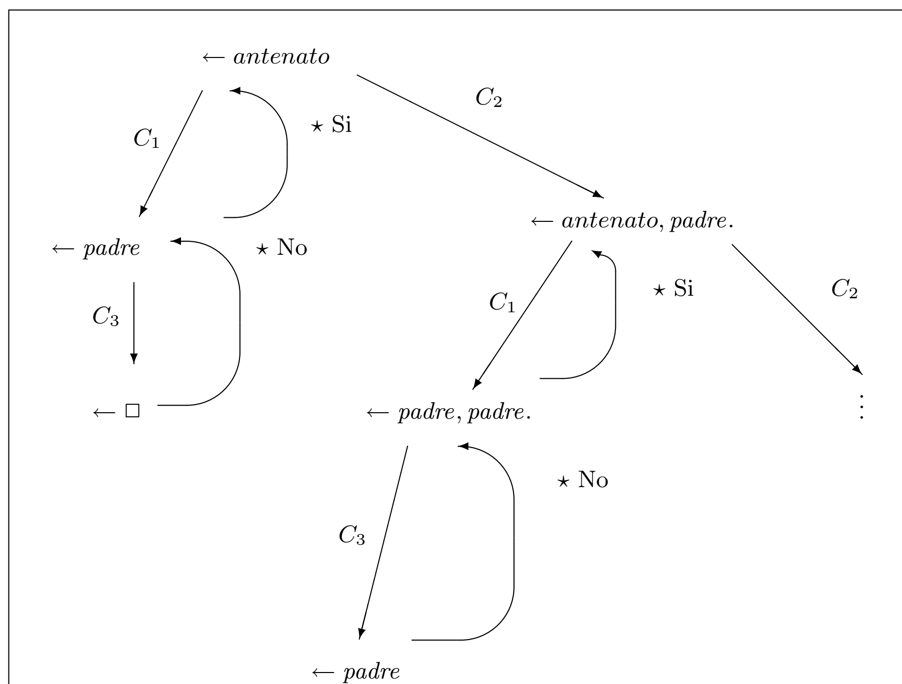


FIGURA 5.5. SLD-albero con backtracking (vedi Figura 5.4)

Qualora non vi siano più clausole applicabili non ancora utilizzate il backtracking viene innescato ricorsivamente: si cerca un ulteriore choice point precedente all'ultimo individuato.

Se invece esiste una clausola applicabile, si aprirà un nuovo ramo della computazione. Se l'SLD-albero in questione è finito il processo termina quando si esauriscono le clausole applicabili all'atomo selezionato nel goal iniziale (la radice dell'SLD-albero). Relativamente all'esempio considerato, l'SLD-albero è infinito (si veda Figura 5.5) e la sua costruzione completa comporterebbe una computazione infinita.

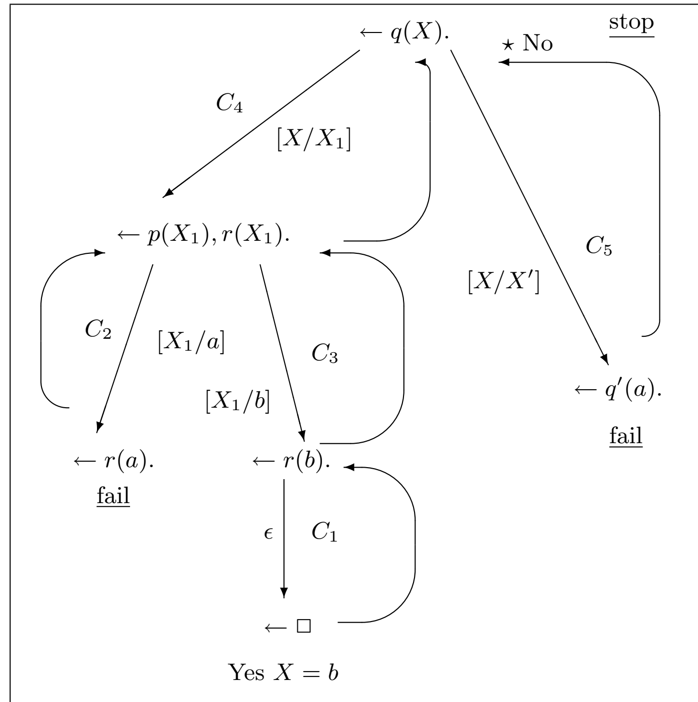


FIGURA 5.6. SLD-albero finito del programma dell'Esempio 5.7

ESEMPIO 5.7. Consideriamo il seguente programma Prolog P:

```
C1:  r(b).
C2:  p(a).
C3:  p(b).
C4:  q(X):-p(X),r(X).
C5:  q(X):-q'(a).
```

L'SLD-albero per il goal $\leftarrow q(X)$ è riportato in Figura 5.6.

ESEMPIO 5.8. Si consideri il programma P:

```
C1:  num(0).
C2:  num(s(X)):-num(X).
```

Sottoponendo il goal $\leftarrow num(X)$, man mano che si richiedono più soluzioni, l'interprete Prolog restituisce come risposte tutti i naturali (rappresentati come termini $s(s(\cdots s(0)\cdots)))$. Ad ogni richiesta viene costruito/visitato un nuovo ramo dell'SLD-albero di Figura 5.7.

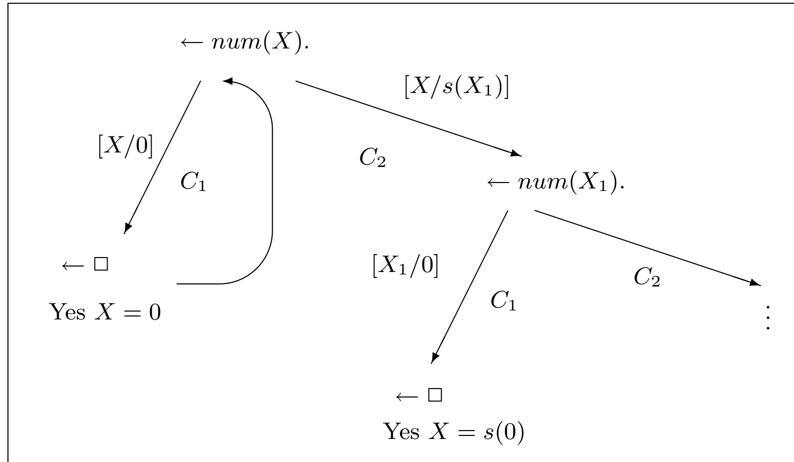


FIGURA 5.7. SLD-albero del programma dell'Esempio 5.8

CAPITOLO 6

Semantica dei programmi logici

Nel capitolo precedente abbiamo descritto la SLD-risoluzione ed abbiamo accennato ad alcune delle scelte implementative, dettate da necessità pragmatiche, adottate nella realizzazione di interpreti Prolog reali (leftmost piuttosto che rightmost, depth-first in luogo di breadth-first, ecc). Ciò che in pratica abbiamo fornito è una semantica operazionale della programmazione logica.

Ci si può chiedere a questo punto se esista un modo per descrivere il significato di un programma logico senza fare riferimento a particolari strategie o metodi di risoluzione.

In questo capitolo illustreremo tre diverse proposte di semantiche della programmazione logica: la semantica osservazionale (basata sulla SLD risoluzione), quella modellistica o logica e quella di punto fisso. Principale risultato del capitolo sarà dimostrare che le tre semantiche, pur mettendo l'accento su aspetti diversi della programmazione logica, sono in realtà coincidenti. Ognuna di esse ci fornirà una visione, un metodo di lettura, diverso per analizzare il significato di un programma definito.

1. Semantica osservazionale dei programmi definiti

L'aver definito la procedura risolutiva ci permette di dare un significato operazionale ad un programma definito. In particolare, dato un programma definito P, si potrebbe definire come semantica operazionale di P, per ogni possibile goal, l'intero SLD-albero associato al goal. Le semantiche osservazionali descritte in questa sezione si possono vedere come approssimazioni di tale semantica.

DEFINIZIONE 6.1. Sia P un programma definito. Si dice *osservabile* del programma P l'insieme di atomi ground così definito:

$$Oss(P) = \{A : A \text{ atomo ground ed esiste una SLD-derivazione di successo per } \leftarrow A\}$$

L'osservabile di un programma P fornisce la sua semantica osservazionale, e viene anche detto *insieme di successo.* La semantica osservazionale è basata sul comportamento operazionale e cerca di descriverlo. Chiaramente gli *atomi ground* dell'insieme di successo saranno ottenuti a partire dagli insiemi di simboli Π_P e $\mathcal{F}_P$ relativi a P. (Ricordiamo che qualora non vi fosse nessun simbolo di costante in $\mathcal{F}_P$, ne aggiungeremmo uno nuovo affinché la base di Herbrand di P non sia vuota.)

ESEMPIO 6.1. Dato il programma P

C_1: `p(a):-p(a).`
C_2: `p(a).`

la sua semantica osservazionale è:

$$Oss(P) = \{p(a)\}.$$

Si noti che l'interprete Prolog entra in ciclo quando dato il programma precedente gli si sottomette il goal $\leftarrow p(a)$. Ciò deriva dalle particolari scelte implementative effettuate relativamente a regola di selezione e strategia di ricerca, nella realizzazione della SLD-risoluzione in Prolog. Tuttavia esiste un SLD-albero contenente una SLD-derivazione di successo per il goal $\leftarrow p(a)$. Pertanto l'implementazione usuale dell'interprete Prolog risulta in un certo senso "incompleta" rispetto alla semantica osservazionale Oss.

ESERCIZIO 6.1. Si descriva $Oss(P)$ per i programmi seguenti:

(1)
```
num(0).
num(s(X)) :- num(X).
```

(2)
```
p(a).
q(X) :- p(X).
p(X) :- q(X).
```

La semantica operazionale Oss può risultare "poco precisa" qualora utilizzata per classificare programmi equivalenti. Il nostro intento è quello di dichiarare due programmi equivalenti se hanno la stessa semantica osservazionale (ovvero lo stesso insieme di successo).

Consideriamo ad esempio tre programmi costituiti da soli fatti:

P_1: `{p(a).   q(a).}`
P_2: `{p(a).   q(a).   q(X).}`
P_3: `{p(a).   q(X).}`

Questi tre i programmi possiedono lo stesso universo di Herbrand, esso è costituito dall'insieme di una sola costante: $\mathcal{H} = \{a\}$. Gli atomi ground costruibili su tale universo sono quindi solamente $p(a)$ e $q(a)$. Si verifica facilmente che per tutti tre i programmi l'osservabile Oss è costituito dall'insieme $\{p(a), q(a)\}$. Questo significa che per la semantica appena specificata i tre programmi sono equivalenti (hanno lo stesso insieme di successo). Tuttavia se facciamo entrare in gioco l'interprete Prolog, relativamente al goal $\leftarrow q(X)$, esso manifesta comportamenti sensibilmente diversi per i tre programmi.

ESERCIZIO 6.2. Si disegnino, per i tre programmi sopra elencati, l'SLD-albero e si analizzi il comportamento dell'interprete Prolog rispetto al goal $\leftarrow q(X)$, anche nel caso si richiedano più di una soluzione.

Consideriamo ora una diversa semantica osservazionale, definita come segue:

DEFINIZIONE 6.2.

$$O_2(P) = \{A : A \text{ è atomo e esiste una SLD-derivazione di successo per } \leftarrow A, \text{ con risposta calcolata } \varepsilon \text{ (o una variante rispetto a } vars(A))\}$$

Questa semantica è basata sull'*insieme di successo non ground* $O_2(P)$.

Riprendiamo ora i tre programmi P_1, P_2 e P_3 sopra riportati. Per individuare i rispettivi insiemi di successo non ground, dobbiamo considerare i seguenti quattro goal:

$$\leftarrow p(a) \qquad \leftarrow q(a) \qquad \leftarrow p(X) \qquad \leftarrow q(X)$$

Conseguentemente, gli osservabili dei tre programmi sono rispettivamente:

- $O_2(P_1) = \{p(a). \quad q(a).\}$

- $O_2(P_2) = \{p(a). \quad q(a). \quad q(X).\}$
- $O_2(P_3) = \{p(a). \quad q(a). \quad q(X).\}$

Quindi rispetto a questa semantica osservazionale il primo ed il secondo programma non sono equivalenti. O_2 modella meglio il comportamento operazionale rispetto a quanto faccia Oss.

Introduciamo un'ulteriore semantica osservazionale:

DEFINIZIONE 6.3. La *semantica di risposta calcolata* di un programma P è così definita:

$$\begin{array}{rcll} O_3(P) & = & \{A: & A \text{ è atomo e esiste } p \in \Pi,\ ar(p) = n \text{ ed} \\ & & & \text{esistono } X_1, \ldots, X_n \in \mathcal{V} \text{ variabili distinte tali che} \\ & & & \text{esiste una SLD-derivazione di successo per } \leftarrow p(X_1, \ldots, X_n) \\ & & & \text{con c.a.s. } \theta \text{ e } A = p(X_1, \ldots, X_n)\theta\} \end{array}$$

Analizziamo di nuovo P_1, P_2 e P_3 per mezzo di questa nuova semantica. Questa volta i goal da considerare sono solamente

$$\leftarrow p(X) \quad \text{e} \quad \leftarrow q(X)$$

Otteniamo che

- $O_3(P_1) = \{p(a), q(a)\}$,
- $O_3(P_2) = \{p(a), q(a), q(X)\}$, e
- $O_3(P_3) = \{p(a), q(X)\}$.

I tre osservabili sono diversi tra loro. Ciò significa che questa semantica è in grado di distinguere i tre programmi ed è dunque più precisa delle precedenti.

Sembra quindi che la semantica osservazionale "migliore" sia l'ultima. Tuttavia, per ragioni che diverranno chiare nelle prossime sezioni, usualmente la semantica osservazionale di un programma definito viene descritta tramite l'insieme Oss.

2. Semantica logica (modellistica) di programmi definiti

Abbiamo già introdotto pre-interpretazioni e interpretazioni di Herbrand nel Capitolo 2. Vediamo come sia possibile dare una semantica basata sui modelli ai programmi definiti. Ricordiamo che una pre-interpretazione di Herbrand fissa l'interpretazione dei termini in un universo composto, in parole povere, dai termini stessi. Una interpretazione di Herbrand è una struttura che estende una pre-interpretazione di Herbrand con l'interpretazione dei simboli predicativi. Come convenzione, nel nostro caso, l'insieme Π_P dei simboli predicativi sarà costituito dai simboli di predicato presenti nel programma P. ricordiamo anche che ogni particolare interpretazione di Herbrand I si può semplicemente descrivere come un insieme di atomi ground (quelli che la interpretazione rende veri). Dato un programma, tra tutte le possibili interpretazioni ci sono ovviamente quelle che soddisfano tutte le clausole del programma, ovvero i modelli del programma.

ESEMPIO 6.2. Dato il programma P

```
p(a).
q(b).
p(f(X)) :- r(X).
```

abbiamo $\mathcal{F}_P = \{a_{/0}, b_{/0}, f_{/1}\}$ e $\Pi_P = \{p, q, r\}$. L'universo di Herbrand $T(\mathcal{F})$ e la base di Herbrand $\mathcal{B}_P$ sono rispettivamente:

$$\begin{array}{rcl} T(\mathcal{F}) & = & \{a, b, f(a), f(b), f(f(a)), f(f(b)), \dots\} \\ \mathcal{B}_P & = & \{p(a), p(b), p(f(a)), p(f(b)), p(f(f(a))), p(f(f(b))), \dots \\ & & q(a), q(b), q(f(a)), q(f(b)), q(f(f(a))), q(f(f(b))), \dots \\ & & r(a), r(b), r(f(a)), r(f(b)), r(f(f(a))), r(f(f(b))), \dots\} \end{array}$$

I modelli di Herbrand per P vanno come sempre cercati tra i sottoinsiemi di $\mathcal{B}_P$. Un particolare modello, ad esempio, è $M = \{p(a), q(b)\}$. Una interpretazione che invece non è modello per P è $I = \{p(f(a)), r(b)\}$.

2.1. Teorie di clausole. L'esistenza di un modello per un insieme di enunciati T non implica in generale l'esistenza di un modello di Herbrand (si veda l'Esempio 2.11). Se però la teoria T (l'insieme di enunciati) è di tipo particolare, ovvero è un insieme di clausole, allora vale il seguente risultato:

TEOREMA 6.1. *Se T è un insieme di clausole soddisfacibile, allora ha modelli di Herbrand.*

DIM. Sia $\mathcal{D} = \langle D, (\cdot)^{\mathcal{D}} \rangle$ un modello di T. Costruiamo un pre-interpretazione con dominio $T(\mathcal{F})$ con $\mathcal{F}$ insieme di simboli di funzioni e di costanti usati in T (più eventualmente un simbolo di costante "ausiliario"). Estendiamo tale pre-interpretazione (di Herbrand) per ottenere una interpretazione di Herbrand scegliendo il seguente insieme di atomi soddisfatti:

$$M = \{p(t_1, \dots, t_n) : p \in \Pi, t_1, \dots, t_n \in T(\mathcal{F}), \mathcal{D} \models p(t_1, \dots, t_n)\}.$$

Per come è definito, M è un modello di Herbrand per T. □

Ecco alcune importanti conseguenze di questo risultato.

COROLLARIO 6.1. Sia T un insieme di clausole. Allora T è insoddisfacibile se e solo se non ammette modelli di Herbrand.

DIM. $(\rightarrow)$: Se T ammette modelli di Herbrand allora ammette modelli, quindi è soddisfacibile.

$(\leftarrow)$: Se T è soddisfacibile allora ammette modelli, quindi per il Teorema 6.1 ammette modelli di Herbrand. □

COROLLARIO 6.2. Sia T un insieme di clausole e A un atomo di B_T. Allora $T \models A$ se e solo se A è vero in tutti i modelli di Herbrand di T.

DIM. $(\rightarrow)$: Per definizione di conseguenza logica, se $T \models A$, allora A è vero in tutti i modelli di T. In particolare lo è nei modelli di Herbrand.

$(\leftarrow)$: Sia A vero in tutti i modelli di Herbrand. Allora $T \cup \{\neg A\}$ non ha modelli di Herbrand. Quindi $T \cup \{\neg A\}$ è insoddisfacibile. Conseguentemente, per il Lemma 2.1, otteniamo $T \models A$. □

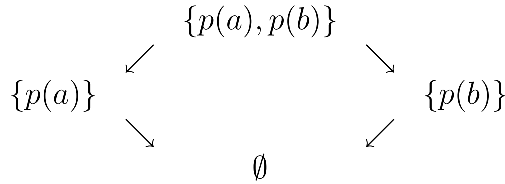
COROLLARIO 6.3. Sia T un insieme di clausole e A un atomo di B_T. Allora $T \models A$ se e solo se $T \cup \{\neg A\}$ non ha modelli di Herbrand.

NOTA 6.1. Facciamo una breve digressione che potrà essere compresa con maggior facilità considerando i contenuti della Sezione 2.3.

Si noti che se A è un atomo ground, la costruzione di una derivazione per $P \cup \{\leftarrow A\}$ corrisponde a dimostrare che $P \models A$ ovvero che in tutti i modelli di P (in particolare in quelli di Herbrand) vale A. Consideriamo però che la scrittura $\leftarrow A$ (che possiamo leggere come implicazione $\mathsf{false} \leftarrow A$) è equivalente alla disgiunzione $\mathsf{false} \vee \neg A$, ovvero al letterale negativo $\neg A$. Il processo di derivazione del goal vuoto $\leftarrow \Box$ (che equivale alla formula false) a partire da $P \cup \{\neg A\}$ corrisponde in realtà a dimostrare che false è derivabile dagli enunciati in $P \cup \{\neg A\}$. Ma dimostrare che da $P \cup \{\neg A\}$ si può derivare il falso significa provare che l'insieme $P \cup \{\neg A\}$ è insoddisfacibile, dato che false non ha modelli. In definitiva trovare una derivazione di successo per $P \cup \{\leftarrow A\}$ significa dimostrare che non esistono modelli di Herbrand per $P \cup \{\neg A\}$.

Un esempio di teoria composta da clausole non di Horn, permette di introdurre dei concetti elementari sul *reticolo* delle interpretazioni di Herbrand:

ESEMPIO 6.3. Dati $T = \{p(a) \vee p(b)\}$ e $B_P = \{p(a), p(b)\}$ possiamo considerare quattro diverse interpretazioni di Herbrand, che possono essere visualizzate come segue enfatizzando le reciproche inclusioni:



Si osservi che:

(1) L'insieme delle interpretazioni di Herbrand mostrate in figura forma un reticolo $\langle \mathcal{P}(B_P), \subseteq \rangle$.
(2) Non esiste un modello minimo, in quanto l'infimum del reticolo, ovvero $\emptyset$, non è un modello di T.
(3) l'intersezione di due modelli non è necessariamente un modello. Infatti, sia $\{p(a)\}$ che $\{p(b)\}$ sono modelli di T, mentre la loro intersezione non lo è.

Il punto cruciale è che la clausola in T non è una clausola di Horn.

Un altro esempio relativo a clausole non di Horn:

ESEMPIO 6.4. Si consideri T definita come:

$$\begin{array}{ccc} (p(a) \vee p(b)) & \wedge & (\neg p(a) \vee p(b)) \quad \wedge \\ (p(a) \vee \neg p(b)) & \wedge & (\neg p(a) \vee \neg p(b)) \end{array}$$

È una teoria costituita da clausole. Il reticolo delle interpretazioni di Herbrand è lo stesso visto per l'Esempio 6.3. Tuttavia questo insieme di clausole è insoddisfacibile.

2.2. Clausole Definite. I due Esempi 6.3 e 6.4 riguardano teorie costituite da clausole non di Horn (e quindi clausole non definite). In particolare abbiamo visto che la teoria dell'Esempio 6.4 non ha modelli. La eventualità che non esistano modelli non sussiste invece nel caso di programmi definiti. Il prossimo teorema stabilisce che per ogni dato programma definito P, esiste sempre almeno un modello di Herbrand. In altre parole, un programma definito è sempre soddisfacibile.

TEOREMA 6.2. *Sia dato un programma di clausole definite P. La base di Herbrand B_P è un modello per P.*

DIM. Ogni clausola in P è una implicazione della forma:

$$\vec{\forall}(H \leftarrow B_1 \wedge \cdots \wedge B_n).$$

Dato che ogni possibile istanza ground di H appartiene a B_P, si ha $B_P \models \vec{\forall} H$. Quindi B_P è modello per la clausola. □

Quindi ogni programma definito ha sempre almeno un modello. I prossimi teoremi ci permetteranno di stabilire che tra i vari modelli di Herbrand possibili, ce ne è uno particolarmente significativo. Iniziamo con una importante proprietà di chiusura della classe dei modelli di Herbrand.

TEOREMA 6.3. *Sia P un programma di clausole definite. Siano $\mathcal{D}_1, \mathcal{D}_2, \ldots$ modelli di Herbrand di P. Allora l'intersezione $\bigcap_{i \geq 1} \mathcal{D}_i$ è un modello di Herbrand di P.*

DIM. Consideriamo una qualsiasi istanza ground di una clausola di P

$$C: \quad H \leftarrow B_1 \wedge \ldots \wedge B_n.$$

Dimostriamo che $\bigcap_{i \geq 1} \mathcal{D}_i$ è modello di C. Se H appartiene a $\mathcal{D}_i$ per ogni $i > 0$ allora H appartiene a $\bigcap_{i \geq 1} \mathcal{D}_i$ e possiamo concludere. Se invece esiste un modello $\mathcal{D}_\ell$ di C tale che H non appartiene a $\mathcal{D}_\ell$, allora almeno uno tra i B_j non appartiene a $\mathcal{D}_\ell$. Conseguentemente tale B_j non appartiene a $\bigcap_{i \geq 1} \mathcal{D}_i$ che quindi è modello di C. □

Una notevole conseguenza:

COROLLARIO 6.4. Sia P un programma di clausole definite. Siano inoltre $\mathcal{D}_1, \mathcal{D}_2, \ldots$ tutti i modelli di Herbrand di P. Allora l'interpretazione

$$M_P = \bigcap_{i \geq 1} \mathcal{D}_i$$

è un modello di Herbrand. M_P è il *modello di Herbrand minimo*.

La semantica modellistica dei programmi definiti viene data come segue. Dato un programma definito P, l'insieme di atomi ground così definito

$$M_P = \{A \in \mathcal{B}_P : P \models A\}$$

(si noti che è proprio il modello minimo) è la *semantica logica* (o modellistica) del programma P.

I risultati precedenti stabiliscono che dato un programma definito esiste sempre un unico modello di Herbrand, per certi versi "migliore" degli altri: il modello minimo M_P. La semantica modellistica dei programmi definiti viene data quindi in termini dei modelli minimi. Il prossimo teorema giustifica questa scelta.

TEOREMA 6.4 (Correttezza e Completezza della SLD-derivazione). *Sia P un programma di clausole definite. Allora $Oss(P) = M_P$.*

DIM. Omessa. □

2.3. Risoluzione. La procedura di SLD-risoluzione studiata nel Capitolo 5 è un caso particolare della procedura di risoluzione introdotta da J. A. Robinson [**Rob65, Rob68**]. Tale procedura risulta applicabile nel caso di teorie composte di clausole generiche (non necessariamente di Horn, o definite).

Nella sua forma più semplice, nel caso di clausole ground, la regola di risoluzione di Robinson opera in questo modo: date due clausole $C_1 : L_1 \vee \cdots \vee L_n$ e $C_2 : H_1 \vee \cdots \vee H_m$, se esistono due letterali L_i e H_j tali che $L_i \equiv \neg H_j$, allora la clausola

$$L_1 \cdots L_{i-1} \vee L_{i+1} \vee \cdots \vee L_n \vee H_1 \vee \cdots \vee H_{j-1} \vee H_{j+1} \vee \cdots \vee H_m$$

è una *risolvente* di C_1 e C_2.
Nella sua generalizzazione per clausole non ground, si opera similmente a quanto accade nel caso della SLD-risoluzione: fissati i due letterali L_i e H_j selezionati, si può effettuare il passo di risoluzione se uno di essi risulta unificabile con la negazione dell'altro. In tal caso la clausola ottenuta viene istanziata tramite il corrispondente m.g.u.. Per maggiori dettagli si suggerisce di riferirsi alla vasta letteratura sull'argomento, ad esempio [**CL73, Lov78**].

Illustriamo intuitivamente, tramite degli esempi, la relazione intercorrente tra SLD-risoluzione e risoluzione. Consideriamo per iniziare una teoria definita ground P:

$$\begin{array}{c} p(a) \leftarrow p(b) \\ p(b) \end{array}$$

Le sue clausole possono essere riscritte così:

$$\begin{array}{c} p(a) \vee \neg p(b) \\ p(b) \end{array}$$

La SLD-derivazione per $P \cup \{\leftarrow p(a), p(b)\}$ si può vedere come la seguente derivazione tramite risoluzione (di Robinson):

$$\begin{array}{cc} \neg p(a) \vee \neg p(b) & p(a) \vee \neg p(b) \\ \hline \neg p(b) \vee \neg p(b) & p(b) \\ \hline \neg p(b) & p(b) \\ \hline \multicolumn{2}{c}{\Box} \end{array}$$

relativa all'insieme di clausole $\{p(a) \vee \neg p(b),\ p(b),\ \neg p(a) \vee \neg p(b)\}$. La derivazione ha prodotto la clausola vuota $\Box$. Tale clausola equivale a false, quindi questa derivazione certifica che l'insieme delle tre clausole è insoddisfacibile (o equivalentemente che l'insieme P ha come conseguenza logica la congiunzione $p(a) \wedge p(b)$). Si noti che in questo caso la derivazione è *lineare*, ovvero ad ogni passo una delle due clausole coinvolte è sempre la clausola derivata al passo precedente. Si noti ovviamente alla analogia con la SLD-derivazione ottenibile per $P \cup \{\leftarrow p(a), p(b)\}$.

Un altro esempio in cui trattiamo una teoria insoddisfacibile (si veda l'Esempio 6.4):

$$\begin{array}{ccc} (p(a) \vee p(b)) & \wedge & (\neg p(a) \vee p(b)) \quad \wedge \\ (p(a) \vee \neg p(b)) & \wedge & (\neg p(a) \vee \neg p(b)) \end{array}$$

In questo caso la costruzione di una derivazione lineare che evolva mimando una SLD-derivazione non è possibile proprio a causa del fatto che P non è un programma definito:

$$\cfrac{\cfrac{\neg p(a) \vee \neg p(b) \qquad p(a) \vee \neg p(b)}{\neg p(b) \vee \neg p(b) \qquad ???}}{\vdots}$$

È facile verificare che il processo continua a generare clausole sempre composte da due letterali. Quindi non otterremo mai la clausola vuota $\Box$.

Per disporre di una procedura completa è necessario affiancare alla regola di Robinson una ulteriore regola detta *factoring.* Tale regola permette (nel caso ground) di fattorizzare i letterali ripetuti di una clausola (ovvero, eliminando le ripetizioni). Con le due regole possiamo ottenere la seguente derivazione di $\Box$:

$$\cfrac{\cfrac{\neg p(a) \vee \neg p(b) \qquad p(a) \vee \neg p(b)}{\neg p(b)} \qquad \cfrac{p(a) \vee p(b) \qquad \neg p(a) \vee p(b)}{p(b)}}{\Box}$$

Anche in tal modo tuttavia, non è possibile ottenere una derivazione lineare, e tantomeno "ripercorribile" tramite passi di SLD-risoluzione.

Abbiamo visto che SLD-risoluzione risulta essere una procedura completa per il frammento delle clausole di Horn. La procedura di Robinson (nella sua versione più generale) risulta invece completa per la logica del primo ordine.

3. Semantica di punto fisso di programmi definiti

In questa sezione forniremo un terzo modo di studiare la semantica dei programmi definiti. Faremo ricorso a dei concetti base della teorie dei reticoli che, per comodità del lettore, sono stati brevemente riassunti nella Appendice A.

3.1. Operatore sui programmi. Come usuale, assumiamo che nei programmi che tratteremo in questa sezione occorra almeno un simbolo di costante. In caso contrario, al fine di generare un universo di Herbrand non vuoto, ne aggiungiamo uno.

Alcune definizioni:

Definizione 6.4. Dato un programma definito P, Definiamo:

$$\begin{array}{rcll} ground(P) & = & \{(A \leftarrow B_1, \ldots, B_n)\theta : & C \equiv A \leftarrow B_1, \ldots, B_n \in P, \\ & & & \theta \text{ è sostituzione ground per tutte le variabili di } C\} \end{array}$$

Definizione 6.5. Dato un programma definito P, definiamo l'operatore di *conseguenza immediata*,

$$T_P : \wp(\mathcal{B}_P) \to \wp(\mathcal{B}_P)$$

che trasforma interpretazioni in interpretazioni, nel modo seguente:

$$T_P(I) = \{A : \text{la clausola } (A \leftarrow B_1, \ldots, B_n) \text{ è in } ground(P) \text{ e } B_1, \ldots, B_n \text{ sono in } I\}$$

Esempio 6.5. Sia P il programma definito:

```
r(b).
p(a).
q(X) :- r(X),p(X).
```

Allora:

$$\begin{array}{rcl} T_P(\{r(a), r(b), p(b)\}) & = & \{r(b), p(a), q(b)\} \\ T_P(\{r(b), p(a), q(b)\}) & = & \{r(b), p(a)\} \end{array}$$

Sussiste il seguente risultato:

TEOREMA 6.5. *Sia P un programma definito ed I una interpretazione di Herbrand. Allora I è modello di Herbrand di P se e solo se $T_P(I) \subseteq I$.*

DIM. ($\rightarrow$) Sia $A \in T_P(I)$; allora esiste un'istanza ground di una clausola $A \leftarrow B_1, \ldots, B_n$ in *ground*(P), con $B_1, \ldots, B_n$ in I. Se I è un modello, allora A è in I.

($\leftarrow$) Sia $A \leftarrow B_1, \ldots, B_n$ in *ground*(P). Assumendo che $B_1, \ldots, B_n$ siano in I, si ha che A deve essere in $T_P(I)$. Essendo $T_P(I) \subseteq I$ per ipotesi, allora $A \in I$ ed I è un modello. □

Come immediata conseguenza abbiamo:

COROLLARIO 6.5. Sia P un programma definito. Allora:

(1) $\mathcal{B}_P$ è modello di P;
(2) Se $T_P(I) = I$, allora I è modello di P.

DIM. Entrambi i punto sono immediati, infatti:

(1) $T_P(\mathcal{B}_P)$ è sempre sottoinsieme di $\mathcal{B}_P$; Si conclude per il Teorema 6.5.
(2) $T_P(I) = I$ implica che $T_P(I) \subseteq I$. Anche ora si conclude per il Teorema 6.5.

□

L'operatore di conseguenza immediata gode delle seguenti proprietà:

LEMMA 6.1. *Dato il programma definito P, allora*

(1) *T_P è un operatore monotono, ovvero*

$$\forall X, Y \in \wp(\mathcal{B}_P), X \subseteq Y \rightarrow T_P(X) \subseteq T_P(Y)$$

(2) *T_P è un operatore continuo, ovvero per ogni catena $I_0 \subseteq I_1 \subseteq I_2 \subseteq \cdots$ vale che*

$$T_P\left(\bigcup_{i \geq 0} I_i\right) = \bigcup_{i \geq 0} T_P(I_i)$$

DIM. Esercizio. □

La continuità dell'operatore T_P garantisce l'esistenza del suo minimo punto fisso, inoltre assicura che questo punto fisso possa essere calcolato effettivamente (vedi Appendice A). Abbiamo anche visto che il Corollario 6.5 asserisce che ogni punto fisso di T_P è un modello per P.

In base a queste proprietà del operatore T_P possiamo descrivere elegantemente la semantica di ogni programma P. Inoltre ciò che si dimostra (vedi il Corollario 6.6) è che questa semantica coincide con le semantiche osservazionale e logica presentate nelle pagine precedenti.

TEOREMA 6.6. *Sia P un programma definito. Allora $M_P = T_P \uparrow \omega$.*

DIM. ($\subseteq$) $T_P \uparrow \omega$ è punto fisso (Tarski). Allora $T_P \uparrow \omega = T_P(T_P \uparrow \omega)$, cioè è modello di P. Allora $M_P \subseteq T_P \uparrow \omega$, perchè è il modello minimo.

($\supseteq$) Poiché M_P modello, allora $T_P(M_P) \subseteq M_P$.

Se fosse $T_P(M_P) = M_P$ allora banalmente $T_P \uparrow \omega \subseteq M_P$ poiché è il minimo punto fisso.

Se invece fosse $T_P(M_P) \subset M_P$, allora per monotonia si avrebbe $T_P(T_P(M_P)) \subseteq T_P(M_P)$. Ma allora $T_P(M_P)$ sarebbe modello di P e incluso strettamente in M_P: assurdo poiché M_P è il modello minimo. □

COROLLARIO 6.6. Sia P un programma definito. Allora $M_P = T_P \uparrow \omega = Oss(P)$.

Chiudiamo questo capitolo dando due spunti di studio. Il primo è relativo ad un alternativo approccio alla semantica, applicabile praticamente ad una ristretta classe di programmi definiti, i programmi DATALOG. Infine considereremo l'operatore duale di T_P e la relazione che sussiste tra i suoi punti fissi e i modelli di P.

3.2. Semantica bottom-up di DATALOG. DATALOG è un linguaggio per l'interrogazione di basi di dati dichiarative la cui sintassi e semantica è esattamente quella dei programmi di clausole definite. Vi è però l'importante restrizione che non sono ammessi simboli di funzione con arità positiva. In altre parole nel programma (solitamente in questo contesto si parla di *base di clausole* o *base di conoscenza*) vi sono solo simboli di costante, variabili e simboli di predicato.

Questa restrizione garantisce che ogni dato programma P l'insieme $T(\mathcal{F}_P)$ è un insieme finito, così come $\mathcal{B}_P$.

Per tali programmi è pensabile fornire una procedura operazionale basata sull'operatore T_P introdotto descrivendo la semantica di punto fisso. Dato un goal ground

$$\leftarrow p(t_1, \ldots, t_n)$$

si costruisce infatti l'insieme $T_P \uparrow \omega$. Tale insieme è sempre costruibile in tempo finito perchè è un sottoinsieme di $\mathcal{B}_P$, che è finito. La costruzione parte dall'insieme vuoto e procede aggiungendo ad ogni passo i nuovi atomi determinati tramite T_P. Il processo termina quando si raggiunge un punto fisso. A questo punto si verifica se l'atomo $p(t_1, \ldots, t_n)$ appartiene o meno all'insieme costruito. In caso affermativo allora $p(t_1, \ldots, t_n)$ è conseguenza del programma P.

Tale procedura *bottom-up* appare certamente più inefficiente di quella *top-down* implementata dalla SLD-risoluzione. Tuttavia si può pensare, ad esempio, essere effettuata off-line e rieseguirla solo in seguito ad aggionamenti o modifiche della base di clausole. In tal modo per rispondere ad un goal è sufficiente verificare l'esistenza di un atomo in un insieme di atomi.

Questa tecnica di ragionamento è un esempio di ragionamento in *forward chaining.* Tale termine enfatizza il fatto che, a partire dalle conoscenze certe (la base estensionale), si proceda utilizzando le regole (base intensionale) "in avanti". Il processo continua fino a che sia possibile inferire nuove conseguenze, o in generale, fino a quando non si ottenga una dimostrazione del goal.

Un semplice esempio.

ESEMPIO 6.6. Supponiamo data la seguente base di clausole P:

```
a :- b,c.
b :- d,e.
b :- g,e.
c :- e.
d.
e.
f :-a,g.
```

Vogliamo sapere se il fatto `g` sia o meno conseguenza della conoscenza descritta da questa base di clausole. Procedendo bottom-up, utilizzando l'operatore T_P, otteniamo la seguente sequenza di insiemi di fatti:

$$\{\}$$
$$\{d, e\}$$
$$\{d, e, b, c\}$$
$$\{d, e, b, c, a\}$$

Il modello minimo è quindi $\{d, e, b, c, a\}$. Possiamo concludere che `g` non è derivabile da P.

Si noti una proprietà importante di questa tecnica di ragionamento. In virtù della finitezza della base di Herbrand, questa procedura è sempre terminante. Contrariamente a ciò, è semplice costruire dei programmi (anche privi di simboli di funzione) per i quali la procedura di SLD-derivazione non sia in grado di fornire una risposta e origini computazioni infinite. Ad esempio

```
padre(X,Y) :- figlio(Y,X).
figlio(X,Y) :- padre(Y,X).
padre(pinco,pallino).
```

Un altro esempio per il quale SLD-risoluzione, nella sua versione implementata nel Prolog, non è in grado di dare una risposta al goal `?-p(a,c)`:

```
p(a,b).
p(c,b).
p(X,Z) :- p(X,Y),p(Y,Z).
p(X,Y) :- p(Y,X).
```

Si noti che `p(a,c)` è conseguenza logica del programma. Il modello minimo ottenibile procedendo bottom-up è infatti:

$$\{p(a,b), p(c,b), p(b,a), p(b,c), p(a,a), p(a,c), p(c,a), p(c,c), p(b,b)\}.$$

Più in generale, è facile verificare che qualsiasi implementazione della SLD-risoluzione che utilizzi una search rule puramente depth-first (indipendentemente dalla regola di selezione dell'atomo), qualora la scelta delle clausole sia determinata dall'ordine di queste, non è in grado di rispondere al goal.

3.3. Operatori *all'ingiù* e asimmetrie. Dato un operatore T, analogamente a quanto fatto con $T \uparrow \omega$, si può definire anche la sua iterazione *all'ingiù* (si veda Appendice A).

Nel contesto dei programmi logici, l'operatore iterato all'ingiù è l'operatore duale dell'-operatore $T \uparrow \omega$ visto nelle pagine precedenti.

Si consideri il seguente programma:

```
p(0) :- q(X)
q(s(X)) :- q(X)
```

Calcoliamo:

$$\begin{array}{rcl} T_P \downarrow 0(\mathcal{B}_P) & = & \{p(0), p(s(0)), p(s(s(0))), p(s(s(s(0))), \ldots \\ & & q(0), q(s(0)), q(s(s(0)), q(s(s(s(0)))), \ldots\} \\ T_P \downarrow 1(\mathcal{B}_P) & = & \{p(0), q(s(0)), q(s(s(0)), q(s(s(s(0)))), \ldots\} \\ T_P \downarrow 2(\mathcal{B}_P) & = & \{p(0), q(s(s(0)), q(s(s(s(0)))), \ldots\} \\ T_P \downarrow 3(\mathcal{B}_P) & = & \{p(0), q(s(s(s(0)))), \ldots\} \\ \vdots & \vdots & \vdots \\ T_P \downarrow \omega(\mathcal{B}_P) & = & \{p(0)\} \end{array}$$

Tuttavia quest'ultimo non è un punto fisso. Infatti riapplicando T_P otteniamo $T_P(\{p(0)\}) = \emptyset = T_P(\emptyset)$. Quindi $\emptyset$ è il punto fisso.

Osserviamo quindi una importante differenza tra l'iterazione all'ingiù e l'iterazione all'insù di un operatore continuo. Il fatto che l'operatore sia continuo e che sia definito su reticolo completo (che in particolare ha massimo) non assicura che iterando all'ingiù si raggiunga un punto fisso in un numero finito di passi. In altre parole non è garantito che $T_P \downarrow \omega(\mathcal{B}_P)$ sia punto fisso. È invece vero che iterando all'insù si raggiunga un punto fisso in un numero finito di passi.

Tuttavia esiste un risultato che assicura l'esistenza di un ordinale α (non necessariamente finito) tale che $T_P \downarrow \alpha(\mathcal{B}_P)$ sia il massimo punto fisso.

Da ciò deduciamo quindi che $T_P \uparrow \omega(\mathcal{B}_P)$ non coincide necessariamente con $T_P \downarrow \omega(\mathcal{B}_P)$. Il primo infatti è garantito essere il minimo punto fisso, mentre il secondo potrebbe anche non essere un punto fisso (nel caso lo sia, è il massimo punto fisso). Ci chiediamo ora se sia possibile che massimo e minimo punto fisso siano distinti.

La risposta è affermativa. Si consideri il seguente esempio:

```
q(a).
q(s(X)) :- q(s(X))
```

È facile verificare che il minimo punto fisso è $M_P = T_P \uparrow \omega = \{q(a)\}$ mentre il massimo punto fisso è $\mathcal{B}_P = T_P \downarrow \omega(\mathcal{B}_P)$.

Esercizio 6.3. Si verifichi che inclusi strettamente tra minimo e massimo punto fisso dell'esempio sopra ci sono una quantità più che numerabile di punti fissi.

Esercizio 6.4. Abbiamo già verificato che se una funzione è Turing calcolabile, allora esiste un programma definito in grado di calcolarla. Si dimostri ora il teorema inverso:

Teorema 6.7. *Sia $f : \mathbb{N}^n \longrightarrow \mathbb{N}$ una funzione calcolata da un programma definito P con SLD-risoluzione. Allora f è Turing calcolabile.*

Dim. Suggerimento: $p_f(X_1, \ldots, X_n, Y) \in Oss(P)$ se e solo se $Y = f(X_1, \ldots, X_n)$. Verificare se vale che $p_f(X_1, \ldots, X_n, Y) \in Oss(P)$ può essere fatto con una visita in ampiezza dell'SLD-albero. L'insieme $Oss(P)$ è ricorsivamente enumerabile. $\Box$

4. Esercizi

Esercizio 6.5. Sia dato il seguente insieme di clausole. Indicare il modello minimo, descrivendo il procedimento utilizzato per determinarlo. Indicare il modello massimo (quello che include tutti gli altri), giustificando la risposta. Indicare un terzo modello diverso sia da quello minimo che da quello massimo.

$$\begin{array}{rcl} p(a) & \leftarrow & p(X), q(X). \\ p(f(X)) & \leftarrow & p(X). \\ q(b). & & \\ q(f(X)) & \leftarrow & q(X). \end{array}$$

Esercizio 6.6. Completare il seguente programma fatto di due clausole
$\{p(f(X)) \ \leftarrow \ p(X). \quad q(a).\}$
aggiungendo delle clausole in modo che:

- Il modello minimo contenga sia l'atomo $q(f(f(b)))$ che l'atomo $q(b)$. Nel caso un tale programma non esista, giustificarne il motivo.
- L'atomo $p(f(f(a)))$ non appartenga al modello minimo e tutti i modelli del programma siano infiniti. Nel caso un tale programma non esista, giustificarne il motivo.

(A vostra scelta, fornire due risposte distinte o anche, nel caso sia possibile, una sola risposta che soddisfi entrambe le condizioni.)

Esercizio 6.7. Descrivere un metodo per ottenere il modello minimo di un programma definito. Usare il metodo descritto per determinare il modello minimo del programma:

$$\begin{array}{rcl} p(a). & & \\ q(f(X)) & \leftarrow & p(X). \\ p(g(g(f(Y)))) & \leftarrow & q(f(Y)). \end{array}$$

CAPITOLO 7

Programmazione in Prolog

1. Liste

Per definire/usare le liste in Prolog è sufficiente disporre di due simboli funzionali: un costruttore di liste di arità 2 (diciamo f) e un simbolo di costante che denota la lista vuota (diciamo `nil`). Pertanto, scegliendo questi due simboli, la lista $[a, b, c]$ può essere rappresentata come:

$$f(a, f(b, f(c, \texttt{nil})))$$

Tuttavia, vista la estrema utilità di questa struttura dati, nel linguaggio Prolog è prevista una facilitazione. Anzichè prevedere che il programmatore introduca dei generici simboli quali f e `nil`, Prolog mette a disposizione un simbolo funzionale binario $[\cdot|\cdot]$ e un simbolo di costante $[]$. Utilizzando quindi questa notazione, la lista $[a, b, c]$ viene denotata dal termine

$$[a \mid [b \mid [c \mid [] \ ] \ ] \ ].$$

Vi è inoltre un secondo livello di astrazione. La lista suddetta può essere rappresentata ancora piú succintamente come $[a, b, c]$.
In generale, la scrittura

$$[s_1, \ldots, s_n|t]$$

denota il termine

$$[s_1|[s_2| \ldots [s_n|t] \ldots]]$$

Inoltre, quando il termine t è la lista vuota $[]$, allora il termine $[s_1|[s_2| \ldots [s_n|[]] \ldots]]$ può essere scritto come $[s_1, \ldots, s_n]$.

Si osservi la differenza tra i due termini $[X, Y]$ e $[X|Y]$:

- come detto, $[X, Y]$ è una notazione abbreviata che denota il termine $[X|[Y|[]]]$, ovvero la lista che contiene esattamente i due elementi X e Y;
- il termine $[X|Y]$ denota invece lista ottenuta aggiungendo l'elemento X in testa alla lista Y.

Dichiariamo alcuni predicati sulle liste:

- ```
  member(X, [X | Resto]).
  member(X, [A | Resto]) :- member(X, Resto).
  ```
  Questo predicato definisce dichiarativamente la appartenenza di un elemento ad una lista, tramite una ricorsione sulla struttura del termine che rappresenta la lista.

  Approfittiamo di questa occasione per introdurre una ulteriore facilitazione notazionale permessa in Prolog. Si noti che nella seconda clausola la variabile `A` occorre una sola volta. Lo stesso dicasi per la variabile `Resto` della prima clausola. Variabili di questo tipo non vengono utilizzate per propagare gli effetti della unificazione ai letterali della clausola. Per questo fatto come nome per tale variabile puó essere
  ```

utilizzato il simbolo di *variabile anonima* "_". Si noti che ogni occorrenza del simbolo di variabile anonima denota una variabile differente. Con questa facilitazione le due clausole precedenti si possono scrivere come:

```
member(X, [X | _]).
member(X, [_ | Resto]) :- member(X, Resto).
```

- Il seguente predicato codifica un test soddisfatto solamente dai termini che hanno la struttura di lista (o, più correttamente: che possono essere istanziati ad una lista):
```
list([]).
list([A | B]) :- list(B).
```
- La proprietà di una lista di essere (unificabile al) prefisso di una altra lista:
```
prefix([], _ ).
prefix([A | R], [A | S]) :- prefix(R, S).
```
- La proprietà di una lista di essere (unificabile ad) una sottolista di una altra lista può essere invece descritta come:
```
sublist(X, Y) :- prefix(X, Y).
sublist(X, [_ | S]) :- sublist(X, S).
```
- Un predicato ternario che è verificato quando il terzo argomento unifica con la concatenazione dei primi due:
```
append([], X, X).
append([A | R], Y, [A | S]) :- append(R, Y, S).
```

Un'altra utile funzione relativa alle liste è `reverse`: un predicato binario soddisfatto quando i due argomenti sono due liste e la prima unifica con l'inversa della seconda.

(1) Una prima versione che calcola il reverse di una lista:
```
reversenaive([],[]).
reversenaive([A | R], S) :- reversenaive(R, Rrev)
                            append(Rrev, [A], S).
```
Si noti che la inversione di una lista tramite questo predicato ha un costo pari a $\frac{n^2}{2}$, con n dimensione della lista.

(2) Dichiariamo un predicato che permetta una ricerca più efficiente delle derivazioni di successo:
```
reverse(L, Lrev) :- reverse(L, [], Lrev).
reverse([X | R], Acc, Y) :- reverse(R, [X | Acc], Y).
reverse([], Y, Y).
```
Questa definizione permette di costruire derivazioni di successo con lunghezza minore. La variabile `Acc` (accumulatore) e viene utilizzata come stack in cui vengono via via inseriti gli elementi della lista da invertire, scandita dalla testa alla coda. Alla fine della scansione `Acc` sarà istanziata alla lista invertita.

Esempio 7.1. Supponiamo di voler invertire la lista `[a, b, c]`. Tramite il predicato `reverse` otteniamo la seguente sequenza di goal:

```
?- reverse([a, b, c], Lrev).
?- reverse([a, b, c], [], Lrev).
?- reverse([b, c], [a, [] ], Lrev).
?- reverse([c], [b, a], Lrev).
?- reverse([], [c, b, a], Lrev).
```

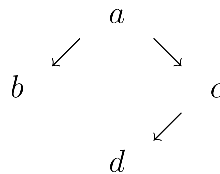
La risposta calcolata è $\theta = [\ Lrev/[c, b, a]\ ]$.

Esercizio 7.1. Si scriva un predicato binario che sia verificato quando i due argomenti sono liste e la prima è una permutazione della seconda.

Esercizio 7.2. Si scriva un predicato binario che sia verificato quando il primo argomento è una lista di numeri (utilizzando la rappresentazione dei numeri introdotta nella Sezione 3 del Capitolo 3) e il secondo argomento è la lista ordinata degli stessi numeri. [SUGGERIMENTO: si utilizzi il predicato `leq` definito a pag. 34 per effettuare i confronti tra numeri.]

2. Alberi

Contrariamente a quanto accade per le liste, in Prolog non vi sono simboli predefiniti per rappresentare gli alberi. La scelta è quindi delegata al programmatore. Utilizziamo ad esempio il simbolo ternario `tree` come costruttore di alberi binari e `nil` per denotare l'albero vuoto. Con questa convenzione l'albero



Viene rappresentato dal termine

```
tree(a, tree(b, nil, nil), tree(c, tree(d, nil, nil), nil))
```

ovvero:

```
tree(a,
     tree(b,
          nil,
          nil),
     tree(c,
          tree(d,
               nil,
               nil),
          nil))
```

Dove il primo argomento di `tree` indica l'etichetta del nodo padre mentre gli altri due argomenti denotano i due sottoalberi.
Basandoci su queste convenzioni, possiamo definire alcuni predicati che codificano delle proprietà degli alberi:

- il predicato unario `istree` sarà vero se l'argomento è un termine unificabile ad un albero:

```
istree(nil).
istree(tree(Label, L, R)) :- istree(L), istree(R).
```

- il predicato membertree codifica un test di occorrenza di un termine in un albero:

```
membertree(X, tree(X,_,_)).
membertree(X, tree(_,L,R)) :- membertree(X,L).
membertree(X, tree(_,L,R)) :- membertree(X,R).
```

- Il prossimo predicato codifica la visita in pre-ordine dell'albero. Le etichette di tutti i nodi vengono raccolte in una lista.

```
preorder(nil ,[]).
preorder(tree(X,LTree,RTree),[X|Res]) :- preorder(LTree,LList),
                                         preorder(RTree,RList),
                                         append(LList,RList,Res).
```

Esercizio 7.3. Si definiscano i predicati relativi alle visite in post-ordine ed in in-ordine.

Esercizio 7.4. Utilizzando le liste, si introduca una rappresentazione per alberi (etichettati) generici, in cui cioè non c'è un limite prefissato al numero di figli che ogni nodo può avere. Definire successivamente i predicati relativi alle visite in pre-ordine, post-ordine ed in-ordine.

3. Grafi

Vi sono due principali tecniche di rappresentazione/manipolazione di grafi in Prolog.

In primo approccio prevede di rappresentare un grafo elencando i suio archi tramite un insieme di fatti. In questo modo la descrizione del grafo è parte del programma. Un esempio di un grafo (diretto) descritto in questo modo:

```
edge(a ,b).
edge(b ,d).
edge(a ,c).
edge(c ,e).
edge(d ,f).
edge(e ,f).
```

Basandosi su questo approccio si possono descrivere dei predicati operanti su grafi, quali ad esempio un predicato che stabilisce se due nodi siano o meno connessi da un cammino:

```
connected(N,N).
connected(N1,N2) :- edge(N1,Int),
                    connected(Int,N2).
```

Esercizio 7.5. Si definisca un predicato cammino(N1,N2,Path) in grado di stabilire la lista dei nodi che occorrono nel cammino trovato per andare da N1 a N2. [SUGGERIMENTO: È plausibile che una semplice soluzione porti a costruire una lista di nodi che risulta invertita. Si utilizzi la tecnica dell'accumulatore per costruirla invece nell'ordine desiderato.]

Una seconda tecnica per utilizzare i grafi è quella di rappresentare il grafo stesso come un particolare termine Prolog: una lista di archi. In grafo precedente sarà quindi rappresentato dalla seguente lista:[1]

```
[edge(a,b), edge(b,d), edge(a,c), edge(c,e), edge(d,f), edge(e,f)]
```

In questo modo il grafo non viene codificato come parte del programma ma come dato del programma. Il predicato `connected` deve essere quindi definito in modo diverso:

```
connected(G,N,N).
connected(G,N1,N2) :- member(edge(N1,Int),G),
                      connected(G,Int,N2).
```

Soffermiamoci ancora sulle principali differenze tra i due approcci:

- Il secondo approccio vede il grafo gestito come *dato*. Si utilizza, in un certo senso, un maggiore grado di astrazione.
- Il primo caso invece permette di accedere ai singoli archi del grafo senza dover utilizzare il predicato `member` e quindi senza dover scandire una lista ogni volta che sia necessario verificare la presenza di un arco. In questo approccio la verifica di esistenza dell'arco è delegata direttamente all'interprete Prolog. Ciò permette di sfruttare le ottimizzazioni proprie che l'interprete adotta nel gestire le clausole del programma (tipicamente per ottimizzare la ricerca delle clausole si implementano delle strutture di hashing).

Esercizio 7.6. L'esistenza di cicli nel grafo (ad esempio a causa dell'aggiunta del fatto `edge(a,a).` al grafo precedente), può causare delle computazioni infinite durante la valutazione del predicato `connected`. Si scriva una versione di `connected` che non soffra di questo difetto.

Esercizio 7.7. I grafi con archi pesati (o etichettati) possono essere rappresentati aggiungendo un argomento al termine `edge` ed utilizzando ad esempio fatti del tipo `edge(a,16,b)`. Si definiscano:

(1) un predicato che stabilisca se due nodi sono connessi.
(2) Un predicato che oltre a far ciò calcola il *costo* del cammino trovato (come somma dei pesi degli archi).
(3) Un predicato che verifichi l'esistenza di un cammino di lunghezza minore ad un certo valore K fornito come input.

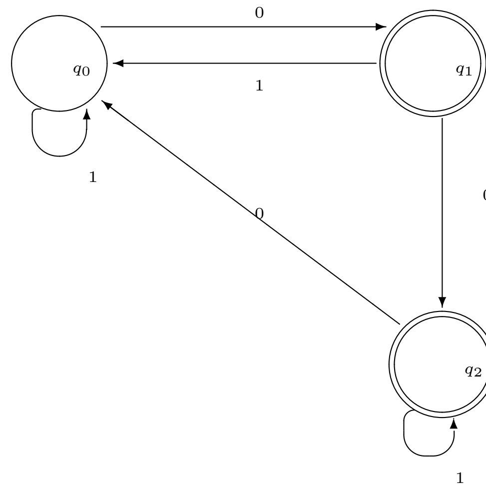
4. Automi finiti

Gli automi finiti possono essere rappresentati similmente ai grafi. Ad esempio consideriamo l'automa

[1] Si noti che una rappresentazione equivalente a questa non fa uso del simbolo di predicato `edge` ma utilizza liste di due elementi per rappresentare gli archi:

```
[[a,b], [b,d], [a,c], [c,e], [d,f], [e,f]]
```

Nel seguito avremo ripetute occasioni di utilizzare questa rappresentazione per i grafi.



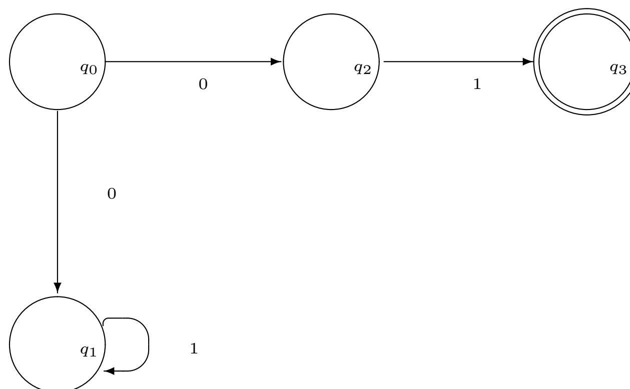
questo automa può essere descritto dichiarando quali siano gli stati iniziali e finali e rappresentando la sua funzione di transizione δ. Il tutto può essere effettuato asserendo i seguenti fatti:

```
delta(q0,0,q1).
delta(q0,1,q0).
...
initial(q0).
final(q1).
final(q2).
```

A questo punto si possono descrivere i predicati che chiudono transitivamente e riflessivamente sia la funzione di transizione δ, che la funzione di accettazione:

```
delta_star(Q,[],Q).
delta_star(Q,[A|R],Qout):- delta(Q,A,Qint),
                            delta_star(Qint,R,Qout).
accetta(S) :- initial(Q_0),
              delta_star(Q_0,S,Q_out),
              final(Q_out).
```

La definizione è estremamente immediata e dichiarativa. Inoltre, lo stesso programma funziona correttamente anche nel caso di automi non deterministici. Consideriamo ad esempio il seguente automa non deterministico



Esso è rappresentabile come:

```
delta(q0,0,q1).
delta(q0,0,q2).
delta(q1,1,q1).
delta(q2,1,q3).
initial(q0).
final(q3).
```

5. Predicati built-in

Descriviamo ora alcuni comandi del linguaggio Prolog che, in virtù della Turing completezza potrebbero essere realizzati tramite programmi definiti. Tuttavia, sia per ragioni di efficienza, sia per facilitare la attività di programmazione e per migliorare la espressività delle implementazioni reali del linguaggio, tali predicati sono *built-in* in molti interpreti Prolog.

- Il predicato `is` risulta verificato se il valore dell'espressione numerica fornita alla sua destra è unificabile con il termine alla sua sinistra. Dato che `is` procede a valutare l'espressione, al momento in cui la valutazione viene effettuata le variabili che occorrono alla destra di `is` devono essere opportunamente istanziate. Alcuni esempi:

```
?- 3 is 1+2.
    yes
?- X is 1+2.
    yes X=3
?- 2+1 is 1+2.
    no
?- 8 is X*2.
    instantiation error
?- X is X.
    instantiation error
```

- Il predicato `<` effettua il confronto tra valori numerici. I due termini vengono valutati e successivamente si confrontano i valori ottenuti. Se la valutazione non è possibile (ad esempio per insufficiente istanziazione o perchè uno dei due termini non è un numero) viene generato un errore. Alcuni esempi

```
?- 2 < 3.
    yes
?- 13 < 2+4.
    no
?- X < 3.
    instantiation error
?- a < 3.
    domain error:  expected expression
```

  Vi sono a disposizione anche gli altri usuali predicati di confronto: `>`, `>=`, e `=<`.
- Per effettuare il confronto tra termini che non sono costanti numeriche si può utilizzare i corrispondenti predicati `@<`, `@>`, `@=<`, e `@>=`. In questo caso il risultato del

confronto viene determinato in base ad un predefinito ordinamento totale di tutti i termini del Prolog. Ad esempio in SICStus Prolog vigono le seguenti convenzioni:

(1) Una variabile precede ogni altro termine non variabile. Tra due variabili, la "più vecchia" precede l'altra; seguono
(2) i numeri reali, nell'ordine usuale; seguono
(3) i numeri interi, nell'ordine usuale; seguono
(4) gli atomi, nell'ordine lessicografico; seguono
(5) i termini composti, ordinati prima rispetto all'arità del simbolo più esterno, poi lessicograficamente rispetto al simbolo più esterno, poi considerando i sottotermini (procedendo da sinistra a destra). Quindi, ecco un esempio di lista di elementi ordinata secondo tale ordine:

```
[Z, -1.0, -90, 10, fi, fo, fo(0), X=Y, fi(1,1), fo(1,1), fo(3,10)]
```

- Il predicato `=` forza l'unificazione tra due termini. Risulta quindi soddisfatto se i due termini possono essere unificati, e in tal caso l'unificazione viene effettuata (quindi le eventuali variabili vengono opportunamente istanziate). Qualora l'unificazione avvenga, viene fornito un m.g.u..

```
?- 2 = 2.
    yes
?- a = b.
    no
?- X = 2.
    yes X=2
?- f(a,V,H)=f(G,h(G,G),R).
    yes G=a, R=H, V=h(a,a)
```

- La negazione di questo predicato è espressa dal predicato `\=`. Il letterale `s\=t` risulta vero quando `s` e `t` non sono unificabili.
- Per effettuare il confronto di uguaglianza sintattica, senza che venga eseguita l'unificazione, si impiega il predicato `==`.

```
?- 2 == 2.
    yes
?- a == b.
    no
?- X == 2.
    no
?- X = 2, X == 2.
    yes X=2
?- X == 2, X = 2.
    no
```

- La negazione di questo predicato è data da `\==`.
- Tramite il predicato `=:=` si valuta l'uguaglianza di espressioni numeriche ground Anche in questo caso, al momento della valutazione delle due espressioni le variabili che vi occorrono devono essere opportunamente istanziate. Alcuni esempi:

Unificazione	`=`	`\=`
Uguaglianza sintattica	`==`	`\==`
Uguaglianza tra espressioni	`=:=`	`=\=`

FIGURA 7.1. Predicati connessi alla nozione di uguaglianza

```
?- 2+1 =:= 4-1.
    yes
?- X =:= 2.
    instantiation error
```

- La diseguaglianza è invece valutabile tramite il predicato `=\=`.

La Figura 7.1 riassume le varie nozioni di "uguaglianza".

6. Predicati di tipo e manipolazione di termini

Vi sono a disposizione dei predicati utili a discriminare tra differenti tipi di termini.

- Il predicato `integer(N)` è soddisfatto da un termine che rappresenta un numero intero.

```
?- integer(0).
    yes
?- integer(1).
    yes
?- integer(1.2).
    no
?- integer(1+2).
    no
```

  Si noti che anche quando la valutazione della espressione restituirebbe un numero intero, il valore di verità di `integer(N)` è false in quanto la espressione di per sé è un termine composto e non un intero. Ad esempio, nell'ultimo dei goal precedenti, la espressione `1+2` non viene valutato ma viene trattato come il termine Prolog `+(1,2)`.
- Analogo è il predicato `real(N)`. È soddisfatto se il termine rappresenta un numero reale.[2]

```
?- real(1.2).
    yes
?- real(c).
    no
```

- Il predicato `atom(c)` è soddisfatto solo se il termine `c` è una costante non numerica.[3]

[2] In molti Prolog, come ad esempio GNU-Prolog e SICStus Prolog il predicato `real` è rimpiazzato dal predicato `float`. In generale, ogni implementazione di Prolog, fornisce un insieme di predicati più ricco di quello descritto in queste pagine. Si veda in merito il manuale relativo alla particolare installazione a vostra disposizione.

[3] Si noti la possibile fonte di confusione generata dall'impiego della parola "atomo" per indicare una costante non numerica di Prolog e come sinonimo di "formula atomica".

```
?- atom(c).
    yes
?- atom(p(c)).
    no
?- atom(1).
    no
```

- Il predicato `number(N)` risulta vero se il termine fornito è istanziato ad un numero.

```
?- number(1).
    yes
?- number(1.2).
    yes
```

- Il predicato `compound(X)` viene invece soddisfatto ogni qualvolta l'argomento è istanziato ad un termine composto.

```
?- compound(f(X,Y)).
    true
?- compound(c).
    no
```

- Il predicato `functor(Term,F,Arity)` è verificato se `F` e `Arity` possono essere istanziati rispettivamente al simbolo funzionale del termine `Term` ed alla sua arità.

```
?- functor(f(a,b,c),f,3).
    yes
?- functor(f(a,b,c),F,N).
    yes F=f, N=3
?- functor(g(a),f,2).
    no
?- functor(T,f,4).
    yes T=f(_,_,_,_)
?- functor(T,F,4).
    instantiation error
```

- Il predicato `arg(N,Term,Arg)` può essere impiegato per estrarre un argomento di un termine.

```
?- arg(2,f(a,b,c),b).
    yes
?- arg(3,f(a,b,c),X).
    yes X=c
?- arg(N,f(a,b,c),c).
    instantiation error
```

  Si noti che, relativamente a predicati quali `arg/3` o `functor/3` vi possono essere delle differenze tra diversi interpreti Prolog nel caso di argomenti non instanziati. Ad esempio il goal `arg(N,f(a,b,c),c)` (con `N` variabile non istanziata) genera un "`instantiation error`" se invocato in SICStus Prolog o in GNU-Prolog, mentre genera la risposta (intuitivamente attesa) "`yes N=3`" in SWI-Prolog.
- L'operatore *univ*, denotato dal simbolo `=..` può essere impiegato per costruire termini o per accedere a sottotermini. Ha successo se l'argomento di destra è unificabile

ad una lista i cui membri sono, nell'ordine, il simbolo funzionale e gli argomenti del termine che compare alla sua sinistra. Qualora uno o entrambi gli argomenti non siano sufficientemente istanziati, viene generato un errore di istanziazione.

```
?- f(a,b,g(X,Y)) =..  [f,a,b,g(X,Y)].
    yes
?- f(a,b,g(X,Y)) =..  [F,X,b,g(X,Y)].
    yes F=f, X=a
?- X =.. Y.
    instantiation error
?- X =.. [Y,a,b].
    instantiation error
```

7. Predicati metalogici o extralogici

Altri predicati che sono inclusi in molte implementazioni di Prolog al fine di rendere la programmazione più semplice.

- Il predicato `var(Term)` è soddisfatto se al momento della sua valutazione il termine `Term` è una variabile.

```
?- var(2).
    no
?- var(X).
    yes
?- var(f(X)).
    no
```

- Il predicato `nonvar(Term)` invece è soddisfatto se al momento della sua valutazione il termine `Term` non è una variabile.

```
?- nonvar(2).
    yes
?- nonvar(X).
    no
```

Esempio 7.2. Utilizzando `var(Term)` e `nonvar(Term)` possiamo fornire una diversa versione della definizione del predicato `plus` data nella Sezione 4 del Capitolo 3:

```
plus(X,Y,Z) :- var(Z), nonvar(Y), nonvar(Z), Z is X+Y.
plus(X,Y,Z) :- var(X), nonvar(Y), nonvar(Z), X is X-Y.
```

Esercizio 7.8. Definire in Prolog

(1) un predicato `thesize(Term,N)` che calcola la *size* di un termine (numero occorrenze di simboli funzionali e di costante).
(2) un predicato `high(Term,H)` tale che `H` sia la altezza dell'albero associato al termine `Term`.

Esercizio 7.9. Senza usare gli eventuali analoghi predicati di Prolog, definire:[4]

[4] Alcune implementazioni di Prolog offrono alcuni di questi predicati come built-in. Ad esempio, in SICSstus esiste il predicato `ground(T)` che risulta vero se il termine `T` è ground. È tuttavia possibile che in diversi interpreti Prolog questi predicati abbiano nomi diversi.

(1) un predicato `isground(Term)` vero se `Term` è ground.
(2) un predicato `isnotground(Term)` vero se `Term` non è ground.
(3) un predicato `unifica(T1,T2)` che effettui l'unificazione (con occur check) di due termini.
(4) un predicato che stabilisca (e trovi le eventuali soluzioni) se una formula logica proposizionale sia o meno soddisfacibile.

8. Predicati di input e output

Un particolare genere di predicati Prolog è quello dei predicati dedicati a input e output. Vi sono diversi modi per effettuare scritture o letture da file o da/su tastiera/schermo. Inoltre molte implementazioni Prolog posseggono solitamente un ricco insieme di predicati di input/output in aggiunta a quelli definiti nel linguaggio standard.

In quanto segue accenneremo solamente a due predicati, `read` e `write`, rimandando al manuale Prolog per la descrizione delle altre possibilità offerte.

Il predicato `read(T)` ha successo se è possibile leggere dallo standard input (solitamente corrisponde alla tastiera) una sequenza di caratteri (che termina con il punto) e questa sequenza (escluso il punto) compone un termine unificabile con il termine `T`.

Il predicato `write(T)` ha successo e scrive sullo standard output (solitamente il terminale) il termine `T`.

Utile è il predicato `nl`. Esso ha successo scrivendo un new-line sullo standard output. Si noti che i predicati di input/output di solito hanno successo una sola volta, ovvero non vengono rivalutati durante il backtracking.

Possiamo quindi semplicemente definire un utile predicato:

```
writeln(T) :- write(T), nl.
```

Esso dopo aver scritto un termine va a capo.

9. Il CUT

Descriviamo in questa sezione una utile funzionalità offerta da Prolog allo scopo di controllare il processo di SLD-derivazione. Il CUT, denotato con il simbolo `!`, è un atomo che ha sempre successo. Il suo scopo consiste nel limitare il backtracking che l'interprete attua durante la ricerca delle soluzioni ad un goal. Intuitivamente si può pensare che l'effetto del CUT sia quello di tagliare una parte dell'SLD-albero in modo che l'interprete non la visiti.

Più precisamente: chiamiamo per semplicità *parent goal* il goal la cui risoluzione ha causato l'impiego di una clausola il cui corpo contiene un CUT. Quindi a seguito di tale passo di SLD-risoluzione il CUT viene inserito nel goal attuale. Successivamente il CUT verrà (prima o poi) selezionato come prossimo atomo del goal ad essere risolto. A questo punto il passo di SLD-risoluzione si compie semplicemente senza la ricerca della clausola applicabile in quanto il CUT non ne necessita (ha immediato successo, e viene quindi rimosso dal goal). Supponiamo ora che per effettuare backtracking sia necessario individuare un choice point precedente al punto in cui si è valutato il CUT. *La presenza del CUT forza l'interprete ad ignorare ogni choice point appartenente al sottoalbero che ha il parent goal come radice. La ricerca del choice point comincia quindi dal goal che precede il parent goal.* (Chiaramente, nel caso particolare in cui il parent goal è proprio il goal iniziale, la computazione ha termine.)

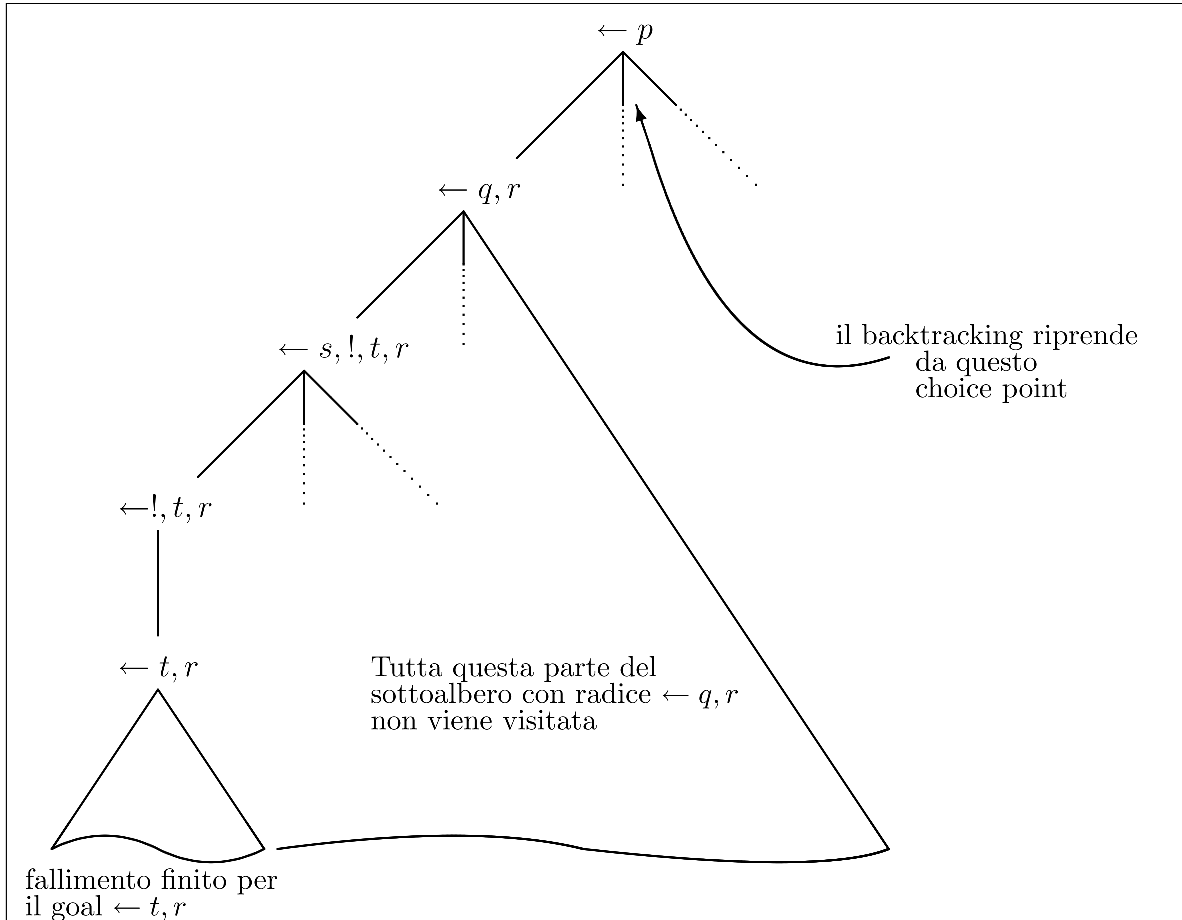


FIGURA 7.2. Effetto del CUT

Consideriamo il seguente programma ove per semplicità indichiamo esplicitamente solo i letterali rilevanti. Inoltre supponiamo che l'unico CUT presente sia quello nella clausola C_4.

C_1 : `p :- q, r.`
C_2 : `p :- ...`
C_3 : `p :- ...`
C_4 : `q :- s, !, t.`
C_5 : `q :- ...`
C_6 : `q :- ...`
C_7 : `s.`
C_8 : `s :- ...`
...

In Figura 7.2 viene illustrato l'SLD-albero relativo al goal $\leftarrow p$. Il parent goal è in questo caso il goal $\leftarrow q, r$. La selezione del CUT quale atomo da processare nel goal $\leftarrow !, t, r$, comporta il successo del passo di SLD-risoluzione con la produzione del nuovo goal $\leftarrow t, r$. Supponiamo che a questo punto non si trovi alcuna soluzione per questo goal, ovvero si genera un sotto-albero di fallimento finito per $\leftarrow t, r$ (una situazione analoga si originerebbe se si trovassero delle soluzioni, ma l'utente ne chiedesse altre digitando ";"). Se il CUT fosse

ignorato, il backtracking porterebbe al choice point più vicino, ovvero a processare clausole alternative per risolvere l'atomo s nel goal $\leftarrow s, !, t, r$. Come illustrato in Figura 7.2, la presenza del CUT fa sì che la ricerca di SLD-derivazioni riprenda dal choice point in cui è stato prodotto il parent goal. L'effetto è quello di evitare la visita della parte non ancora visitata del sottoalbero che ha come radice il parent goal.

Consideriamo un ulteriore esempio. Il programma

```
p(X):- q(X), r(X).
p(c).
q(a).
q(b).
r(b).
```

Secondo quanto studiato nel Capitolo 6, la semantica del programma, unitamente al goal

```
?- p(X).
```

risulta completamente descrivibile tramite l'SLD-albero in Figura 7.3. Chiaramente ci si attende che l'interprete Prolog fornisca le risposte calcolate $X = b$ e $X = c$.

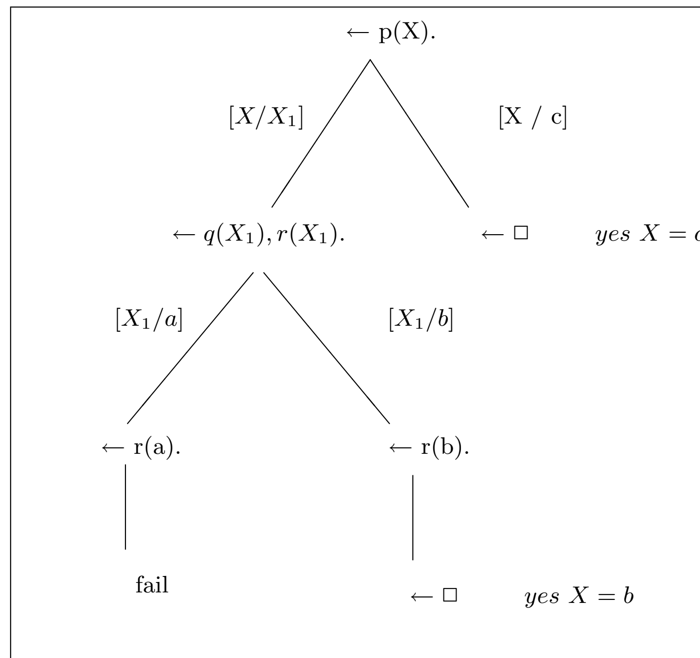


FIGURA 7.3. SLD-albero senza CUT

Consideriamo ora il programma ottenuto inserendo un CUT:

```
p(X):- !, q(X), r(X).
p(c).
q(a).
q(b).
r(b).
```

Il CUT blocca il backtracking impedendo di trovare la soluzione $X = c$ ma lasciando accessibile la soluzione $X = b$. Una illustrazione diagrammatica del funzionamento del CUT si può

ottenere inserendo nell'SLD-albero degli archi direzionati (indicati con il simbolo del *diodo*). Ciò ad indicare che questi archi possono essere attraversati solo in un verso (Figura 7.4). Si noti che un uso non controllato del CUT può impedire all'interprete di trovare soluzioni. Ad esempio, sottoponendo il goal $\leftarrow p(X)$ relativamente al programma seguente, non si ottiene alcuna risposta calcolata, (mentre una risposta esiste per il programma privo del CUT).

```
p(X):- q(X), !, r(X).
p(c).
q(a).
q(b).
r(b).
```

Solitamente si ritiene corretto un uso del CUT che non cambi la semantica del programma, ovvero che non modifichi l'insieme delle soluzioni calcolate. A seconda che ciò accada o meno, si identificano due classiche metodologie di utilizzo del CUT.

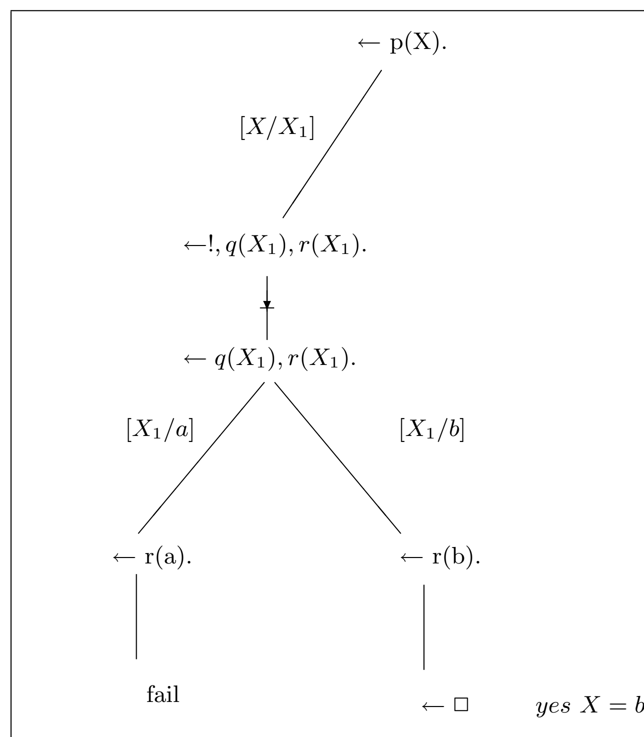


FIGURA 7.4. Effetto del CUT sull'SLD-albero di Figura 7.3

9.1. Il CUT verde. È il caso del CUT "corretto" ovvero che non rende irraggiungibili alcune delle soluzioni. L'effetto è quindi sempre quello di tagliare l'SLD-albero, ma la parte tagliata non contiene soluzioni. Si elimina quindi parte del non-determinismo del programma: l'inserimento "saggio" del CUT in corrispondenza di choice point permette di evitare la visita di sotto-alberi privi di soluzioni. Ad esempio, nel seguente programma, i CUT seguono delle condizioni mutualmente esclusive: se l'atomo `X<Y` ha successo allora né l'atomo `X==Y`, né l'atomo `Y<X` possono avere successo. I CUT permettono all'interprete di ridurre la parte dell'SLD-albero da visitare.

```
merge([X|X_s],[Y|Y_s],[X|Z_s]) :- X<Y, !, merge(X_s,[Y|Y_s],Z_s).
merge([X|X_s],[Y|Y_s],[X,Y|Z_s]) :- X==Y, !, merge(X_s,Y_s,Z_s).
merge([X|X_s],[Y|Y_s],[Y|Z_s]) :- Y<X, !, merge([X,X_s],Y_s,Z_s).
```

Si noti che il CUT nell'ultima clausola può essere omesso in quanto non vi sono comunque altre scelte possibili.

9.2. Il CUT rosso. Un CUT che modifica l'insieme delle soluzioni calcolabili è detto CUT rosso. Programmi che sembrano uguali, ma differiscono solo per la presenza di CUT rossi, hanno quindi comportamenti operazionali diversi.

Consideriamo il seguente programma:

```
norep_member(X,[X|X_s]).
norep_member(X,[Y|Y_s]) :- X\==Y, norep_member(X,Y_s).
```

Esso è ottenuto modificando la definizione del predicato `member` vista a pag. 89. Questo programma controlla se un elemento occorre in una lista. Operativamente, il controllo procede scandendo la lista fino a che l'elemento viene trovato o si giunge alla fine della lista. Solo una soluzione viene quindi generata anche nel caso in cui l'elemento occorra più volte nella lista. Si veda ad esempio l'SLD-albero per il goal $\leftarrow norep_member(a,[a,b])$ in Figura 7.5.

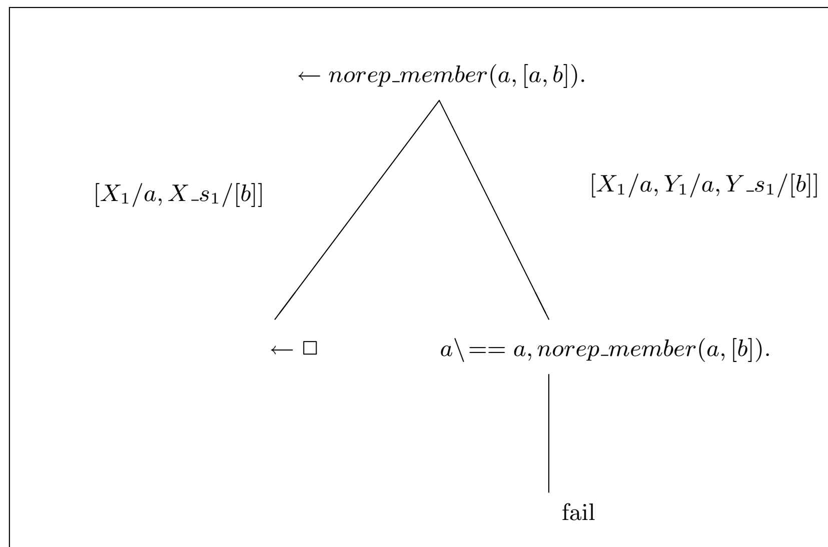


FIGURA 7.5. SLD-albero relativo alla definizione di *norep_member* senza CUT

Quindi nell'intento del programmatore è sempre sufficiente determinare una sola soluzione (la prima). Lo stesso effetto si può ottenere utilizzando un CUT (rosso) che eviti all'interprete di trovare altre soluzioni. Il programma può essere riscritto quindi come segue:

```
norep_member(X,[X|X_s]) :- !.
norep_member(X,[Y|Y_s]) :- norep_member(X,Y_s).
```

In questo modo, invece di generare il fallimento tramite l'atomo `X\==Y`, si impedisce di attivare il backtracking sulla seconda clausola ogni volta che la prima è stata impiegata (si veda Figura 7.6).

In chiusura di questa sezione ricordiamo ancora che un CUT (verde o rosso) comporta solitamente una diminuzione del tempo di computazione necessario per risolvere un goal.

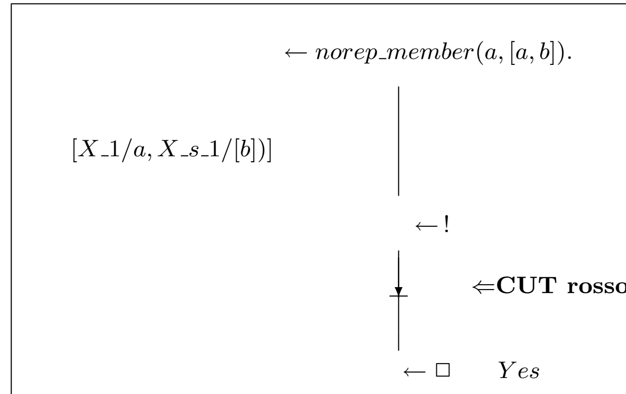


FIGURA 7.6. Effetto del CUT rosso della definizione di *norep_member*

Questo perchè, come detto, parte dell'SLD-albero non viene visitata. Nel caso del CUT rosso si ha anche una modifica dell'insieme delle soluzioni ottenibili. È necessario quindi porre attenzione nell'uso dei CUT al fine di non introdurre inconsapevolmente CUT rossi. Il comportamento dell'interprete Prolog in questo caso potrebbe non corrispondere a quello atteso.

10. Il predicato FAIL

Il predicato `fail` è un predicato nullario (privo di argomenti) che fallisce sempre. Lo si può pensare come un atomo il cui valore è sempre false (ad esempio si potrebbe scrivere, in sua vece, l'atomo $a = b$).

Analizziamone un esempio d'uso utilizzando l'albero genealogico dell'esempio della Figura 3.1 (che riportiamo per comodità in Figura 7.7). Abbiamo visto che un programma definito che rappresenta questa situazione è il seguente:

```
padre(antonio,bruno).
padre(antonio,carlo).
padre(bruno,davide).
padre(bruno,ettore).
```

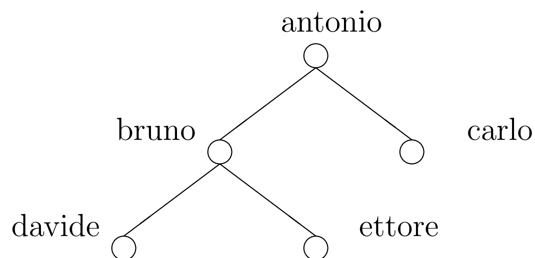


FIGURA 7.7. Albero genealogico

Scriviamo un predicato che stampi sullo schermo le coppie padre-figlio:

```
padri :- padre(X,Y),
         write(X),
         write(' padre di ')
         write(Y),
         nl,
         fail.
```

L'SLD-albero di derivazione (semplificato nei write) è riportato in Figura 7.8.
L'interprete Prolog, alla ricerca di una SLD-derivazione di successo per il goal `?- padri.`, visita tutto l'SLD-albero. Tuttavia, a causa del predicato `fail`, l'SLD-albero è un albero di fallimento finito, quindi non ci sarà alcuna risposta calcolata e la risposta finale sarà `no`. Nel cercare la soluzione però, l'interprete istanzierà in tutti i modi ammessi le variabili `X` e `Y` e ad ogni valutazione dei predicati `write(X)` e `write(Y)` verrà prodotta una stampa.

La tecnica di programmazione che abbiamo appena illustrato realizza in Prolog una iterazione molto simile al tipo di iterazione che si può utilizzare in linguaggi imperativi, ad esempio tramite il costrutto repeat-until. Questo modo di sfruttare il backtracking per realizzare iterazioni è detto *failure-driven loop*.

Se come esito finale della esecuzione del goal `:- padri` avessimo voluto ottenere `yes` come risposta, avremmo dovuto aggiungere una ulteriore clausola (un fatto) in coda al precedente programma:

```
padri.
```

Così facendo si aggiunge una unica (e banale) SLD-derivazione di successo che viene trovata solo dopo aver visitato tutto l'SLD-albero.

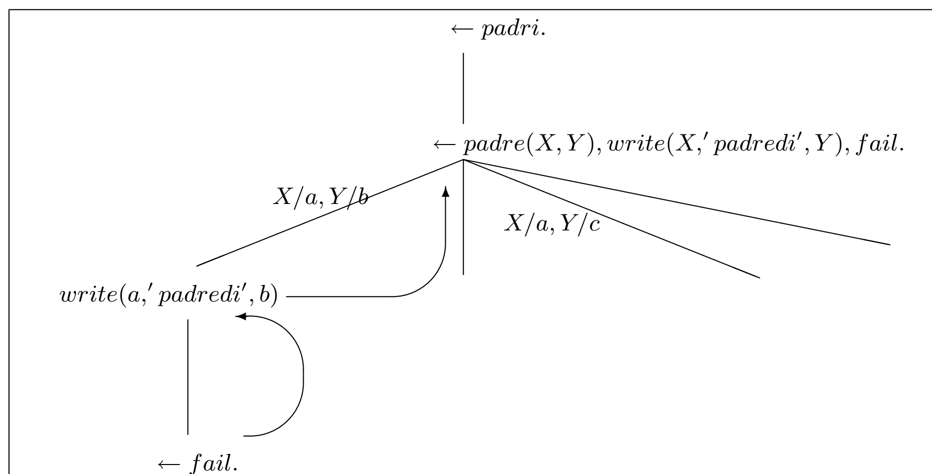


FIGURA 7.8. Uso combinato di `write` e `fail`

11. Operatori

Prolog offre la possibilità di definire, tramite delle direttive solitamente poste all'inizio del programma, degli *operatori* infissi, postfissi e prefissi. Questo è un modo per svincolare l'utente dall'obbligo di impiegare le parentesi nella scrittura dei termini. Analogamente a quanto accade scrivendo `2+5*3` invece di `+(2,*(5,3))`.

La definizione di un operatore avviene tramite una direttiva del tipo:

```
:- op(numero, tipologia, opers).
```

dove *numero* è un intero che indica la priorità dell'operatore che si sta definendo (più è alto il numero più bassa è la priorità); *opers* è l'operatore che si sta definendo, oppure una lista di operatori (in questo caso avranno tutti la stessa priorità e tipologia); *tipologia* indica se l'operatore sia infisso, prefisso o suffisso secondo queste regole:

- `xfx` definisce un operatore binario infisso;
- `xfy` definisce un operatore binario infisso, associativo a destra. Quindi, se si dichiara

  ```
  :- op(1100,xfy,bum)
  ```

  allora il termine `(a bum b bum c)` corrisponde a `bum(a,bum(b,c))`;
- `yfx` definisce un operatore binario infisso, associativo a sinistra;
- `fx` definisce un operatore unario prefisso;
- `fy` definisce un operatore unario prefisso, associativo;
- `xf` definisce un operatore unario suffisso;
- `yf` definisce un operatore unario suffisso, associativo.

Ecco alcuni esempi di direttive che definirebbero alcuni operatori spesso utilizzati in Prolog. Si noti che questi operatori sono però già predefiniti in Prolog.

```
:- op( 500, yfx, [ +, - ]).
:- op( 400, yfx, [ *, /, mod, rem ]).
:- op( 200, xfx, [ ** ]).
:- op( 200, fy, [ +, - ]).
```

12. Meta-variable facility

Una peculiare ed utile caratteristica di Prolog è che esso utilizza gli stessi strumenti per rappresentare i dati e i programmi. La struttura di un termine è infatti, in genere, del tipo `simbolo(`$\texttt{arg}_1$`,...,`$\texttt{arg}_k$`)`. Questa è la stessa struttura di una formula atomica (se deroghiamo dalla distinzione tra simboli di funzione e di predicato). Ne consegue che tramite i *costruttori* di termini descritti nella Sezione 6 è possibile costruire dinamicamente sia termini che letterali.

A questo proposito osserviamo la presenza di operatori particolari, automaticamente disponibili in Prolog, che vengono utilizzati nella costruzione delle clausole stesse. Tra essi riconosciamo "`:-`" e "`,`" che potrebbero essere definiti così:

```
:- op( 1200, xfx, :- ).
:- op( 1000, xfy, , ).
```

Questa possibilità di trattare alla stessa stregua termini e predicati (ovvero dati e programmi) permette ad un programma Prolog di manipolare le clausole stesse allo stesso modo dei termini. Vedremo nella Sezione 4 del Capitolo 9 come la possibilità di manipolare atomi permetta di realizzare *meta-programmi* ovvero programmi che utilizzano altri programmi come dati.

Una funzionalità offerta da Prolog che rientra in questo ordine di idee va sotto il nome di *meta-variable facility.* Essa permette ad una variabile di apparire come letterale nel corpo di una clausola. Il programmatore dovrà farsi carico di assicurare che, durante la

computazione, quando il processo di SLD-risoluzione giungerà a selezionare una variabile come prossimo atomo da risolvere, tale variabile sia già stata istanziata ad un termine che sia (sintatticamente) accettabile come corpo di una clausola (ovvero uno o più letterali separati da “,”).

Il prossimo esempio illustra un possibile utilizzo della meta-variable facility.

Esempio 7.3. Si vuole realizzare un programma Prolog che legga un goal che l'utente digita da tastiera. Una volta acquisito, il goal viene sottoposto all'interprete (ovvero si innesca lo stesso processo di valutazione che si sarebbe attivato se l'utente avesse sottoposto il goal direttamente all'interprete Prolog).

Il seguente codice Prolog implementa una semplice shell che presenta un prompt e legge un termine tramite `read(GOAL)`. Successivamente si utilizza la meta-variable facility per eseguire il goal `GOAL`: (Si noti che la shell così realizzata è minimale; ad esempio, manca della gestione degli errori ed assume che il termine digitato dall'utente sia sintatticamente corretto).

```
shell :- scriviprompt, read(GOAL), esegui(GOAL).
esegui(exit) :- !.
esegui(GOAL) :- ground(GOAL), !, risolviGoalGround(GOAL), shell.
esegui(GOAL) :- risolviGoalNonGround(GOAL), shell.
risolviGoalNonGround(GOAL) :- GOAL, write(GOAL), nl, fail.
risolviGoalNonGround(GOAL) :- write('Soluzioni finite'), nl.
risolviGoalGround(GOAL) :- GOAL, !, write('Yes'), nl.
risolviGoalGround(GOAL) :- write('No'), nl.
scriviprompt :- write('Digita un goal?  ').
```

Si noti in particolare:

- l'uso del CUT per impedire il backtracking nelle clausole relative ai predicati `esegui` e `risolviGoalGround`.
- Il diverso trattamento riservato ai goal ground: per essi una soluzione è ritenuta sufficiente.
- Il modo in cui ogni soluzione ad un goal non ground viene comunicata. Ciò avviene tramite l'atomo `write(GOAL)`. Questo viene sempre risolto dopo aver risolto l'atomo `GOAL` che lo precede nel corpo della clausola. D'altronde la risoluzione di `GOAL` (visto nella sua veste di atomo) causa la sua istanziazione (visto nella sua veste di termine). La stampa del termine `GOAL` istanziato a seguito della sia risoluzione è in pratica la stampa di una soluzione.
- L'impiego di un failure-driven loop per enumerare tutte le soluzioni dei goal non ground. Per essi infatti si sfrutta pienamente il backtracking (sull'atomo `GOAL`).

Solitamente gli interpreti Prolog mettono a disposizione il predicato meta-logico

`call(·)`

La invocazione di `call(Goal)` dove `Goal` è opportunamente istanziato ad un termine che sia accettabile come corpo di una clausola, causa la esecuzione del goal `Goal`.[5] Quindi la meta-variable facility non è altro che una convenzione sintattica per una invocazione a `call(·)`.

[5]Alcuni interpreti Prolog, e questo è il caso di SICStus (ma non di GNU-Prolog), permettono l'uso della meta-variable facility, e quindi di `call(Goal)`, solamente se il predicato impiegato nel goal `Goal` è stato

Ad esempio la clausola

```
risolviGoalNonGround(GOAL) :- GOAL, write(GOAL), nl, fail.
```

del programma dell'Esempio 7.3 potrebbe essere scritta come:

```
risolviGoalNonGround(GOAL) :- call(GOAL), write(GOAL), nl, fail.
```

13. Esercizi

Esercizio 7.10. Si assuma che un grafo diretto sia rappresentato da un insieme di fatti del tipo `arco(A,B)`. Scrivere un programma Prolog in cui si definisca il predicato `path(X,Y,Cammino)`. Tale predicato sarà soddisfatto quando `Cammino` è la lista dei nodi di un cammino semplice (ovvero che passa al più una volta per ogni nodo) che dal nodo `X` porta al nodo `Y` (l'ordine dei nodi in `Cammino` è ovviamente quello in cui si incontrano percorrendo il cammino).

Esercizio 7.11. Scrivere un programma Prolog in cui si definisca il predicato

`prof(Termine1,Prof).`

Il predicato deve essere vero quando `Prof` è la profondità del termine `Termine1` (ovvero il numero massimo di nidificazioni di funtori). Si assuma 1 la profondità delle costanti e 0 la profondità delle variabili.

Esercizio 7.12. Si assuma che un grafo diretto sia rappresentato da un insieme di fatti del tipo `arco(A,B)`. Scrivere un programma Prolog in cui si definisca il predicato `ciclo(X)`. Tale predicato sarà soddisfatto quando il nodo `X` appartiene a (almeno) un circuito (ovvero un cammino che parte da `X` e vi torna dopo aver attraversato almeno un arco).

Esercizio 7.13. Scrivere un programma Prolog in cui si definisca il predicato

`penultimo(Lista,Elemento).`

Il predicato deve essere vero quando `Lista` è una lista di almeno due elementi e `Elemento` unifica con il penultimo elemento della lista `Lista`.

Esercizio 7.14. Scrivere un programma Prolog in cui si definisca il predicato

`palindroma(Lista).`

Il predicato deve essere vero quando `Lista` è istanziato ad una lista palindroma di costanti, ovvero che "si legge allo stesso modo nelle due direzioni". Ad esempio `[e,2,3,d,d,3,2,e]` è palindroma.

Si scriva un secondo programma Prolog in cui si definisca un predicato

`palindromaVar(+Lista).`

che abbia successo quando `Lista` è una lista di termini Prolog (quindi può contenere atomi, variabili e/o termini composti) e tale lista risulta essere palindroma se si ignorano le differenze tra nomi di variabili (Ad esempio `palindromaVar([A,f,Z,h(Y),c,h(X),Y,f,A])` sarà vero, mentre `palindromaVar([a,f,Z,h(Y),h(X),c,f,a])` sarà falso).

---

dichiarato di tipo `dynamic`, con una direttiva quale:

`:- dynamic(`*nomepredicato*`).`

I predicati di un programma in genere vengono compilati durante l'operazione di consult del file che li contiene. Per i predicati di tipo `dynamic` la consultazione e la gestione può avvenire invece in modo differente (si veda anche Sezione 3.2 del Capitolo 9).

Esercizio 7.15. Scrivere un programma Prolog in cui si definisca il predicato

espandi(ListaCoppie,Lista).

Il predicato deve essere vero quando l'argomento ListaCoppie è istanziato una lista di coppie *(intero,termine)*, e Lista unifica con la lista ottenuta da ListaCoppie rimpiazzando ogni coppia [n,t] con n occorrenze consecutive del termine t. Ad esempio, sarà vero che:

espandi([[4,a],[1,b],[2,c],[2,a],[3,d]], [a,a,a,a,b,c,c,a,a,d,d,d])

(assumendo per semplicità che *intero* sia non nullo e i termini siano ground).

Esercizio 7.16. Scrivere un programma Prolog in cui si definisca il predicato

mymember(+Elemento,+Lista).

Si assuma che il predicato venga invocato con i due parametri istanziati a: Lista una lista di termini qualsiasi (anche variabili), e Elemento un termine qualsiasi (anche una variabile). Il predicato deve essere vero quando Elemento occorre in Lista. Si richiede che né Elemento né Lista vengano istanziati dal predicato.

Esercizio 7.17. Si consideri liste ordinate (in ordine crescente) di interi. Scrivere un programma Prolog in cui si definisca il predicato

ounion(+OListai1,+OLista2,?OLista).

Il predicato sarà vero quando OLista è la lista ordinata e senza ripetizioni degli elementi che occorrono in almeno una delle liste ordinate OLista1 e OLista2. Ad esempio sottoponendo il goal ?- ounion([2,5,5,6,10],[1,4,12], U), si otterrà la risposta U=[1,2,4,5,6,10,12].

Esercizio 7.18. Si consideri liste ordinate (in ordine crescente) di interi. Scrivere un programma Prolog in cui si definisca il predicato

ointer(+OListai1,+OLista2,?OLista).

Il predicato deve essere vero quando OLista è la lista ordinata e senza ripetizioni degli elementi che occorrono in entrambe le liste ordinate OLista1 e OLista2. Ad esempio sottoponendo il goal ?- ointer([2,5,5,6,10],[1,5,10,12], U), si otterrà la risposta U=[5,10].

Esercizio 7.19. Si consideri liste ordinate (in ordine crescente) di interi. Scrivere un programma Prolog in cui si definisca il predicato

osimdif(+OListai1,+OLista2,?OLista).

Il predicato deve essere vero quando OLista è la lista ordinata e senza ripetizioni degli elementi che occorrono in una ed una sola delle liste ordinate OLista1 e OLista2. Ad esempio sottoponendo il goal ?- osimdif([2,5,5,6,10],[1,5,10,13], U), si otterrà la risposta U=[1,2,6,13].

Esercizio 7.20. Scrivere un programma Prolog in cui si definisca il predicato

forma(+Termine1,?Termine2).

Il predicato deve essere vero quando il termine Termine2 si può ottenere dal termine Termine1, sostituendo il simbolo f ad ogni simbolo di funzione e il simbolo a ad ogni costante o variabile. Ad esempio sottoponendo il goal ?- forma(h(b,h(1,2),g(1,a,X)), T), si otterrà la risposta T=f(a, f(a,a), f(a,a,a)).

Esercizio 7.21. Scrivere un programma Prolog in cui si definisca il predicato

`alberello(+N).`

Il predicato ha sempre successo e se `N` è intero positivo, stampa un "albero di natale" alto `N`. Ad esempio sottoponendo il goal `?- alberello(6)`, si produrrà la stampa della figura:

```
     0
    000
   00000
  0000000
 000000000
00000000000
    I I
```

Esercizio 7.22. Scrivere un programma Prolog in cui si definisca il predicato

`diamante(+N).`

Il predicato ha sempre successo e se `N` è intero positivo dispari, stampa un "diamante" alto `N`. Ad esempio sottoponendo il goal `?- diamante(5)`, si produrrà la stampa della figura:

```
  0
 000
00000
 000
  0
```

Esercizio 7.23. Scegliere una possibile rappresentazione in Prolog della struttura dati albero. Si assuma che ogni nodo interno possa avere un numero arbitrario di figli. Basandosi sulla rappresentazione scelta, scrivere un programma Prolog che effettui una visita in post-ordine di un albero.

Esercizio 7.24. Si consideri il generico polinomio di grado $n > 2$ nella variabile x, a coefficienti interi:

$$p(x) \quad \equiv \quad a_n x^n + a_{n-1}x^{n-1} + a_{n-2}x^{n-2} + \cdots + a_2x^2 + a_1x + a_0.$$

Scegliere una possibile rappresentazione in Prolog di un polinomio di questo tipo. Si assuma che la variabile sia sempre x, ma che il grado n non sia fissato a priori.

Scrivere un programma Prolog che dato un polinomio $p(x)$, a coefficienti interi (nella rappresentazione descritta al punto precedente), e due numeri interi I e J (assumendo $I \leq J$) abbia successo se esiste (almeno) una radice intera di $p(x)$ compresa tra I e J, o fallisca in caso contrario.

CAPITOLO 8

Programmi generali

Nei capitoli precedenti abbiamo studiato una classe particolare di programmi logici, i programmi definiti. Tali programmi hanno la particolarità di trattare solamente conoscenza *positiva*. Come abbiamo visto infatti, asserire una clausola definita corrisponde ad affermare che qualora siano soddisfatte delle premesse allora devono essere vere delle conseguenze. Nonostante la Turing completezza dei programmi definiti, è semplice immaginare delle situazioni in cui le limitazioni espressive imposte dall'impiego di clausole definite risultano troppo restrittive.

Ad esempio supponiamo di avere a disposizione due predicati `studente` e `sposato` che modellano le proprietà di una persona di "essere studente" e di "essere sposato":

```
studente(mark).
studente(bill).
sposato(joe).
sposato(mark).
sposato(bob).
```

Vogliamo ora definire un predicato che caratterizzi le persone che sono studenti non sposati. Nel definire il predicato `studente_single` risulta utile disporre di una forma di negazione dei letterali occorrenti in un goal. In tal modo potremmo scrivere:

```
studente_single(X) :- studente(X), not sposato(X).
```

Un ulteriore esempio. Supponiamo di voler definire un predicato binario che sia soddisfatto quando i suoi due argomenti sono due liste che non hanno elementi in comune. Potremmo risolvere il problema definendo i seguenti predicati:

```
interseca(X,Y) :- member(Z,X), member(Z,Y).
disgiunte(X,Y) :- not interseca(X,Y).
```

Questi due programmi non sono programmi definiti. Infatti nelle clausole sopra riportate occorrono più di un letterale positivo. Ci si può rendere facilmente conto di ciò riscrivendo le clausole contenenti la negazione sotto forma di disgiunzioni. Ad esempio, così facendo, la clausola Prolog

```
studente_single(X) :- studente(X), not sposato(X).
```

appare come:

$$\texttt{studente_single(X)} \vee \neg\texttt{studente(X)} \vee \texttt{sposato(X)}$$

ove occorrono due letterali positivi.

In quanto segue studieremo una classe di programmi logici più generali dei programmi definiti. La seguente definizione la caratterizza.

Definizione 8.1. Un *programma generale* è un programma in cui sono ammessi letterali negativi nel corpo delle clausole. Un *goal generale* è un goal in cui sono ammessi letterali negativi.

Come vedremo, l'ammettere la negazione nel corpo delle clausole distrugge molte delle proprietà dei programmi studiate nei capitoli precedenti (in particolare nel Capitolo 6). In particolare,

(1) Le clausole, chiaramente, non sono più clausole di Horn. Come abbiamo menzionato, infatti, una clausola della forma
$$p(a) \leftarrow \neg q(a)$$
è equivalente a $p(a) \vee q(a)$.
(2) Un programma comunque ammette sempre un modello: l'insieme di tutti gli atomi ground $\mathcal{B}_P$, soddisfacendo tutte le teste, è ancora un modello.
(3) L'intersezione di modelli non è necessariamente un modello. Ad esempio, la clausola precedente possiede i due modelli $\{p(a)\}$ e $\{q(a)\}$; tuttavia la loro intersezione non è un modello.
(4) Può non esistere un unico modello minimo; in generale possono esistere più modelli minimali (si veda l'esempio dei due modelli sopra riportato).
(5) Si può definire anche per i programmi generali un operatore T_P. Il modo naturale per fare ciò è:
$$\begin{array}{rcll} T_P(I) & = & \{a : & a \leftarrow b_1, \ldots, b_m, \neg c_1, \ldots, \neg c_n \in ground(P) \\ & & & b_1 \in I, \ldots, b_m \in I, \\ & & & c_1 \notin I, \ldots, c_n \notin I\} \end{array}$$
Tuttavia, si osservi che $T_P(\emptyset) = \{p(a)\}$ e $T_P(\{q(a)\}) = \emptyset$. Pertanto T_P non è in generale nemmeno monotono. Di conseguenza non possiamo sfruttare il Teorema di punto fisso (Tarski) come accade per i programmi definiti.
(6) La SLD-risoluzione non è una procedura adeguata ai programmi generali. Si consideri ad esempio il programma:

```
p(a) :- p(b)
p(a) :- not p(b)
p(b) :- p(a)
```

(ovvero: $p(a) \vee \neg p(b)$)
(ovvero: $p(a) \vee p(b)$)
(ovvero: $\neg p(a) \vee p(b)$)

congiuntamente al goal

```
?- p(a), p(b)
```

(ovvero: $\neg p(a) \vee \neg p(b)$).

L'insieme di queste quattro clausole è insoddisfacibile (si veda la Sezione 2.3 del Capitolo 6). Come illustrato nel Capitolo 6 per costruire una derivazione del goal vuoto a partire da queste quattro clausole è necessario disporre della procedura di risoluzione (alla Robinson, ad esempio) più potente della SLD-risoluzione.

Siamo quindi in presenza di una classe di programmi per i quali non valgono i risultati di equivalenza tra le semantiche descritte nel Capitolo 6. Inoltre non disponiamo di una procedura risolutiva completa. Vedremo nelle prossime sezioni alcuni approcci mirati a risolvere (almeno in parte) questi problemi. Vedremo come siano state proposte delle semantiche operazionali per programmi generali. Nel Capitolo 12 studieremo un approccio alternativo alla programmazione dichiarativa. In tale occasione forniremo un criterio per individuare dei

modelli preferenziali per programmi generali (modelli stabili) e un metodo bottom-up per calcolarli.

1. Semantica operazionale della negazione

Vi sono principalmente tre proposte operazionali per il trattamento di programmi (e goal) generali che evitano di ricorrere alla procedura di risoluzione nella sua forma completa. Queste proposte sono:

- la *Closed World Assumption*,
- la *Negation as (finite) failure*, e
- la *regola di Herbrand.*

Per semplicità, nel descrivere questi approcci focalizzeremo il trattamento al caso di un programma definito P interrogato tramite un goal generale. Inoltre tratteremo il caso in cui A è ground.

1.1. Closed World Assumption. Il punto di partenza è la domanda:

"Quando si deve fornire risposta positiva ad un goal della forma: $\leftarrow \neg A$?"

La regola Closed World Assumption (in breve, CWA) prevede di rispondere *yes* quando non esiste alcuna SLD-derivazione per il goal $\leftarrow A$ dal programma P. Ovvero, per i risultati di equivalenza studiati nel Capitolo 6, quando vale che

$$A \notin T_P \uparrow \omega.$$

Questo primo approccio, introdotto da Reiter [**Rei78**] nel contesto delle basi di dati deduttive, non è però applicabile ai programmi Prolog, in quanto la proprietà su cui è basato si dimostra essere semidecidibile (ovvero non esiste un algoritmo che, dato un qualsiasi P e un qualsiasi A, sia in grado di stabilire sempre se valga $A \notin T_P \uparrow \omega$ oppure se valga $A \in T_P \uparrow \omega$. Nel caso valga $A \notin T_P \uparrow \omega$ infatti la ricerca della risposta potrebbe richiedere un tempo infinito).

1.2. Negation as Failure. La regola di Negazione per fallimento finito (in breve, NaF) è stata introdotta da Clark [**Cla78**]. Essa stabilisce di rispondere *yes* al goal $\leftarrow \neg A$ quando l'SLD-albero per $\leftarrow A$ è un albero di fallimento finito.

La regola NaF è chiaramente una approssimazione della CWA. Infatti consideriamo un programma P e un atomo ground A. Utilizzando CWA procederemmo costruendo l'SLD-albero per $\leftarrow A$. Possono verificarsi tre possibilità:

- esiste una SLD-derivazione di successo. In questo caso abbiamo dimostrato A e quindi non possiamo inferire $\neg A$.
- Esiste un SLD-albero di fallimento finito per $\leftarrow A$. In questo caso possiamo inferire $\neg A$ perchè siamo sicuri che non sia possibile dimostrare A.
- Non troviamo una SLD-derivazione di successo e l'SLD-albero è infinito. In questo caso non possiamo ottenere una risposta in tempo finito, perchè prima di concludere che vale $\neg A$ dovremmo visitare tutto l'albero.

La regola NaF prevede di rispondere negativamente al goal $\leftarrow \neg A$ quando si verifica il primo caso e affermativamente al goal $\leftarrow \neg A$ quando si verifica il secondo caso. Tuttavia nel terzo caso non può essere data alcuna risposta.

Chiariamo meglio le differenze tra CWA e la regola NaF con degli esempi:

ESEMPIO 8.1. Consideriamo il programma definito

```
p(a).
p(b).
q(a).
r(X) :- p(X), q(X).
```

congiuntamente al goal `?- not r(b).`

- La regola CWA risponde *yes*, perché $r(b)$ non appartiene al modello minimo del programma che è $M_P = \{p(a), p(b), q(a), r(a)\}$.
- NaF opera come segue: viene iniziata una computazione *ausiliaria* per il goal `?- r(b)` (si veda Figura 8.1). Tale computazione ausiliaria genera un SLD-albero finito e fornisce la risposta *no*. Quindi la risposta al goal generale iniziale sarà *yes*.

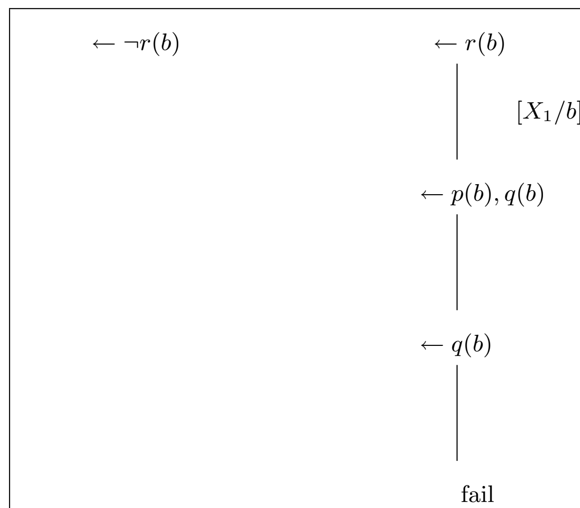


FIGURA 8.1. Derivazione tramite NaF del goal `?- not r(b).`

ESEMPIO 8.2. Consideriamo ora il programma:

```
q(X) :- q(s(X)).
p(X) :- q(0).
```

ed il goal `?- not p(0).`

- La risposta ottenuta utilizzando CWA è *yes*, dato che $M_P = \emptyset$.
- La regola NaF invece prevede di verificare se il goal ausiliario `?- p(0)` sia dimostrabile. La computazione ausiliaria tuttavia risulta infinita (si veda Figura 8.2). Pertanto NaF non risponderà nulla in questo caso.

La procedura risolutiva impiegata per implementare la NaF è chiamata SLDNF. Essa estende la SLD-risoluzione con la negazione per fallimento finito. SLDNF opera nel seguente modo:

DEFINIZIONE 8.2. Sia $G = \leftarrow L_1, \ldots, L_n$ un goal generale. Sia L_i il letterale selezionato.

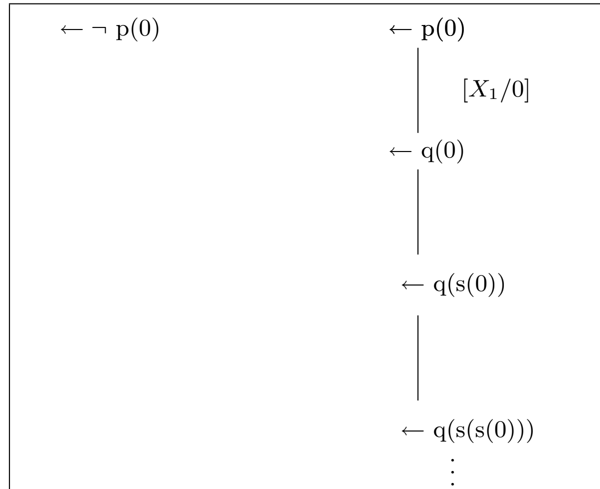


FIGURA 8.2. Derivazione tramite NaF del goal `?- not p(0).`

(1) Se il letterale selezionato L_i è positivo, allora si esegue un passo standard di *SLD*-risoluzione (Definizione 5.2);
(2) se il letterale selezionato L_i è negativo (ovvero è della forma $\neg A$) e l'atomo A è ground, allora si inizi una SLDNF-derivazione ausiliaria per il goal $\leftarrow A$. Inoltre,
   - se tale computazione termina e fornisce una risposta affermativa, allora la derivazione del goal G è dichiarata derivazione di fallimento.
   - Se la computazione ausiliaria termina e fornisce risposta negativa (si ottiene cioè un albero di fallimento finito), allora il goal risolvente di G è

$$G' = \leftarrow L_1, \ldots, L_{i-1}, L_{i+1}, \ldots, L_n.$$

Analizziamo ora come l'impiego della negazione e della procedura SLDNF renda il comportamento dell'interprete non in sintonia rispetto alla semantica puramente logica delle clausole.

ESEMPIO 8.3. Il seguente programma P

```
p(a) :- not q(a)
```

Esso è logicamente equivalente alla clausola $p(a) \vee q(a)$. Consideriamo i seguenti goal

(1) $\leftarrow p(a)$. Risolvendo con l'unica clausola del programma si produce il goal $\leftarrow \neg q(a)$. A questo punto viene iniziata una computazione ausiliaria per il goal $\leftarrow q(a)$. Tale computazione è di fallimento finito. Dunque la risposta finale sarà *yes*.
(2) $\leftarrow \neg p(a)$. Viene iniziata una computazione ausiliaria per il goal $\leftarrow p(a)$. Tale computazione, come visto al punto precedente, fornisce risposta *yes*. Quindi la risposta finale sarà *no*;
(3) $\leftarrow q(a)$. La risposta è *no*. Non ci sono infatti clausole la cui testa unifichi con l'unico atomo del goal.
(4) $\leftarrow \neg q(a)$. Viene iniziata una computazione ausiliaria per il goal $\leftarrow q(a)$. Da quanto detto al punto precedente, questa computazione fornisce la risposta *no*. Pertanto la risposta finale sarà *yes*.

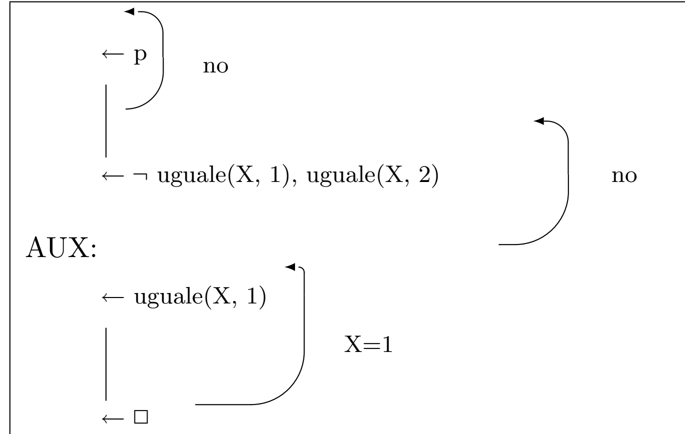


FIGURA 8.3. SLDNF-derivazione per P_1 e il goal `?- p`.

L'insieme $Oss(P)$ risulta quindi essere l'insieme $\{p(a)\}$ oppure, usando una nozione più estesa che menziona anche i letterali negativi veri: $\{p(a), \neg q(a)\}$.
Si noti una discordanza tra la completa simmetria presente nel significato logico della clausola (la disgiunzione $p(a) \vee q(a)$ è ovviamente logicamente equivalente a $q(a) \vee p(a)$), e la asimmetria presente nella semantica osservazionale. L'insieme $Oss(P)$ esprime infatti un particolare modello minimale di P, che non è il modello minimo (che non esiste).

Illustriamo ora il motivo di una restrizione che abbiamo imposto sul goal generale fin dall'inizio. Ovvero perchè nella definizione di SLDNF-derivazione si impone che l'atomo A sia ground. Un esempio chiarificatore.

ESEMPIO 8.4. Si considerino le quattro clausole

```
(1)   uguale(X,X).
(2)   p :- not uguale(X,1), uguale(X,2).
(3)   p :- uguale(X,2), not uguale(X,1).
(4)   q :- not p
```

E consideriamo i programmi così definiti:

$$P_1 = \{(1), (2), (4)\}$$

e

$$P_2 = \{(1), (3), (4)\}$$

Questi due programmi sono logicamente equivalenti. Tuttavia analizziamo cosa accade cercando di dimostrare il goal `?- p` nei due casi.

- Programma $P_1 = \{(1), (2), (4)\}$. Una SLDNF-derivazione per il goal `?- p` è riportata in Figura 8.3.
- Programma $P_2 = \{(1), (3), (4)\}$. Una SLDNF-derivazione per il goal `?- p` è riportata in Figura 8.4.

Si osservi che l'aver utilizzato la procedura di SLDNF per trattare un goal non ground ha causato la perdita di una soluzione. Questo esempio dimostra che, qualora si rilassi il vincolo che i letterali negativi del goal siano ground, la procedura di SLDNF-risoluzione diventa una procedura *incompleta*.

Se consideriamo invece il goal `?- q` otteniamo che, sempre nel caso in cui il passo SLDNF-derivazione sia applicato anche a letterali non ground, la risposta fornita non è quella attesa. Pertanto si ha che la SLDNF-risoluzione, in questo caso, è una procedura *scorretta*.

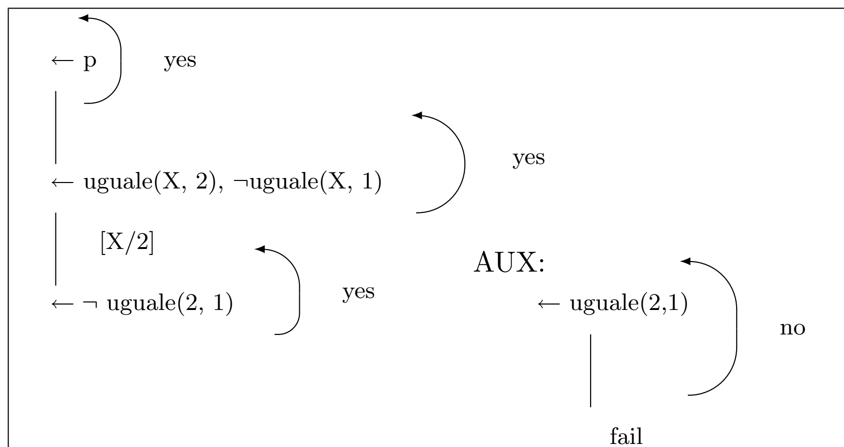


FIGURA 8.4. SLDNF-derivazione per P_2 e il goal `?- p.`

Vi sono in letteratura molti studi mirati a identificare condizioni che garantiscano che la SLDNF-risoluzione sia corretta e completa. Ne forniamo qui una sufficiente ma non necessaria:

> *Il letterale negato deve essere ground e nei goal prodotti costruendo l'albero ausiliario non devono occorrere altre negazioni.*

Come buona norma di programmazione si deve quindi scrivere programmi tali che quando si impone di dimostrare un letterale negato, tutte le variabili presenti in esso siano già state rese ground da qualche istanziazione.

DEFINIZIONE 8.3. Dato un programma P e un goal generale G, diremo che la computazione per il goal G *flounders* se ad un certo punto della computazione si genera un goal che contiene solamente letterali negativi non ground.

Per quanto detto, se una computazione *flounders* allora, in generale, non siamo in grado di procedere nella derivazione neanche utilizzando SLDNF.[1]

1.3. Regola di Herbrand. Un terzo approccio alla negazione è noto come Regola di Herbrand. Tale regola computazionale prescrive di inferire il goal $\leftarrow \neg A$ da un programma P quando A è falso in tutti i modelli di Herbrand del completamento $Comp(P)$ di P.

Dobbiamo quindi introdurre il completamento $Comp(P)$ di un programma. Vediamolo in un caso specifico. Sia P il programma:

[1]Una traduzione approssimativa che renda in italiano l'idea di *flounders* potrebbe essere *barcolla* o *comportarsi maldestramente* o ancora *impantanarsi*.

```
r(a, c).
p(a).
p(b).
q(X) :- p(X), r(X).
q(X) :- s(X).
```

Trasformiamolo innanzitutto in una forma *normalizzata*, indicata con $norm(P)$. La forma normalizzata si ottiene trasformando le clausole in modo che tutti gli argomenti della testa siano variabili. Per far ciò si introducono nuovi atomi (delle uguaglianze) nei corpi delle clausole.

```
r(X1,X2) :- X1=a, X2=c.
p(X1) :- X1=a.
p(X1) :- X1=b.
q(X1) :- p(X1), r(X1,X2).
q(X1) :- s(X1).
```

L'aver utilizzato uniformemente delle variabili nuove ci permette di raccogliere a fattore comune le teste uguali. Così facendo si costruisce a partire da $norm(P)$ la teoria $\mathit{iff}(P)$ nel modo seguente:

$$\begin{array}{l} \mathtt{r(X_1,X_2)} \leftrightarrow \mathtt{(X_1=a} \wedge \mathtt{X_2=c)} \\ \mathtt{p(X_1)} \leftrightarrow \mathtt{(X_1=a)} \vee \mathtt{(X_1=b)} \\ \mathtt{q(X_1)} \leftrightarrow (\exists\ \mathtt{Y_1(p(X_1),\ r(X_1,Y_1)))} \vee \mathtt{s(X_1)} \\ \mathtt{s(X_1)} \leftrightarrow \mathsf{false} \end{array}$$

Dato che il predicato `s` (usato nella definizione di `q`) non è definito (ovvero, non figura in alcuna delle teste delle clausole del programma iniziale), tramite l'ultima formula lo si forza ad essere falso. Si noti inoltre che la variabile Y_1, che occorre nel corpo ma non nella testa della quarta clausola, viene quantificata esistenzialmente in $\mathit{iff}(P)$. Il completamento di P è definito come:

$$Comp(P) = \mathit{iff}(P) \wedge \mathbf{(F1)} \wedge \mathbf{(F2)} \wedge \mathbf{(F3)}$$

dove **(F1)**, **(F2)** e **(F3)** sono gli assiomi della *Clark's Equality Theory* (vedi Definizione 4.10).

2. Confronti tra le tre regole

Abbiamo visto che regola CWA coinvolge un problema semi-decidibile, pertanto non è praticamente applicabile. Anche la regola di Herbrand, essendo basata sui modelli del completamento, non appare facilmente automatizzabile. Tuttavia il seguente teorema giustifica l'introduzione delle tre regole e ne individua le reciproche relazioni relativamente alla semaptica di punto fisso.

Teorema 8.1. *Sia P un programma definito e A un atomo di $\mathcal{B}_P$. Allora:*

(1) *$\neg A$ è inferibile tramite CWA se e solo se $A \in \mathcal{B}_P \setminus T_P \uparrow \omega$*
(2) *$\neg A$ è inferibile dalla regola di Herbrand se e solo se $A \in \mathcal{B}_P \setminus gfp(T_P)$.*
(3) *$\neg A$ è inferibile tramite NaF se e solo se $A \in \mathcal{B}_P \setminus T_P \downarrow \omega$.*

Sappiamo che $gfp(T_P) = T_P \downarrow \alpha$ per qualche ordinale $\alpha \geq \omega$ (quindi, un ordinale non necessariamente finito). Sappiamo inoltre che $lfp(T_P) = T_P \uparrow \omega \subseteq gfp(T_P)$. Quindi gli insiemi di atomi ground inferibili dalle tre regole sono inclusi l'uno nell'altro: se $\neg A$ è inferibile tramite NaF allora è inferibile dalla regola di Herbrand (ma non è detto il viceversa);

se $\neg A$ è inferibile tramite la regola di Herbrand allora è inferibile tramite CWA (ma non è detto il viceversa).

Esercizio 8.1. Si verifichi sul seguente programma definito, che i tre insiemi caratterizzati dal Teorema 8.1 sono inclusi strettamente l'uno nell'altro.

```
p(f(X)) :- p(X)
q(a) :- p(X)
q(f(X)) :- q(f(X))
r(a).
```

3. Negazione costruttiva

Abbiamo visto che goal contenenti letterali negativi non ground non possono essere gestiti dalla NaF. Vediamo se è possibile oltrepassare questo vincolo. Sia dato il programma

```
p(a).
p(b).
p(c).
q(a).
q(c).
r(X) :- p(X), q(X).
```

Come potremmo comportarci per rispondere al goal `?- not r(X)` ? Iniziamo col costruire l'SLD-albero per il goal `?-r(X)` (vedi Figura 8.5).

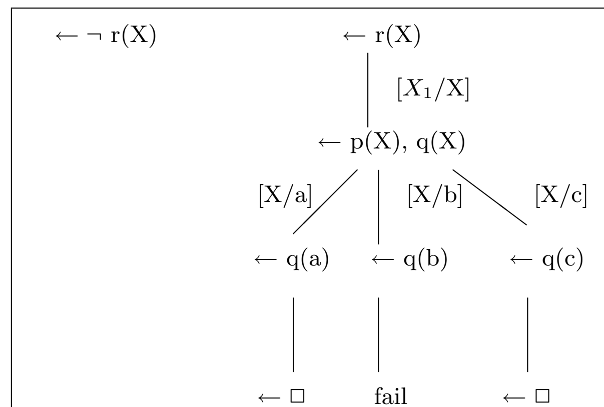


Figura 8.5. Derivazione per i goal `?- not r(X)` e `?- r(X)`

Dato che l'SLD-albero per `?- r(X)` è finito, esso contiene tutte le risposte calcolate (esse sono $X = a$ e $X = c$). Prendendole in considerazione tutte possiamo scrivere che

$$r(X) \leftrightarrow (X = a) \vee (X = c)$$

Ciò ci suggerisce che potremmo complementare tale disgiunzione per fornire una risposta al goal `?- not r(X)`. Il risultato sarebbe quindi

$$yes, (X \neq a) \wedge (X \neq c).$$

Sembra un metodo applicabile, a patto di ammettere (assieme alle sostituzioni) l'impiego di disuguaglianze. Ció al fine di indicare le *non-soluzioni*, ovvero dei vincoli che escludono dei valori. Dobbiamo pertanto modificare la definizione di risposta calcolata.

Tuttavia il precedente era un esempio, in un certo senso, fortunato. Consideriamo infatti la singola clausola:

```
p(X,Y) :- X=f(Z), Y=g(Z).
```

Abbiamo visto che è una scrittura che indica l'enunciato

$$\forall X, Y \Big(p(X, Y) \leftrightarrow \exists Z \, (X = f(Z) \wedge Y = g(Z) \Big)$$

Per applicare la idea sopra illustrata, nel rispondere al goal `?- not p(X,Y)`, dobbiamo considerare l'SLD-albero relativo al goal `?- p(X,Y)`. Questo è finito e presenta una sola risposta calcolata: $[X/f(Z), Y/g(Z)]$. Agendo come prima, complementando, otterremmoa che

$$\neg p(X, Y) \leftrightarrow \forall Z \, (X \neq f(Z) \vee Y \neq g(Z))$$

La formula di destra, in questo caso, può essere riscritta come

$$\forall X \, (X \neq f(Z)) \;\vee\; \exists Z \, (X = f(Z) \wedge \forall Z \, (Y \neq g(Z)))$$

fornendo la risposta al goal `?- not p(X,Y)`.

Si nota quindi che le risposte ad un goal generale non ground potrebbero coinvolgere anche formule universalmente quantificate.

Concludiamo facendo presente il fatto che la negazione costruttiva può essere effettivamente implementata. Tuttavia, anche in questo caso sussistono delle assunzioni (ad esempio, la finitezza degli SLD-alberi) necessarie al fine di ottenere risposte in tempo finito. Inoltre, come suggeriscono i precedenti esempi, le risposte che si ottengono sono spesso troppo implicite per essere di aiuto all'utente. Nei Prolog commerciali allo stato attuale non si trovano implementazioni della negazione costruttiva. Per approfondimenti, si veda [**Stu95, DPR00**].

NOTA 8.1. Si noti che nei moderni interpreti Prolog la negazione non viene indicata dal simbolo `not` ma dal simbolo `\+`. Solitamente l'uso del simbolo `not` viene disincentivato ed è previsto solamente per retro-compatibilità rispetto a vecchie implementazioni di Prolog.

4. Implementazione della NaF

In Prolog, la negazione viene implementata nel seguente modo. In presenza di un goal della forma

```
?- not P.
```

si agisce come se nel programma fossero presenti le clausole:

```
not(P) :- P, !, fail.
not(P).
```

NOTA 8.2. Si noti l'impiego della meta-variable facility nella prima delle due precedenti clausole. Ricordiamo che tramite questa funzionalità offerta da Prolog, possiamo utilizzare un termine come se fosse un atomo (ovviamente, rispettando le restrizioni sulla sintassi degli atomi). In questo caso `P` occorre con due ruoli diversi: come termine (argomento di `not`) e come atomo. Inoltre si noti che viene utilizzato un CUT rosso.

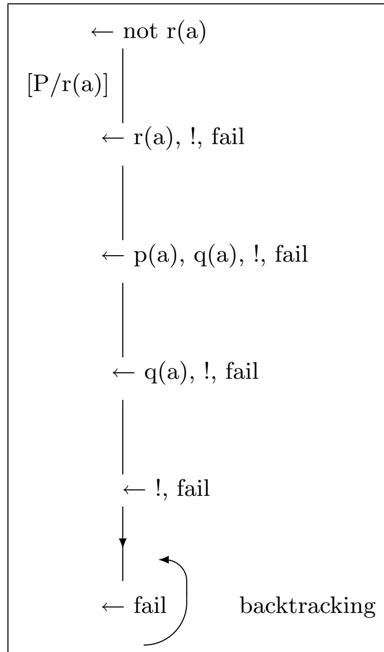


FIGURA 8.6. Derivazione per il goal `?- not r(a).`

Nelle due clausole precedenti abbiamo impiegato le parentesi, ad esempio in `not(P)`, per facilitare il lettore. Tale impiego è tuttavia superfluo in quanto molti Prolog definiscono autonomamente l'operatore `not` come prefisso, come se fosse implicitamente eseguita la direttiva:[2]

```
:- op(900, fy, not).
```

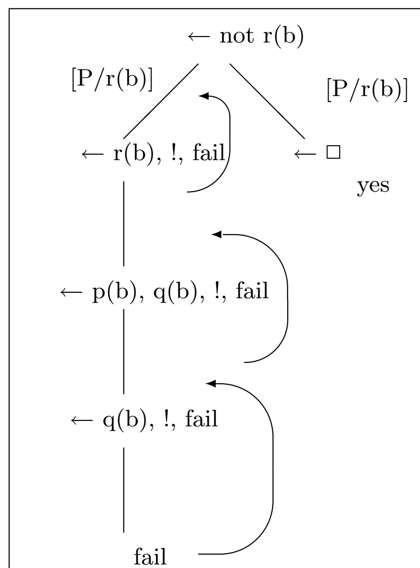


FIGURA 8.7. Derivazione per il goal `?- not r(b).`

[2]Ovviamente, per gli interpreti Prolog che utilizzano `\+` in luogo di `not` avremo `:- op(900, fy, \+).`

Vediamo tramite due esempi come questa implementazione della negazione possa funzionare: Consideriamo il programma

```
p(a).
p(b).
q(a).
r(X) :- p(X), q(X).
```

Sottoponiamo all'interprete Prolog due diversi goal:

- Il goal `?- not r(a)`. L'idea di base è che appena viene trovata una soluzione per `?- r(a)` si risponde *no*. L'ottenimento di questa risposta è causato dalla combinazione di `fail` e CUT. Infatti si giunge al predicato `fail` solo dopo aver ottenuto una dimostrazione dell'atomo `r(a)`, il primo atomo nel corpo della clausola

  ```
  not(P) :- P, !, fail.
  ```

  utilizzata per *riscrivere* il `not`. A questo punto il predicato `fail` innescherebbe il backtracking alla ricerca di una altra soluzione. Tuttavia, subito prima del `fail` è stato eseguito un CUT che impedisce il backtracking. Quindi la risposta sarà *no*. Si veda Figura 8.6.
- Si noti che se nel caso precedente non fosse stato possibile trovare una soluzione a `?- r(a)`, si sarebbe verificato un fallimento nel tentativo di risolvere il primo atomo del corpo della clausola `not(P) :- P, !, fail.`. Conseguentemente, la seconda clausola (ovvero il fatto `not(P).`) sarebbe entrata in gioco e avrebbe generato una risposta positiva. Ciò accade con il goal `?- not r(b)`, come si vede in Figura 8.7.

Esercizio 8.2. Applicare questo modo di implementare la negazione alle derivazioni presentate in questo capitolo che soffrono del problema del *floundering* (Definizione 8.3), quelle cioè che presentano goal con soli letterali negati e non ground.

5. Esercizi

Esercizio 8.3. Si scelga un numero intero $n \geq 3$. Si scriva un programma definito che abbia almeno n modelli distinti e si indichino tali modelli.

Esercizio 8.4. Si scelga un numero intero $n \geq 3$. Si scriva un programma Prolog che abbia esattamente n modelli minimali distinti. Si indichino tali modelli.

CAPITOLO 9

Programmazione dichiarativa

Per programmazione dichiarativa si intende una metodologia di programmazione il più vicina possibile alla definizione delle specifiche del problema che si vuole risolvere. In questo capitolo si forniranno alcuni principi di base e si illustreranno alcuni programmi costruiti allo scopo di risolvere specifici problemi, nel modo più dichiarativo possibile. Per approfondimenti e ulteriori esempi si veda ad esempio [**SS97**].

Oltre a porre il programmatore ad un livello di astrazione superiore, programmare in modo dichiarativo permette di aumentare chiarezza e leggibilità dei programmi. Nel caso della programmazione in Prolog, la particolare concisione solitamente permette di scrivere un programma in unico file. Questo fa sì che alcuni problemi legati alla non composizionalità della semantica di Prolog non emergano.[1]

Diamo in queste pagine dei principi generali che possono essere di aiuto nello scrivere programmi dichiarativi migliori.

Una prima regola di buona programmazione è: scegliere dei buoni *nomi* per i

- simboli predicativi,
- simboli di funzione,
- variabili

Per quanto riguarda i simboli predicativi, è buona norma cercare di assegnare dei nomi che ricordino il significato dei predicati che si definiscono (le proprietà logiche che essi codificano), non lo scopo o il ruolo che questi rivestono nel risolvere un particolare problema.

Malgrado un buon programma Prolog sia solitamente di facile lettura, i commenti sono sempre benvenuti. In particolare, sarebbe auspicabile l'inserimento (solitamente subito prima del gruppo di clausole che definiscono un predicato), come commenti, delle seguenti specifiche relative al predicato che si sta definendo:

- predicate p(T1,...,Tn)
- type T1: (ad esempio: "lista di variabili" o "intero" o ...)

  ⋮
- type Tn: ...
- Significato della procedura in corso di definizione, proprietà che rende soddisfatta la/le clausole
- Modes: questo è un aspetto legato al modo in cui il programmatore pensa che un atomo (che unifica con p(T1,...,Tn)) debba venir risolto. Ad esempio, se ci

[1] Lo studio di queste problematiche va oltre gli scopi di questo corso, tuttavia, a titolo di esempio, si consideri i due programmi $P_1 = \{p \leftarrow q\}, P_2 = \{q\}$. Si ha che $M_{P_1} = \emptyset$ e $M_{P_2} = \{q\}$, mentre $M_{P_1 \cup P_2} = \{p, q\}$. La presenza di p nella semantica del programma complessivo appare alquanto inattesa se ci si limita a osservare separatamente le semantiche di P_1 e P_2.

si aspetta che al momento della unificazione dell'atomo con la testa p(T1,...,Tn) alcuni argomenti siano delle variabili e/o altri siano istanziati. Intuitivamente, ci si può attendere che gli argomenti richiesti essere variabili vengano successivamente istanziati quando si risolveranno gli atomi del corpo della clausola. Si può pensare a questi argomenti come degli output. Viceversa, richiedere che un argomento sia istanziato ad un termine ground ricorda i parametri di input della programmazione imperativa. Si tenga comunque presente che più il programma è scritto in modo dichiarativo, più viene persa la distinzione intuitiva tra argomenti di input e di output. Per indicare i modes vi è una notazione tipica:
  - + indica che l'argomento (al momento della risoluzione) è supposto essere un termine ground;
  - \- indica che l'argomento è supposto essere un termine non ground (come caso particolare, una variabile), un termine che quindi può subire successive istanziazioni;
  - ? sta per entrambi.

  Un mode del tipo $(+,+,?),(+,?,+)$ indica quindi che il predicato dovrebbe essere utilizzato con T1 e T2 istanziati oppure con T1 e T3 istanziati.
- Molteplicità: per ogni mode è bene indicare se ci si aspetta un comportamento deterministico (ovvero, al massimo una soluzione) o non-deterministico.

Il debugging di un programma può essere fatto per mezzo di tecniche tradizionali (tracer dell'interprete (tramite i comandi `trace`, o `spy`, o ...), comandi di `write` inseriti in vari punti del programma) o automatiche.

È buona norma usare una giusta commistione tra dichiaratività ed efficienza. Cercare la massima efficienza snaturando la leggibilità e la dichiaratività del programma non fa parte dei principi della programmazione dichiarativa. In tal caso è più proficuo utilizzare un diverso paradigma di programmazione. Vediamo un esempio in tale direzione. Consideriamo i due programmi equivalenti (ed egualmente dichiarativi):

P_1:
```
sumlist([],0).
sumlist([X|Xs],Sum) :- sumlist(Xs,S), Sum is S + X.
```

P_2:
```
sumlist([X|Xs],Sum) :- sumlist(Xs,S), Sum is S + X.
sumlist([],0).
```

Si può assumere che nella maggior parte dei casi questi predicati saranno utilizzati in presenza di una lista non vuota, come in `?- sumlist([`t_1`,...,`t_n`],L)`. Per goal di questo tipo il secondo programma risulta più efficiente in quanto nello sviluppare la derivazione non viene tentata l'unificazione dell'atomo selezionato con la testa della prima clausola. Si compiono pertanto n esecuzioni dell'algoritmo di unificazione in meno.

1. Esempi di programmazione ricorsiva

Diamo in questa sezione alcuni esempi di programmi Prolog che risolvono dei semplici problemi.

1.1. Problema della *massimo comune divisore*. Il seguente programma calcola il massimo comune divisore di due numeri naturali. Usiamo la rappresentazione dei numeri naturali introdotta nel Capitolo 3.

```
mcd(X,X,X) :- gt(X,0).
mcd(X,Y,MCD) :- gt(X,Y), plus(Y,X1,X), mcd(X1,Y,MCD).
mcd(X,Y,MCD) :- gt(Y,X), plus(X,Y1,Y), mcd(X,Y1,MCD).
```

dove `gt` è così definito:

```
gt(s(X),0) :- num(X).
gt(s(X),s(Y)) :- gt(X,Y).
```

Il predicato `mcd(X,Y,Z)` sarà vero se è possibile istanziare `X`, `Y`, e `Z` in modo che `Z` sia il massimo comune divisore di `X` e `Y`. Si noti l'utilizzo di `plus` in modo da calcolare la differenza invece che la somma.

1.2. Eliminazione dei doppioni da una lista. Vogliamo scrivere un predicato Prolog `elim_doppi(Lista,Insieme)` che sia vero se la lista `Insieme` è una lista che contiene tutti gli elementi contenuti dalla lista (ad esempio di numeri, ma non necessariamente) `Lista`, ma senza ripetizioni. Ecco una possibile implementazione:

```
elim_doppi(Xs,Ys):-elim_doppi_aux(Xs,[],Ys).

elim_doppi_aux([],Ys,Ys).
elim_doppi_aux([X|Xs],Acc,Ys) :- member(X,Acc),
                                 elim_doppi_aux(Xs,Acc,Ys).
elim_doppi_aux([X|Xs],Acc,Ys) :- nonmember(X, Acc),
                                 elim_doppi_aux(Xs,[X|Acc],Ys).
```

Si noti l'impiego dei una lista ausiliaria `Acc` utilizzata come "accumulatore" degli elementi che faranno parte della soluzione. Così facendo la lista `Insieme` conterrà gli elementi di `Lista` nell'ordine inverso.

Per completare il programma è necessario disporre delle definizioni dei predicati `member` e `nonmember`. Solitamente negli interpreti Prolog questi predicati (almeno il primo) sono predefiniti o, come in SICStus, sono disponibili consultando una libreria.[2] Comunque per comodità riportiamo di seguito delle possibili definizioni di questi due utili predicati:

```
member(X,[X|Xs]).
member(X,[Y|Ys]) :- member(X,Ys).

nonmember(X,[Y|Ys]) :- X\==Y, nonmember(X,Ys).
nonmember(X,[]).
```

Esercizio 9.1. Modificare il programma precedente in modo che la lista `Insieme` contenga gli elementi di `Lista`, sempre senza ripetizioni, ma nello stesso ordine in cui appaiono in `Lista`.

[2] In particolare la libreria di SICStus si consulta tramite la direttiva `:- use_module(library(lists)).`

1.3. Problema del *merge sort*. Il seguente programma implementa il merge sort: l'ordinamento di una lista effettuato spezzando la lista in due parti, ordinando separatamente le parti e infine fondendo le due parti ordinate.

```
merge_sort([],[]).
merge_sort([X],[X]).
merge_sort([D,P|Xs],Ys) :- split([D,P|Xs],Dispari,Pari),
                           merge_sort(Dispari,DispariOrdinati),
                           merge_sort(Pari,PariOrdinati),
                           ordered_merge(DispariOrdinati,PariOrdinati,Ys).
```

Il predicato `merge_sort(Xs,Ys)` sarà quindi vero se `Ys` è istanziabile alla versione ordinata della lista `Xs`.
Il predicato `split` ha il compito di spezzare in due la lista. In particolare `split(Lista,Ds,Ps)` è soddisfatto se `Ds` è la lista degli elementi che si trovano in posizione dispari in `Lista`, e similmente per `Ps`.

```
split([],[],[]).
split([X],[X],[]).
split([X,Y|Xs],[X|Dispari],[Y|Pari]) :- split(Xs,Pari,Dispari).
```

Il predicato `ordered_merge` compie la fusione di due lista ordinate. Più precisamente, `ordered_merge(Xs,Ys,Zs)` è soddisfatto se è possibile istanziare `Xs`, `Ys`, e `Zs` in modo che `Zs` sia la lista ordinata ottenibile fondendo le liste ordinate `Xs` e `Ys`.

```
ordered_merge([],Ys,Ys).
ordered_merge([X|Xs],[],[X|Xs]).
ordered_merge([X|Xs],[Y|Ys],[X|Zs]) :- X<Y, !, ordered_merge(Xs,[Y|Ys],Zs).
ordered_merge([X|Xs],[Y|Ys],[Y|Zs]) :- X>Y, !, ordered_merge([X|Xs],Ys,Zs).
ordered_merge([X|Xs],[Y|Ys],[X,Y|Zs]) :- X==Y, ordered_merge(Xs,Ys,Zs).
```

1.4. Un predicato per generalizzare una lista ground. Vogliamo scrivere un programma che definisca il predicato `to_hollow_list(+GLista,-Lista,+Constanti)`. Vogliamo che tale predicato sia vero quando `GLista` è una lista di elementi ground, `Constanti` è una lista di costanti (atomi o numeri), e `Lista` è la lista ottenibile da `GLista` sostituendo ad ogni elemento che appare in `Constanti` una nuova variabile. Ovviamente occorrenze diverse dello stesso elemento devono essere sostituite con occorrenze la stessa variabile. Si noti che le costanti che non figurano nella lista `Constanti` non saranno sostituite. Inoltre, il mode (+,-,+) che viene richiesto indica che si vuole un comportamento rispettoso delle specifiche solo nel caso in cui il primo e il terzo argomento siano ground, mentre il secondo è supposto essere una variabile. Ad esempio potremmo ottenere le seguenti risposte (modulo i nomi delle variabili nelle risposte, essi possono dipendere dallo specifico interprete Prolog utilizzato):

```
?- to_hollow_list([],Lista,[f,a,c,e]).
   yes   Lista = []
?- to_hollow_list([a,b,c,d,c,b],Lista,[f,a,c,e]).
   yes   Lista = [A1,b,A2,d,A2,b]
?- to_hollow_list([4,4,foo,6,5,boo,bar],Lista,[12,4,bar,5,8]).
   yes   Lista = [X1,X1,foo,6,X2,boo,X3]
```

Ecco una possibile definizione:

```
to_hollow_list(GLista,L,Costanti) :- length(GLista,N),
                                     length(L,N),
                                     unifica_vars(GLista,L,Costanti).

unifica_vars(GL,VL,Costanti) :- unifica_vars_aux(GL,VL,Costanti,[]).

unifica_vars_aux([],[],Costanti,Done).
unifica_vars_aux([C|R],[VC|VL],Costanti,Done) :-
                                member(C,Costanti),
                                nonmember(C,Done),
                                !,
                                unifica_var(C,VC,R,VL),
                                unifica_vars_aux(R,VL,Costanti,[C|Done]).
unifica_vars_aux([C|R],[VC|VL],Costanti,Done) :-
                                nonmember(C,Costanti),
                                !, C=VC,
                                unifica_vars_aux(R,VL,Costanti,Done).
unifica_vars_aux([C|R],[VC|VL],Costanti,Done) :-
                                unifica_vars_aux(R,VL,Costanti,[C|Done]).

unifica_var(C,V,[],[]).
unifica_var(C,V,[H|T],[VH|VT]) :- C==H, !, V=VH,
                                unifica_var(C,V,T,VT).
unifica_var(C,V,[H|T],[VH|VT]) :- unifica_var(C,V,T,VT).
```

L'idea base del programma consiste nel creare una lista di variabili nuove tutte distinte lunga quanto la lista di costanti `GLista`. Successivamente si scandisce ripetutamente la lista di variabili unificando le variabili che devono essere uguali. Gli elementi di `GLista` che non vanno rimpiazzati vengono inseriti nel risultato istanziando la variabile omologa. Si osservi l'impiego del predicato ausiliario `unifica_vars_aux` che gestisce l'argomento supplementare `Done`. Questo viene utilizzato per tenere traccia delle costanti già rimpiazzate, al fine di evitare scansioni superflue della lista di variabili.

Esercizio 9.2. Realizzare un programma che definisca un predicato

`to_hollow_term(+GTermine,-Termine,+Constanti)`

In analogia a `to_hollow_list` tale predicato deve essere vero quando `GTermine` è istanziato ad un termine ground, `Termine` è una variabile e `Constanti` è una lista di costanti (atomi o numeri). In tale situazione la variabile `Termine` viene istanziata ad un termine (non necessariamente ground) che ha la stessa struttura di `GTermine` ed è ottenibile da quest'ultimo sostituendo le costanti che appartengono a `Constanti` con delle variabili nuove. Anche in questo caso si vuole che occorrenze diverse dello stesso elemento devono essere sostituite con occorrenze la stessa variabile; inoltre le costanti che non figurano nella lista `Constanti` non devono essere sostituite. Ad esempio:

```
?- to_hollow_term(f(a,b,h(g(b,c,a),a)),Termine,[b,c,e]).
   yes   Termine = f(a,X1,h(g(X1,X2,a),a))
```

1.5. Problema delle *torri di Hanoi*. Il problema delle torri di Hanoi può essere formulato come segue. Sono dati N dischi che possono essere infilati in tre perni, A, B, C. I dischi sono tutti di diametro diverso e sono disposti in ordine decrescente sul perno A. Si deve disporre i dischi in ordine decrescente sul perno B. Per fare ciò è necessario spostare i dischi uno alla volta da un perno ad un altro (qualsiasi) facendo attenzione a non disporre mai un disco sopra ad uno più piccolo.[3]

Il seguente è un semplice programma Prolog che dichiarativamente definisce la soluzione del problema.

```
:-op(100,xfx,to).

hanoi(s(0),A,B,C,[A to B]).
hanoi(s(s(N)),A,B,C,Moves):- hanoi(s(N),A,C,B,M1),
                             hanoi(s(N),C,B,A,M2),
                             append(M1,[A to B|M2],Moves).
```

Si noti la definizione dell'operatore infisso `to` (si veda in merito la Sezione 11 del Capitolo 7). Il predicato `hanoi(N,A,C,B,Mosse)` è vero quando `Mosse` è la sequenza di mosse necessarie per spostare una torre di `N` dischi dal perno `A` al perno `B` utilizzando `C` come perno ausiliario. Usiamo anche in questo caso la rappresentazione dei numeri naturali introdotta nel Capitolo 3. Una mossa è rappresentata tramite l'operatore `to`: `X to Y` significa che il disco in cima al perno `X` viene spostato in cima al perno `Y`. Con queste premesse, leggiamo cosa è specificato dichiarativamente dalle clausole precedenti. La prima clausola asserisce che per spostare una torre di un solo disco dal perno `A` al perno `B` basta fare una mossa: `A to B`. La seconda clausola riduce il problema di spostare una pila di $N+2$ dischi dal perno `A` al perno `B` a tre sotto-problemi: spostare una pila di $N+1$ dischi da `A` a `C`; spostare una pila di 1 disco da `A` a `B`; ed infine spostare una pila di $N+1$ dischi da `C` a `B`.

2. Approccio *generate and test* alla soluzione di problemi

La metodologia *generate and test* è una metodologia ampiamente utilizzata in programmazione dichiarativa per cercare soluzioni ad un dato problema. Lo schema base è il seguente (dove $\bar{X}$ indica una sequenza di variabili):

```
find(X̄) :- genera(X̄),
           testa(X̄).
```

Il predicato `genera` è usualmente non-deterministico. Il suo scopo è quello di generare ad una ad una (tutte) le potenziali soluzioni. Ciò solitamente comporta visitare uno spazio

[3] Il problema delle torri di Hanoi deriva da una antica leggenda indiana. "Nel tempio di Brahma a Benares, su di un piatto di ottone, sotto la cupola che segna il centro del mondo, si trovano 64 dischi d'oro puro che i monaci spostano uno alla volta infilandoli in un ago di diamanti, seguendo l'immutabile legge di Brahma: nessun disco può essere posato su un altro più piccolo. All'inizio del mondo tutti i 64 dischi erano infilati in un ago e formavano la Torre di Brahma. Il processo di spostamento dei dischi da un ago all'altro è tuttora in corso. Quando l'ultimo disco sarà finalmente piazzato a formare di nuovo la Torre di Brahma in un ago diverso, allora giungerà la fine del mondo e tutto si trasformerà in polvere." Comunque, dato che per ultimare il loro compito i monaci dovranno compiere 2^{64} mosse, abbiamo ancora sufficiente tempo per finire il corso....

Più realisticamente, pare che questo gioco (e la relativa leggenda) sia stato inventato dal matematico francese Edouard Lucas ed appaia per la prima volta nel 1883.

delle soluzioni che, tipicamente ma non necessariamente, ha dimensioni esponenziali. Il predicato `testa` invece, è solitamente deterministico ed efficiente; ha il compito di verificare se una particolare soluzione candidata soddisfa determinati requisiti (e quindi è realmente una soluzione).

Il seguente esempio mostra una possibile implementazione della verifica di intersezione non vuota fra due liste adottando la metodologia generate and test.

```
interseca(X,Y) :-
     member(Z,X), %% generazione di una Z
     member(Z,Y). %% verifica se Z appartiene a Y.
```

Immaginiamo di sottoporre all'interprete Prolog il goal `?-interseca(`*lista1*`,`*lista2*`)`, con due liste (ground) come argomenti. Allora il primo predicato `member` seleziona un elemento dalla prima lista, mentre il secondo viene utilizzato per verificare se tale elemento appartiene alla seconda lista.

Il prossimo esempio mostra invece come sia possibile risolvere (in modo inefficiente) il problema dell'ordinamento di una lista con la metodologia generate and test.

```
ordina(X,Y) :-
     permutazione(X,Y), %% generazione delle possibili permutazioni
     ordinata(Y). %% verifica se la lista è ordinata
```

Ove i predicati `ordinata` e `permutazione` e l'ausiliario `seleziona` sono così definiti:

```
ordinata([X,Y|R]) :- X<Y,
                     ordinata([Y|R]).
ordinata([]).
ordinata([X]).

permutazione(X_s,[Z|Z_s]) :- seleziona(Z,X_s,Y_s),
                             permutazione(Y_s,Z_s).
permutazione([],[]).

seleziona(X,[Y|R],[Y|S]) :- seleziona(X,R,S).
seleziona(X,[X|R],R).
```

2.1. Il problema delle *N* regine. L'obiettivo del problema delle N-Regine è posizionare N regine su una scacchiera $N \times N$ in modo tale che nessuna regina attacchi un'altra (una regina attacca qualsiasi altra regina posizionata nella stessa riga, colonna o diagonale).

Ecco una formulazione generate and test del problema:

```
regine(N,Allocazione) :- genera(N,Allocazione),
                         safe(Allocazione).
```

Chiaramente `genera` ha lo scopo di generare piazzamenti delle N regine mentre `safe` verifica che in un dato piazzamento non vi siano due regine che possono attaccarsi.

Dobbiamo ora decidere come rappresentare le allocazioni delle N regine. Osserviamo che ogni allocazione si può rappresentare mediante una lista di interi. Ad esempio le due

configurazioni seguenti

<table>
<tr><td></td><td>*</td><td></td><td></td></tr>
<tr><td></td><td></td><td></td><td>*</td></tr>
<tr><td>*</td><td></td><td></td><td></td></tr>
<tr><td></td><td></td><td>*</td><td></td></tr>
</table>

<table>
<tr><td></td><td>*</td><td></td><td></td></tr>
<tr><td></td><td></td><td>*</td><td></td></tr>
<tr><td>*</td><td></td><td></td><td></td></tr>
<tr><td></td><td></td><td></td><td>*</td></tr>
</table>

saranno rappresentate, rispettivamente, dalle liste `[3,1,4,2]` e `[3,1,2,4]`. Si noti che la prima è una configurazione sicura, la seconda no.

Definiamo quindi il predicato *genera*:

```
genera(N,Allocazione) :- length(Allocazione,N),
                         valori_ammessi(N,ListaValori),
                         funzione(Allocazione,ListaValori).
```

Dove il predicato `length` viene usato, con N noto, al fine di generare una lista di N variabili (ogni variabile corrisponderà ad una colonna della scacchiera). Il predicato `valori_ammessi` genera invece una lista contenente gli elementi da 1 a N. Il predicato `funzione` genera una particolare allocazione scegliendo una casella (ovvero un numero dalla lista di elementi da 1 a N) in ogni colonna della scacchiera. Questi predicati sono così definiti:

```
valori_ammessi(N,[N|R]) :- N>0,
                           M is N-1,
                           valori_ammessi(M,R).
valori_ammessi(0,[]).

funzione([A|R],Cod) :- member(A,Cod),
                       funzione(R,Cod).
funzione([],Cod).
```

Resta da dichiarare il predicato `safe` che realizza la fase di testing. Ecco il suo codice unitamente ad alcuni predicati ausiliari:

```
safe([]).
safe([Q|Qs]) :- safe(Qs),
                not attacca(Q,Qs).

attacca(X,Xs) :- att(X,1,Xs).

att(X,Offset,[Y|R]) :- X is Y+Offset.
att(X,Offset,[X|R]).                      (*)
att(X,Offset,[Y|R]) :- X is Y-Offset.
att(X,Offset,[Y|R]) :- N1 is Offset+1,
                       att(X,N1,R).
```

Si osservi che l'uso della negazione è corretto perchè in quel punto i termini coinvolti saranno termini ground.

Analizziamo il problema così formulato dal punto di vista computazionale. La valutazione dell'atomo `genera(N,Allocazione)` genera N^N configurazioni possibili: saranno verificate

man mano che vengono generate fino a quando una di esse viene accettata. La valutazione dell'atomo `safe(Allocazione)` è invece deterministica ed ha un ordine di complessità quadratico.

Per velocizzare la ricerca della soluzione si deve cercare di ridurre lo spazio di ricerca. Una prima ottimizzazione possibile è quella di non generare tutte le N^N potenziali soluzioni ma solo le permutazioni della lista `[1,...,N]`. In tal modo si ottiene una complessità pari a $N!$. Il fatto $(*)$ utilizzato nel predicato `att` può così essere omesso.

```
genera(N,Allocazione) :- valori_ammessi(N,ListaValori),
                         permutazione(ListaValori,Allocazione).
```

Una migliore efficienza si ottiene però adottando una diversa formulazione del problema. Si noti infatti che una soluzione può essere costruita in modo *incrementale* ovvero collocando le regine una ad una e verificando che ogni volta che una regina viene collocata sulla scacchiera essa sia posizionata in una cella sicura (con questo approccio si mescola in un certo senso i passi della fase di generate e di quella di test).

```
regine(N,Allocazione) :- valori_ammessi(N,ListaValori),
                         aggiungi_regina(ListaValori,[],Allocazione).

aggiungi_regina(NonMesse,Sicure,Allocazione) :-
                seleziona(Q,NonMesse,Rimanenti),
                not attacca(Q,Sicure),
                aggiungi_regina(Rimanenti,[Q|Sicure],Allocazione).
aggiungi_regina([],Allocazione,Allocazione).
```

2.1.1. *Esempi di tempi di computazione.* Il programma appena descritto per la soluzione del problema delle regine è stato testato utilizzando i tre differenti implementazioni di Prolog: SICStus Prolog, ECLiPSe, e SWI-Prolog. La Tabella 1 riporta i risultati degli esperimenti effettuati su un Pentium4, 1.6GHz, sistema operativo Windows XP, 256MB Ram. I tempi sono espressi in millisecondi.

SICStus Prolog è stato utilizzato sia `consult`-ando il file che compilandolo tramite il comando `compile`. ECLiPSe non permette il consult e i test sono stati effettuati solo compilando il codice. Vi sono tuttavia a disposizione sia una modalità di compilazione che permette il debug (e che genera codice solitamente più lento) sia una ottimizzata che lo esclude (la scelta di quale modalità di compilazione attivare si compie impostando l'opzione `debug/nodebug`. Si veda per i dettagli la documentazione di ECLiPSe). Infine, SWI-Prolog consente solamente il consult.

In questi test il Prolog più veloce risulta essere SICStus in modalità compile. Prendendo questi tempi come termine di paragone, ECLiPSe risulta essere il secondo, con un fattore di circa 2 nella modalità "nodebug", e di circa 3 in modalità "debug". SWI-Prolog impiega invece circa 10 volte il tempo impiegato da SICStus in modalità compile. Le esecuzioni più lente risultano essere quelle di SICStus in modalità consult.

Esercizio 9.3. Il predicato built-in di SICStus `statistics`, utilizzato in un letterale del tipo `statistics(walltime,[T,_])`, ha successo istanziando T al tempo assoluto trascorso da quando la sessione Prolog è iniziata (compreso il tempo impiegato dalla eventuale attività di garbage collection. Per tenere conto solo del tempo di computazione effettivo va utilizzata

N	SICStus consult	SICStus compile	ECLiPSe compile debug	ECLiPSe compile no debug	SWI consult
14	170	10	41	30	150
15	120	20	30	20	120
16	971	90	220	170	882
17	571	50	130	100	570
18	5017	460	1122	881	4467
19	401	30	70	60	340
20	30093	3045	6349	4817	25517
21	1563	140	350	261	1322
22	306480	26909	68479	49561	263609
23	5428	470	1212	861	4687

Tabella 1. Problema delle regine: diversa efficienza dei Prolog

l'opzione `runtime` invece di `walltime`). Impiegando `statistics` si compari l'efficienza dei tre approcci al problema delle N regine visti in questo capitolo.

Esercizio 9.4. Si scriva un programma Prolog che risolva il problema analogo di posizionare M cavalli in una scacchiera $N \times N$ (si noti che a seconda dei valori scelti per M ed N potrebbero esistere una, nessuna o molte soluzioni).

2.2. Il problema del *map coloring*. Il problema del *map coloring* prevede di colorare una cartina geografica usando un dato insieme di colori in modo che non si abbiano mai due regioni confinanti colorate con lo stesso colore.

Possiamo rappresentare una mappa geografica costituita da N regioni con una struttura a grafo composta da N nodi. Gli archi rappresentano la relazione di confinanza (ad esempio, la presenza tra gli archi di `[X_1,X_2]` modella il fatto che la regione `X_1` confina con la regione `X_2`). Il seguente programma risolve il map coloring utilizzando l'approccio generate and test:

```
coloring(Nodi,Archi,Colori) :- funzione(Nodi,Colori),
                               verifica(Archi).
```

Si noti che, per come definito il predicato `coloring` nella clausola precedente, si prevede di utilizzare `coloring(Nodi,Archi,Colori)` in modo che `Nodi` sia una lista di variabili distinte, una per ogni nodo del grafo; `Archi` sia una lista di coppie di variabili (scelte tra quelle occorrenti in `Nodi`), ogni coppia modella un arco. `Colori` è intesa essere una lista di colori (ad esempio delle costanti). In caso di successo, una soluzione viene fornita istanziando le variabili di `Nodi` con termini estratti dalla lista dei colori. Il prossimo esempio chiarisce l'impiego del predicato.

Esempio 9.1. Consideriamo il grafo con quattro nodi n_1, n_2, n_3, n_4 e con gli archi

$$(n_1, n_2), (n_1, n_3), (n_1, n_4), (n_2, n_3), (n_3, n_4).$$

Supponiamo di disporre dei colori `giallo`, `verde`, `nero` e `bianco`. Il seguenti goal illustrano come utilizzare il predicato `coloring` (al nodo n_i viene fatta corrispondere la variabile `Xi`):

```
?- coloring([X1,X2,X3,X4],
         [[X1,X2],[X1,X3],[X1,X4],[X2,X3],[X3,X4]],
         [giallo,verde,nero,bianco]).
     yes   X1 = giallo, X2 = verde, X3 = nero, X4 = verde ?   ;
     yes   X1 = giallo, X2 = verde, X3 = nero, X4 = bianco ?   ;
     ⋮
```

Mentre:

```
?- coloring([X1,X2,X3,X4],
         [[X1,X2],[X1,X3],[X1,X4],[X2,X3],[X3,X4]],
         [giallo,verde]).
     no
```

Esercizio 9.5. Considerando le rappresentazioni dei grafi introdotte nella Sezione 3 del Capitolo 7, scrivere i predicati necessari per ottenere dalla rappresentazione di un grafo (data tramite i fatti `edge(·,·)` o tramite una lista degli archi) le liste di variabili e di coppie di variabili adeguate ad essere impiegate come argomenti del predicato `coloring`.

Completiamo il programma definendo il predicato `verifica` (il predicato `funzione` lo abbiamo definito a pagina 132):

```
verifica([]).
verifica([[X,Y]|R]) :- X \= Y, verifica(R).
```

Notiamo ora che il predicato `funzione(Nodi,Colori)` rappresenta la fonte principale della alta complessità computazionale di questa soluzione. Tramite questo predicato infatti vengono generate tutte le possibili configurazioni candidate ad essere soluzioni. Esse sono in numero esponenziale rispetto alla dimensione del problema: $|\texttt{Colori}|^{|\texttt{Nodi}|}$. Per ottenere un programma più efficiente sostituiamo il ricorso al predicato `funzione` con il predicato `assegna(Archi,Colori)`:

```
assegna([],_).
assegna([[X,Y]|R],Colori) :- member(X,Colori),
                             member(Y,Colori),
                             X \== Y,
                             assegna(R,Colori).
```

Questo predicato assegna un colore ad ogni nodo (istanziando le variabili con elementi della lista dei `Colori`) diverso dai colori assegnati ai nodi ad esso adiacenti. Il test di verifica è ora sempre soddisfatto grazie alla costruzione effettuata da `assegna`. Può essere quindi rimosso. Si noti che questa seconda soluzione non assegna colore ad eventuali nodi isolati. Tuttavia per i nodi isolati qualunque colore va bene.

Si osservi che l'algoritmo funziona correttamente perchè una volta che una variabile `Xi` (che rappresenta un nodo) viene istanziata con un elemento della lista `Colori` (tramite `member(Xi,Colori)`, ciò rappresenta l'assegnamento di un colore ad un nodo), questa istanziazione viene propagata a tutte le occorrenze di `Xi` nella lista `Archi`. Conseguentemente quando il programma processerà un successivo arco incidente in `Xi` troverà `Xi` già istanziata al colore scelto in precedenza.

Esercizio 9.6. Verificare i diversi tempi di esecuzione dei due algoritmi per il map coloring su grafi ottenuti da cartine geografiche reali (si utilizzi ad esempio `statistics` come suggerito nell'Esercizio 9.3).

2.3. Il problema del *commesso viaggiatore*. Il problema del commesso viaggiatore (in breve, *TSP*, dall'inglese traveling salesperson problem) è il problema di visitare un insieme di città esattamente una volta ritornando al punto di partenza, il tutto con un costo inferiore ad una certa costante (ad ogni spostamento da una città ad un'altra è infatti associato un costo).

Rappresentiamo le connessioni tra le città mediante un grafo diretto con archi pesati. Ogni arco quindi sarà un fatto del tipo: `arco(N_1,Costo,N_2)`. Il predicato principale sarà:

```
tsp(I,Nodi,[I|Path],K) :- permutazione(Nodi,[I|Path]),
                          verifica_lunghezza(I,[I|Path],K).
```

dove `Nodi` è inteso rappresentare la lista dei nodi da attraversare. `I` rappresenta il nodo di partenza (e di arrivo), `Path` risulterà istanziato (alla fine) con l'itinerario calcolato, mentre `K` indica il massimo costo accettabile per un cammino.
Il predicato `verifica_lunghezza` viene definito come:

```
verifica_lunghezza(I,Path,K) :- lunghezza(I,Path,C), C < K.

lunghezza(I,[N],C) :- arco(N,C,I).
lunghezza(I,[A,B|R],C) :- lunghezza(I,[B|R],C1),
                          arco(A,C2,B),
                          C is C1 + C2.
```

Come possibile ottimizzazione modifichiamo l'algoritmo in modo che gli archi siano aggiunti uno alla volta, fermando la costruzione di ogni soluzione candidata quando si supera il valore `K`:

```
tsp_fast(I,Nodi,Path,K) :- unoaduno(I,I,Nodi,Path,_,K).

unoaduno(I,S,[S],[S],C,K) :- arco(S,C,I), C < K.
unoaduno(I,S,Nodi,[S|Srim],C,K) :- seleziona(S,Nodi,Nodirim),
                                   Nodirim \= [],
                                   arco(S,C1,T),
                                   unoaduno(I,T,Nodirim,Srim,C2,K),
                                   C is C1 + C2,
                                   C < K.
```

3. Predicati di secondo ordine

3.1. Collezionare soluzioni. Nel Capitolo 7 abbiamo visto come sia possibile, usando i predicati `fail` e `write`, stampare l'elenco degli X per cui è verificato il predicato `p(X)`. Per far ciò tuttavia abbiamo sfruttato un side-effect (ovvero il fatto che `write` scriva sullo schermo). In realtà non abbiamo, di fatto, collezionato le risposte al goal `?-p(X)` in un termine Prolog (per esempio una lista).

In Prolog sono presenti tre primitive meta-logiche (o di logica del secondo ordine) che consentono di collezionare insiemi di risposte a goal. Esse sono: `findall`, `bagof` e `setof`. Spieghiamoli tramite degli esempi.

Consideriamo il seguente programma:

```
p(a).
p(a).
p(b).
p(a).
```

Analizziamo le risposte ai seguenti goal:

```
?- findall(X,p(X),L).
       yes  L=[a,a,b,a]

?- bagof(X,p(X),L).
       yes  L=[a,a,b,a]

?- setof(X,p(X),L).
       yes  L=[a,b]
```

Abbiamo che: `findall` colleziona tutte le risposte al goal `?-p(X)`; le soluzioni sono fornite (con eventuali ripetizioni) nell'ordine in cui l'SLD-risoluzione del Prolog le restituirebbe. Il predicato `setof` invece fornisce la lista ordinata senza ripetizioni delle soluzioni.

La differenza tra `findall` e `bagof` invece viene chiarita nell'esempio seguente. Consideriamo il programma

```
q(a,b).
q(a,c).
q(a,c).
q(b,d).
q(b,e).
```

Otteniamo:

```
?- findall(Y,q(X,Y),L).
       yes  L=[b,c,c,d,e]
```

In questo caso si ottengono tutti i valori che la `Y` assume nelle diverse soluzioni. Viene persa la relazione fra i valori che `Y` e `X` assomono nelle varie soluzioni: in un certo senso la variabile `X` viene mascherata (o meglio, i valori raccolti sono quelli assunti dal primo argomento di `findall`). Si può pensare che ciò che si ottiene sia l'insieme (a meno di ordine e ripetizioni) $\{$`Y:`$\exists$ `X q(X,Y)`$\}$.

Contrariamente a ciò, consideriamo le risposte che Prolog fornisce al seguente goal (quando l'utente digita ";" per ottenerle tutte)

```
?- bagof(Y,q(X,Y),L).
       X=a, L=[b,c,c] ;
       X=b, L=[d,e]
```

In questo caso Prolog fornisce, tutte i valori assunti da `Y` (gli stessi forniti da `findall`), ma partizionati rispetto ai corrispondenti valori delle altre variabili (nel caso, solo `X`). Ciò corrisponde a determinare l'insieme $\{$`Y:q(X,Y)`$\}$ per ogni possibile valore di `X`.

Si noti la differenza rispetto al goal seguente:

```
setof(Y,q(X,Y),L).
          yes  X=a, L=[b,c] ;
          yes  X=b, L=[d,e]
```

In quest'ultimo caso si hanno i medesimi risultati ottenuti con `bagof` ma con l'eliminazione di eventuali ripetizioni.

Si vuol comunque far notare come `setof` non sia un un vero operatore insiemistico. Infatti alla richiesta: `?- setof(Y,q(a,Y),L).` viene fornito come risultato `yes  L=[b,c]` (come ci si attenderebbe). Tuttavia al goal `?- setof(Y,q(a,Y),[c,b]).` il risolutore fallisce nella ricerca di una risposta e risponde *no*, anche se dal punto di vista insiemistico i due insiemi $\{b, c\}$ e $\{c, b\}$ sono equivalenti.

Esercizio 9.7. Utilizzando `setof`, definire un predicato di ordinamento di una lista con eliminazione di ripetizioni.

3.2. Alterazione dinamica del programma: assert e retract. Prolog mette a disposizione dei predicati che hanno l'effetto di modificare (dinamicamente, ovvero durante la ricerca delle soluzioni ad un goal) l'insieme delle clausole del programma.

Mediante i comandi

- `assert` (e le sue varianti `asserta` e `assertz`);
- `retract`

è possibile aggiungere (`assert`) o rimuovere (`retract`) fatti o regole dal programma. In particolare `asserta(Clause)` aggiunge la clausola `Clause` all'inizio del programma. Invece, `assertz(Clause)` aggiunge `Clause` in fondo al programma.[4]

Il predicato `retract(Clause)`, elimina dal programma, una alla volta, tutte le clausole che unificano con il termine `Clause`. Se `Clause` è della forma p(t_1,...,t_n) allora saranno rimossi i fatti che unificano con p(t_1,...,t_n); se invece si vuole eliminare una clausola si dovrà utilizzare un atomo del tipo retract(H :- B_1,...,B_k), dove H,B_1,...,B_k, sono atomi.

L'impiego di una clausola come argomento dei predicati `assert` e `retract`, come illustrato sopra, è possibile in quanto (come menzionato nella Sezione 12 del Capitolo 7) Prolog implicitamente definisce i due operatori "`:-`" e "`,`" come segue:

```
:- op( 1200, xfx, :- ).
:- op( 1000, xfy, , ).
```

Conseguentemente, una clausola è nel contempo una regola ed un termine.

Ci si potrebbe chiedere quando una modifica dinamica del programma manifesti i suoi effetti sulla computazione di un goal. Solitamente (e questo è ciò che accade in SICStus Prolog), al momento in cui si effettua la invocazione di un goal (ovvero si risolve un atomo

[4]Alcuni interpreti Prolog, e questo è il caso di SICStus, permettono la modifica dinamica della definizione di un predicato, (tramite `retract`, `assert` e sue varianti) solamente se il predicato in questione è stato dichiarato di tipo `dynamic`, con la direttiva:

:- dynamic(*nomepredicato/arità*).

Ciò perchè, in genere, per permettere modifiche dinamiche della definizione di un predicato, l'interprete Prolog deve effettuare una gestione diversa da quella prevista per predicati statici.

utilizzando delle clausole del programma) si compie un *congelamento* della definizione del predicato dinamico; la computazione relativa al goal procede tenendo conto solo delle clausole presenti alla invocazione. Quindi anche se si eseguono degli `assert` o dei `retract`, l'eventuale backtracking terrà conto solo delle clausole che esistevano al momento della invocazione iniziale. Si può dire, intuitivamente, che solo al termine della computazione del goal si rendono effettivi gli effetti degli eventuali `assert` e `retract`. Tali effetti saranno quindi visibili solo alla successiva invocazione di un goal.

Un semplice esempio di utilizzo di `assert` e `retract` è la definizione di un contatore:

```
inizializza_contatore :- assert(timer(0)).

incrementa_contatore :- retract(timer(T)),
                        T1 is T + 1,
                        assert(timer(T1)).

get_val_contatore(T) :- timer(T).
```

Chiudiamo questa sezione osservando che l'uso di `assert` e `retract`, comportando modifiche dinamiche del programma, può rendere il comportamento stesso dell'interprete difficilmente prevedibile a seguito di una semplice lettura (statica) del programma. Inoltre uno stile di programmazione che ricorra a questi predicati porta a scrivere programmi poco dichiarativi che tendono a basarsi su metodologie di programmazione tipiche della programmazione imperativa. I programmi che utilizzano `assert` e `retract` tendono quindi ad essere poco leggibili, poco prevedibili, e difficilmente modificabili.

4. Meta-interpretazione

Nella Sezione 3.2 abbiamo visto che sia le clausole che i fatti sono in realtà legittimi termini Prolog (costruiti tramite gli operatori "`:-`" e "`,`"). Abbiamo inoltre descritto alcuni predicati, quali `assert` e `retract`, che permettono di sfruttare la duplice veste delle clausole Prolog. In questa sezione approfondiremo tale aspetto introducendo i concetti base della *meta-programmazione*. Un *meta-programma* è un programma in grado di processare come input un altro programma. Nel caso in cui il linguaggio del meta-programma sia lo stesso del suo input si parla di *meta-interpretazione*.

Il linguaggio Prolog offre il predicato extra-logico `clause/2` che permette di accedere al contenuto della base di clausole. L'invocazione del goal

```
?- clause(Testa, Corpo).
```

ha successo se le variabili `Testa` e `Corpo` sono unificabili, rispettivamente, con la testa ed il corpo di una delle clausole del programma correntemente in esecuzione. Nel caso `Testa` unifichi con un fatto allora il corpo viene assunto essere il termine `true`. Tramite il backtracking il letterale `clause(Testa, Corpo)` determinerà diverse istanziazioni corrispondenti a diverse clausole del programma.

Il seguente è un semplice meta-interprete per il nucleo puramente dichiarativo del linguaggio Prolog:

```
risolvi(true).
risolvi((A,B)) :- risolvi(A),
                  risolvi(B).
risolvi(A) :- clause(A,B),
              risolvi(B).
```

Esercizio 9.8. Si estenda il semplice meta-interprete descritto sopra in modo che possa trattare anche predicati built-in ed extra-logici, quali ad esempio `functor` e `setof`.

Il precedente meta-interprete presenta una estrema semplicità, ma il suo schema generale può essere impiegato per realizzare dei meta-programmi più sofisticati. Il seguente è un meta-interprete che oltre a risolvere un goal appoggiandosi sul sottostante interprete Prolog, costruisce una dimostrazione del goal stesso.

```
dimostra(true,true).
dimostra((A,B),(ProvaDiA,ProvaDiB)) :- dimostra(A,ProvaDiA),
                                       dimostra(B,ProvaDiB).
dimostra(A,ProvaDiA) :- clause(A,B),
                        dimostra(B,ProvaDiB).
```

Esercizio 9.9. Si scriva un meta-interprete che utilissi la regola di selezione del letterale rightmost in luogo della regola leftmost del Prolog.

Esercizio 9.10. Si scriva un meta-interprete che effettui la costruzione/visita dell'SLD-albero procedendo in ampiezza anziché in profondità come avviene nell'interprete Prolog. [SUGGERIMENTO: si veda anche Capitolo 10.]

Come ultimo esempio di meta-interpretazione mostriamo come sia possibile arricchire l'interprete Prolog in modo da associare ad ogni clausola, fatto e goal un coefficiente di probabilità. In questo modo realizzeremo un semplicissimo e minimale interprete per un "Prolog probabilistico". I valori di probabilità saranno come usuale numeri reali compresi tra 0 e 1 (si veda anche [**SS97**]).

```
probabile(true,1).
probabile((A,B),Prob) :- probabile(A,ProbA),
                         probabile(B,ProbB),
                         minimo(ProbA,ProbB).
probabile(A,Prob) :- clause_prob(A,B,ProbC),
                     probabile(B,ProbB),
                     Prob is ProbC*ProbB.
```

Si noti l'impiego del predicato `clause_prob` in luogo di `clause`, al fine di gestire clausole Prolog con assegnato un coefficiente di incertezza.

Esercizio 9.11. Si completi il meta-interprete "probabilistico" fornendo le definizioni dei predicati `clause_prob` e `minimo`. Si tenga presente che per completare il meta-interprete, sarà necessario gestire i coefficienti di probabilità assegnati alle clausole del programma.

Un ulteriore esempio nello stesso spirito prevede di assegnare ad ogni clausola un coefficiente di incertezza e di stabilire una soglia al di sotto del quale il valore di incertezza denota la falsità:

```
incertezza(true,1,Soglia) :- !.
incertezza((A,B),C,Soglia) :- !, incertezza(A,C1,Soglia),
                                incertezza(B,C2,Soglia),
                                minimo(C1,C2,C).
incertezza(A,C,Soglia) :- clause_prob(A,B,C1),
                          C1 > Soglia,
                          NuovaSoglia is Soglia/C1,
                          incertezza(B,C2,NuovaSoglia),
                          C is C1*C2.
```

5. Esercizi

Esercizio 9.12. Scrivere un programma Prolog in cui si definisca il predicato

`mcd(+Numero1,+Numero2,?Numero).`

Il predicato deve essere vero quando `Numero` è il massimo comune divisore di `Numero1` e `Numero2`. A differenza di quanto visto nella Sezione 1.1 non si utilizzi la rappresentazione dei numeri interi del due numeri Capitolo 3, ma si definisca un predicato che operi sugli interi di Prolog.

Esercizio 9.13. Scrivere un programma Prolog in cui si definisca il predicato

`primi_tra_loro(+Numero1,+Numero2).`

Il predicato deve essere vero quando `Numero1` e `Numero2` sono due numeri naturali positivi primi tra loro.

Esercizio 9.14. Scrivere un programma Prolog in cui si definisca il predicato

`setaccio(Lista,Selezione).`

Si assuma che tale predicato venga utilizzato fornendo una lista di interi come primo argomento. Il predicato sarà vero se `Selezione` unifica con la lista degli interi i che appaiono in posizione i-esima nella lista `Lista`. L'ordine in cui appaiono viene preservato. Le posizioni nella lista sono determinate contando da sinistra a destra. Ad esempio, l'atomo `setaccio([1,7,3,8,2,6,4,1],[1,3,6])` sarà vero.

Esercizio 9.15. Risolvere l'Esercizio 9.14 assumendo che le posizioni nella lista siano determinate contando da destra a sinistra. Ad esempio, l'atomo `setaccio([1,7,3,8,2,6,4,1], [7,1])` sarà vero.

Esercizio 9.16. Scrivere un programma Prolog in cui si definisca il predicato

`specchio(+Termine1,?Termine2).`

Il predicato deve essere vero quando il termine Prolog `Termine2` è ottenuto specchiando `Termine1`, ovvero invertendo l'ordine dei suoi argomenti e applicando ricorsivamente la stessa trasformazione a tutti gli argomenti. Ad esempio sottoponendo il goal

```
?- specchio(f(a,h(1,2),g(1,a,X)), T),
```

si otterrà la risposta

```
yes   T=f(g(X,a,1), h(2,1), a)
```

(Si assuma che le variabili non vengano rinominate dall'interprete Prolog).

Esercizio 9.17. Scrivere un programma Prolog in cui si definisca il predicato

`unico(+Lista,?Elemento)`

. Si assuma che tale predicato venga utilizzato fornendo una lista di interi come primo argomento (si osservi il *mode* del predicato: `(+,?)`). Il predicato sarà vero se `Elemento` unifica con un elemento che compare una ed una sola volta nella lista `Lista`.

Esercizio 9.18. Scrivere un programma Prolog in cui si definisca il predicato

`fresh(+Termine,-OutTerm)`

(si osservi il *mode* del predicato: `(+,-)`). Il predicato deve essere vero quando `OutTerm` è un termine che ha la stessa struttura del termine `Termine` con l'unica differenza che tutte le variabili che occorrevano in `Termine` sono state rimpiazzate da variabili NUOVE. Il programma può essere realizzato anche facendo in modo che tutte le occorrenze di variabili (nuove) in `OutTerm` siano tra loro distinte. (Ovvero NON si richiede che a occorrenze diverse della stessa variabile in `Termine` corrispondano occorrenze diverse della stessa (nuova) variabile in `OutTerm`.)

Esercizio 9.19. Scrivere un programma Prolog in cui si definisca il predicato

`isovar(+Termine1,+Termine2)`

Il predicato deve essere vero quando i due argomenti sono due termini ottenibili l'uno dall'altro rinominando le variabili (la rinomina deve essere coerente, ovvero se a X si sostituisce Y, allora a tutte e sole le occorrenze della X deve venir sostituita Y).

Esercizio 9.20. Scrivere un programma Prolog in cui si definisca il predicato

`select_n(+Lista,+N,?Elemento,?Resto).`

Il predicato deve essere vero quando all'atto della sua invocazione `N` è istanziato ad un numero naturale, `Lista` è istanziata ad una lista di almeno `N` oggetti. Inoltre il predicato sarà vero quando `Elemento` unifica con l'elemento di posizione `N`-esima, e `Resto` con la lista ottenuta da `Lista` eliminando l'elemento di posizione `N`-esima.

CAPITOLO 10

Searching

In questo capitolo vedremo come sia possibile utilizzare tecniche di ricerca in Prolog per risolvere dei problemi. Per molti problemi interessanti infatti non esistono algoritmi efficienti (ovvero di complessità polinomiale) mentre si dispone di metodi efficienti per verificare se una soluzione candidata sia effettivamente la soluzione desiderata (è il caso della classe di problemi NP [**GJ79**]. Si veda ad esempio [**PS98**], per maggiori dettagli sulle problematiche legate alla complessità computazionale).

Le tecniche di searching permettono di individuare una soluzione (se esiste) a seguito di una enumerazione (di parte) delle soluzioni candidate. Sotto questo punto di vista, l'approccio risulta simile a quanto illustrato descrivendo la strategia generate-and-test nel Capitolo 9. In questo capitolo porremo particolare enfasi allo sviluppo della fase generate, studiando come sia possibile disciplinare la generazione delle soluzioni.

L'approccio che utilizzeremo prevede di rappresentare un problema, in modo dichiarativo, in termini di uno *spazio degli stati*. Questo spazio ha la struttura di un grafo (solitamente infinito) in cui i nodi identificano gli stati del problema (ognuno rappresenta una configurazione che è potenzialmente soluzione). Tra due nodi è presente un arco se è possibile trasformare uno stato nell'altro, e quindi effettuare una *mossa* da una configurazione ad un'altra. Si distingue uno stato iniziale da cui inizia la ricerca.

Un approccio alternativo prevede di descrivere (dichiarativamente, per quanto ci riguarda) uno *spazio dei problemi*. La struttura di questo spazio è ancora un grafo in cui però ogni nodo modella un problema, mentre ogni arco uscente da un nodo n individua una possibile decomposizione o trasformazione del problema n. Come esempio citiamo la procedura di SLD-risoluzione: essa visita uno spazio dei problemi alla ricerca di un problema *risolto* (rappresentato dal goal vuoto).

Indipendentemente dall'approccio, la struttura sottostante è quella di grafo e la metodologia adottata è quella di cercare un cammino tra un nodo iniziale e un nodo che codifica una delle soluzioni. In quanto segue studieremo le principali tecniche di ricerca su grafi.

1. Depth-first search

Il seguente programma implementa lo schema generale di una ricerca con strategia depth-first (in profondità). Il predicato `mossa_possibile` identifica una mossa possibile relativamente allo stato attuale. In base alla mossa generata, il predicato `prossimo_stato` determina un prossimo stato candidato ad essere visitato, mentre `stato_ammesso` verifica se quest'ultimo sia ammissibile (in relazione ad eventuali vincoli imposti dalla descrizione del problema).

Si noti come la strategia depth-first sia implementata appoggiandosi direttamente sul meccanismo del backtracking dell'interprete Prolog.

```
risolvi_dfs(Stato,StatiVisitati,[]) :- finale(Stato).
risolvi_dfs(Stato,StatiVisitati,[Mossa|MosseFatte]) :-
            mossa_possibile(Stato,Mossa),
            prossimo_stato(Stato,Mossa,NuovoStato),
            stato_ammesso(NuovoStato),
            not member(NuovoStato,StatiVisitati),
            risolvi_dfs(NuovoStato,[NuovoStato|StatiVisitati],MosseFatte).
```

La ricerca viene iniziata partendo dallo stato iniziale:

```
?-iniziale(StatoIniz),risolvi_dfs(StatoIniz,[StatoIniz],Mosse).
```

Per utilizzare questa strategia nella soluzione di un particolare problema è necessario

- scegliere una rappresentazione per gli stati del problema;
- dichiarare i predicati `stato_ammesso`, `prossimo_stato` e `mossa_possibile`.

Il prossimo esempio illustra una possibile soluzione del problema dei cannibali e dei missionari che impiega una ricerca depth-first.

Esempio 10.1. Si consideri seguente problema: sulla sponda di un fiume vi sono tre cannibali e tre missionari. Tutti vogliono recarsi sull'altra sponda del fiume. I cannibali hanno una barca che può trasportare però al più due persone alla volta. I cannibali sono molto sospettosi e non vogliono restare in minoranza, su nessuna delle due sponde altrimenti, temono, verranno convertiti.

Descriviamo una configurazione tramite il termine `status(Pos,ML,CL,MR,CR)`, dove `Pos` indica la posizione della barca, mentre `ML` (`MR`) e `CL` (`CR`) indicano il numero di missionari e cannibali sulla riva sinistra (risp. destra) del fiume. Lo stato iniziale sarà quindi descritto dal termine `status(sinistra,3,3,0,0)`, mentre quello finale sarà `status(destra,0,0,3,3)`.

Per risolvere il problema utilizzando `risolvi_dfs` forniamo le seguenti definizioni dei predicati `stato_ammesso`, `prossimo_stato` e `mossa_possibile`:

```
iniziale(status(sinistra,3,3,0,0)).

finale(status(destra,0,0,3,3)).

mossa_possibile(status(sinistra,M,_,_,_), attraversa(1,0)) :- M>=1.
mossa_possibile(status(sinistra,_,C,_,_), attraversa(0,1)) :- C>=1.
mossa_possibile(status(sinistra,M,C,_,_), attraversa(1,1)) :- M>=1, C>=1.
mossa_possibile(status(sinistra,M,_,_,_), attraversa(2,0)) :- M>=2.
mossa_possibile(status(sinistra,_,C,_,_), attraversa(0,2)) :- C>=2.
mossa_possibile(status(destra,_,_,M,_), attraversa(1,0)) :- M>=1.
mossa_possibile(status(destra,_,_,_,C), attraversa(0,1)) :- C>=1.
mossa_possibile(status(destra,_,_,M,C), attraversa(1,1)) :- M>=1, C>=1.
mossa_possibile(status(destra,_,_,M,_), attraversa(2,0)) :- M>=2.
mossa_possibile(status(destra,_,_,_,C), attraversa(0,2)) :- C>=2.
```

```
prossimo_stato(status(sinistra,ML0,CL0,MR0,CR0), attraversa(MB,CB),
               status(destra,ML,CL,MR,CR)) :- ML is ML0-MB, CL is CL0-CB,
                                              MR is MR0+MB, CR is CR0+CB.
prossimo_stato(status(destra,ML0,CL0,MR0,CR0), attraversa(MB,CB),
               status(sinistra,ML,CL,MR,CR)) :- ML is ML0+MB, CL is CL0+CB,
                                                MR is MR0-MB, CR is CR0-CB.

stato_ammesso(status(_,_,3,_,0)) :- !.
stato_ammesso(status(_,_,0,_,3)) :- !.
stato_ammesso(status(_,M,M,N,N)).
```

In questo programma il termine `attraversa(M,C)` indica che la mossa da effettuare è lo spostamento di `M` missionari ed `C` cannibali da una sponda all'altra (ovviamente con `M`+`C`≤ 2).

Esercizio 10.1. Nell'Esempio 10.1 si è optato per una rappresentazione degli stati che è ridondante. Infatti sapendo che vi sono tre missionari e tre cannibali sarebbe stato sufficiente rappresentare i presenti su una sola delle due rive. Si modifichi il programma dell'Esempio 10.1 un tal senso.

Esercizio 10.2. Si scriva un programma Prolog che risolva una variante del problema dei missionari e cannibali in cui vi sono N missionari e M cannibali (con $M \geq N$).

2. Hill climbing search

Abbiamo visto che nella strategia depth-first si impiega un predicato `mossa_possibile` per determinare quale siano le mosse attuabili. Queste sono solitamente generate in modo non deterministico. Una possibile miglioria dell'algoritmo di depth-first consiste nell'applicare una euristica per scegliere, tra le varie possibilità, la mossa *più promettente*. Entra in gioco quindi una funzione di valutazione che impone un ordinamento sull'insieme delle mosse possibili. In pratica si assegna ad ogni possibilità un punteggio e si sceglie la mossa con il punteggio più alto.

La strategia hill climbing è una variante della strategia depth-first in cui la scelta cade appunto sulla mossa con punteggio migliore. Come suggerisce il nome, si può immaginare una computazione che adotta hill climbing allo stesso modo di una scalata ad una collina: si procede percorrendo stati sempre più vicini alla soluzione, rappresentata dalla cima della collina.

Lo schema del programma è ottenibile da quello depth-first sostituendo al predicato `mossa_possibile` il seguente predicato che a sua volta è definito in base a `mossa_possibile`:

```
hill_climb(Stato,MossaMigliore) :- setof(M,mossa_possibile(Stato,M),Mosse),
                                   ordina_mosse(Mosse,Stato,MosseOrdinate),
                                   member([MossaMigliore,Peso],MosseOrdinate).
```

Dato lo stato attuale vengono individuate tutte le mosse possibili (tramite `setof`). Il predicato `ordina_mosse` determina l'insieme degli stati raggiungibili (in una mossa) e, sfruttando il predicato `valuta`, ordina tali mosse:

```
ordina_mosse(Mosse,Stato,MosseOrdinate) :-
                             ordina_mosse(Mosse,Stato,[],MosseOrdinate).

ordina_mosse([Mossa|Mosse],Stato,Acc,MosseOrdinate) :-
                           prossimo_stato(Stato,Mossa,NuovoStato),
                           valuta(NuovoStato,Peso),
                           insert([Mossa,Peso],Acc,Acc1),
                           ordina_mosse(Mosse,Stato,Acc1,MosseOrdinate).
ordina_mosse([],Stato,Acc,Acc).

insert(X,[],[X]).
insert([M,P],[[M1,P1]|X],[[M,P],[M1,P1]|X]) :- P >= P1.
insert([M,P],[[M1,P1]|X],[[M1,P1]|X1]) :- P < P1, insert([M,P],X,X1).
```

Chiaramente, il predicato `valuta` deve essere dichiarato dal programmatore contestualmente allo specifico problema affrontato ed alla sua rappresentazione. In questo programma si assume che `member` sia definito nel modo usuale (come a pag. 127); in questo modo la selezione delle mosse avviene partendo dalla migliore (la prima della lista `Mosse`) e procedendo verso la peggiore (l'ultima della lista).

Sfruttando la precedente definizione del predicato `hill_climb` l'algoritmo di visita hill climbing viene specificato come segue:

```
risolvi_hill_climbing(Stato,StatiVisitati,[]) :-
                     finale(Stato).
risolvi_hill_climbing(Stato,StatiVisitati,[MossaMigliore|MosseFatte]) :-
                     hill_climb(Stato,MossaMigliore),
                     prossimo_stato(Stato,MossaMigliore,NuovoStato),
                     stato_ammesso(NuovoStato),
                     not member(NuovoStato,StatiVisitati),
                     risolvi_hill_climbing(NuovoStato,
                                           [NuovoStato|StatiVisitati]
                                           MosseFatte).
```

Esempio 10.2. Si consideri il problema dei tre missioneri e tre cannibali. La seguente è una possibile dichiarazione per il predicato `valuta`:

```
valuta(status(_,ML,CL,_,_),1) :- ML+CL =:= 1, !.
valuta(status(left,ML,CL,_,_),Peso) :- Peso is (ML+CL-2)*2+1.
valuta(status(right,ML,CL,_,_),Peso) :- Peso is (ML+CL)*2.
```

Esercizio 10.3. Nell'algoritmo di visita hill climbing sopra descritto si determinano tutti gli stati raggiungibili (a partire da quello attuale) tramite una mossa ammessa. Successivamente, una volta scelta la mossa migliore, si ricalcola lo stato raggiungibile effettuando la mossa migliore. Modificare il programma in modo da non ripetere questo calcolo.

3. Breadth-first search

Considerando le possibili opzioni nella scelta delle funzioni di valutazione, ovvero del modo in cui ordinare le mosse possibili, possiamo descrivere una strategia che privilegia la mossa che conduce allo stato più vicino allo stato iniziale (ovvero il nodo raggiungibile percorrendo il cammino più corto a partire dal nodo iniziale). Tale strategia è la breadth-first search (in ampiezza).

```
ricerca_bf(Mosse) :- iniziale(StatoIniziale),
                     coda_vuota(Coda0),
                     accoda(status(StatoIniziale,[]),Coda0,Coda),
                     risolvi_bf(Coda,[],Mosse).

risolvi_bf(Coda,_,MosseFatte) :-
                     estrai_da_coda(status(Stato,Cammino),Coda,_),
                     finale(Stato),
                     reverse(Cammino,[],MosseFatte).
risolvi_bf(Coda0,Passato,MosseFatte) :-
                     estrai_da_coda(status(Stato,Cammino),Coda0,Coda1),
                     findall(M,mossa_possibile(Stato,M),Mosse),
                     filtra_stati(Mosse,Stato,Cammino,Passato,Coda1,Coda),
                     risolvi_bf(Coda,[Stato|Passato],MosseFatte).
```

In questo programma si impiega una coda per memorizzare gli stati non ancora visitati, ordinati in base alla lunghezza del cammino che li connette allo stato iniziale. Ogni elemento della coda è un termine `status(Stato,Cammino)` dove accanto allo `Stato` si rappresenta anche il `Cammino` che vi conduce. Il predicato `filtra_stati` ha lo scopo di inserire nella coda degli stati da visitare tutti gli stati ottenibili dallo stato attuale con una mossa e non ancora visitati (ovvero non presenti nella lista `Passato`).

```
filtra_stati([],_,_,_,Coda,Coda).
filtra_stati([Mossa|Mosse],Stato,Cammino,Passato,Coda0,Coda) :-
                  prossimo_stato(Stato,Move,NuovoStato),
                  stato_ammesso(NuovoStato),
                  not member(NuovoStato,Passato),
                  !,
                  accoda(status(NuovoStato,[Move|Cammino]),Coda0,Coda1),
                  filtra_stati(Mosse,Stato,Cammino,Passato,Coda1,Coda).
filtra_stati([_|Mosse],Stato,Cammino,Passato,Coda0,Coda):-
                  filtra_stati(Mosse,Stato,Cammino,Passato,Coda0,Coda).
```

Il predicato `reverse` è stato definito nel Capitolo 7. La struttura dati coda viene invece implementata semplicemente tramite le seguenti tre clausole:

```
coda_vuota(q(0, Ys, Ys)).
accoda(Elem, q(Lunghezza,Ys,[Elem|Zs]), q(s(Lunghezza),Ys,Zs)).
estrai_da_coda(Elem, q(s(Lunghezza),[Elem|Ys],Zs), q(Lunghezza,Ys,Zs)).
```

dove il primo argomento del termine `q(N,X,Y)` indica la lunghezza della coda che si ottiene come *differenza* tra `X` e `Y`.

4. Best-first search

Studiando la strategia hill climbing si osserva che essa fornisce le prestazioni migliori quando la funzione di valutazione assegna i pesi in modo che vi sia una sola collina. Ciò perchè la strategia procede in base a valutazioni locali: si sceglie lo stato migliore tra quelli vicini allo stato attuale.

La strategia best-first è una generalizzazione della strategia breadth-first che permette di superare questo limite della strategia hill climbing. L'idea base consiste nel mantenere una *frontiera*, ovvero un insieme di stati da visitare, ma senza il vincolo che questo insieme sia gestito come una coda (breadth-first) o come una pila (depth-first). Ogni volta che è necessario aggiungere degli stati alla frontiera, questi vengono pesati (similmente a quanto accade nella strategia hill climbing) e in base al loro peso vengono inseriti, in ordine, nella frontiera.

Ecco una possibile implementazione in Prolog di questa strategia:

```
risolvi_bestf([status(Stato,Peso,Cammino)|_],StatiVisitati,Mosse) :-
                    finale(Stato),
                    reverse(Cammino,[],Mosse).

risolvi_bestf([status(Stato,Peso,Cammino)|Frontiera],StatiVisitati,Mosse) :-
                    setof(M,mossa_possibile(Stato,M),Ms),
                    aggiorna_frontiera(Ms,Stato,Cammino,StatiVisitati,
                                       Frontiera,NuovaFrontiera),
                    risolvi_bestf(NuovaFrontiera,
                                  [NuovoStato|StatiVisitati],Mosse).
```

Il predicato `aggiorna_frontiera` viene utilizzato per ottenere la nuova frontiera aggiungendo alla attuale tutti gli stati che si possono raggiungere con una mossa dallo stato corrente. Ogni stato viene rappresentato nella frontiera con un termine della forma `status(A,P,C)` che codifica oltre al nodo/stato vero e proprio, il suo peso `P` e la sequenza `C` di mosse compiute per raggiungerlo.

```
aggiorna_frontiera([Mossa|Mosse],Stato,Cammino,
                   StatiVisitati,Frontiera,NuovaFrontiera) :-
              prossimo_stato(Stato,Mossa,NuovoStato),
              stato_ammesso(NuovoStato),
              not member(NuovoStato,StatiVisitati),
              valuta(NuovoStato,Peso),
              ins_in_frontiera(status(NuovoStato,Peso,[Mossa|Cammino]),
                               Frontiera,Frontiera0),
              aggiorna_frontiera(Mosse,Stato,Cammino,
                                 StatiVisitati,Frontiera0,
                                 NuovaFrontiera).
aggiorna_frontiera([],Stato,Cammino,StatiVisitati,Frontiera,Frontiera).
```

La inserzione di un nuovo stato avviene confrontando il suo peso con il peso degli stati già presenti nella frontiera:

```
ins_in_frontiera(Status,[],[Status]).
ins_in_frontiera(status(S,P,C),[status(S,P,C1)|Rest],
                 [status(S,P,C)|Rest]).
ins_in_frontiera(status(S,P,C),[status(S1,P1,C1)|Rest],
                 [status(S,P,C),status(S1,P1,C1)|Rest]) :- S\==S1, P<P1.
ins_in_frontiera(status(S,P,C),[status(S1,P1,C1)|Rest],
                 [status(S,P,C),status(S1,P1,C1)|Rest]) :- S\==S1, P1<P.
```

5. Lower cost first search e A* search

Le strategie di ricerca illustrate finora fanno uso di una funzione di valutazione del peso di ogni stato e ordinano di conseguenza l'insieme degli stati da visitare. Il peso è quindi da intendersi come una stima della distanza dello stato corrente dalla soluzione. Questa stima in genere è ottenuta tramite delle valutazioni euristiche in base alle informazioni e alla conoscenza che si possiede sullo stato corrente (o più in generale sugli stati già visitati).

Sovente risulta importante anche valutare la soluzione in termini del suo costo. Frequentemente, infatti, alle mosse che conducono da uno stato ad un altro (ovvero gli archi del grafo degli stati) sono associati dei costi. Ad una soluzione risulta quindi associato un costo globale determinato dalla somma dei costi relativi al cammino che la connette allo stato iniziale (nel caso più semplice di costi uguali per tutte le mosse, allora il costo globale corrisponderà alla lunghezza del cammino). Si osservi che il costo dello stato corrente (al pari dei costi di tutti gli stati già visitati) è sempre noto in quanto è determinato dal cammino costruito per raggiungere lo stato stesso.

È possibile quindi ipotizzare una strategia di ricerca che, come unico criterio, tenda solamente a minimizzare il costo globale della soluzione generata. Si ottiene così la strategia lower cost first: la visita procede scegliendo di volta in volta come prossimo stato quello che presenta il cammino più economico.

Qualora la selezione del prossimo stato avvenga non solo in base alla stima della sua distanza dalla soluzione, ma tenga conto anche del costo dello stato corrente (ovvero del costo globale del cammino percorso dallo stato iniziale allo stato corrente) si ottiene la strategia A*.

6. Esercizi

Esercizio 10.4. Si fornisca un programma Prolog che implementi la strategia lower cost first.

Esercizio 10.5. Si fornisca un programma Prolog che implementi la strategia A*.

Esercizio 10.6. Si confronti l'efficienza delle varie strategie descritte, confrontandone i comportamenti rispetto ai problemi menzionati negli esempi ed esercizi in questo capitolo.

CAPITOLO 11

Parsing e DCG

In questo capitolo illustreremo una applicazione del Prolog al problema del parsing. Tratteremo principalmente il caso delle grammatiche libere dal contesto. Lo strumento che impiegheremo sono le *definite clause grammar*, brevemente DCG.

1. Difference list

Le definite clause grammar sono basate su una rappresentazione delle liste che rende possibile l'operazione di concatenazione in modo particolarmente efficiente. Questa struttura dati è detta *difference list*.

Una difference list può essere vista come un termine della forma `Lista-Suffisso`. Dove sia `Lista` che `Suffisso` sono due liste di oggetti. Sussiste il vincolo che `Suffisso` sia la parte finale della lista `Lista`; quindi gli elementi della difference list `Lista-Suffisso` sono gli elementi della lista `Lista` privata della parte finale `Suffisso`. Ad esempio, la difference list `[10,3,15,7,2,8]-[7,2,8]` ha come elementi `10`, `3`, e `15`. La difference list `[a,b,g]-[a,b,g]` è invece vuota. In questa descrizione abbiamo scelto di utilizzare l'operatore infisso "`-`" per rappresentare una difference list con un unico termine; tuttavia questa è solo una convenzione, in quanto una difference list può essere descritta semplicemente fornendo una lista e un suo suffisso.

Osserviamo che la stessa difference list può essere rappresentata in modi alternativi. Ad esempio, la difference list `[a,t,d,b,f]-[b,f]`, che contiene gli elementi `a,t,d`, potrebbe essere rappresentata anche come `[a,t,d,f,k,r,u]-[f,k,r,u]`, oppure `[a,t,d,f]-[f]`, ecc. In generale, possiamo lasciare implicito il suffisso e quindi rappresentare la difference list precedente come `[a,t,d|S]-S`, dove `S` è una variabile. Così facendo possiamo sfruttare la variabile `S` come una sorta di "puntatore" alla fine della lista. Il principale vantaggio di questa rappresentazione è che, grazie alla unificazione, la operazione di concatenazione di due liste (append) può venire implementata in Prolog con complessità costante. La seguente è una semplice definizione del predicato `append_dl`:

```
append_dl(X-Y,Y-Z,X-Z).
```

Il modo in cui avviene l'operazione di append è semplice. Ad esempio, supponiamo di voler concatenare le difference list `[a,t,d,b,f|S1]-S1` e `[1,2,3,4|S2]-S2`. Il seguente goal risolve il problema:

```
?-append_dl([a,t,d,b,f|S1]-S1,[1,2,3,4|S2]-S2,Risultato).
```

Un solo passo di SLD-risoluzione è sufficiente. Tramite l'algoritmo di unificazione dai due atomi si ottiene infatti il seguente m.g.u.:

```
[X/[a,t,d,b,f,1,2,3,4|S2],  Y/[1,2,3,4|S2],  Z/S2,
   S1/[1,2,3,4|S2],  Risultato/[a,t,d,b,f,1,2,3,4|S2]-S2]
```

istanziando `Risultato` alla concatenazione delle difference list.

2. Definite clause grammar

Come noto una grammatica libera dal contesto può essere descritta da un insieme di regole atte a riscrivere simboli *non terminali* fino a che si ottenga una sequenza di simboli *terminali*.

In questa sezione studieremo come sia possibile realizzare in Prolog un parser a discesa ricorsiva.

ESEMPIO 11.1. La seguente è una grammatica libera dal contesto per il linguaggio delle espressioni aritmetiche sui numeri interi.

$$\begin{array}{rcl}
\langle expr\rangle & ::= & \langle fattore\rangle\ \ \langle mulop\rangle\ \ \langle expr\rangle \\
\langle expr\rangle & ::= & \langle fattore\rangle \\
\langle fattore\rangle & ::= & \langle addendo\rangle\ \ \langle addop\rangle\ \ \langle fattore\rangle \\
\langle fattore\rangle & ::= & \langle addendo\rangle \\
\langle addendo\rangle & ::= & (\ \langle expr\rangle\ ) \\
\langle addendo\rangle & ::= & \langle numero\rangle \\
\langle numero\rangle & ::= & \langle cifra\rangle \\
\langle numero\rangle & ::= & \langle cifra\rangle\ \ \langle numero\rangle \\
\langle cifra\rangle & ::= & 0|1|2|3|4|5|6|7|8|9 \\
\langle addop\rangle & ::= & +|- \\
\langle mulop\rangle & ::= & *|/
\end{array}$$

Il simbolo non terminale $\langle expr\rangle$ è detto simbolo iniziale. I simboli terminali sono 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, (,), +, −, ∗ e /

Si può descrivere l'algoritmo di un parser a discesa ricorsiva nel seguente modo. L'input è costituito da una stringa di *lessemi* (o *token*, corrispondenti ai simboli terminali della grammatica). L'output, nel caso più semplice, è una risposta affermativa se la stringa appartiene al linguaggio generato dalla grammatica, negativa altrimenti. L'algoritmo procede leggendo la stringa e costruendo un albero i cui nodi sono etichettati con simboli terminali e non della grammatica. Inizialmente l'albero è costituito dal solo simbolo iniziale e l'algoritmo è pronto a leggere (diciamo con una testina di lettura) il primo lessema della stringa. L'algoritmo procede ripetendo questi passi, fino a che esiste un nodo (che come vedremo potrà essere solo una foglia) etichettato con simbolo non terminale:

(1) selezionare la foglia più a sinistra tra quelle etichettate con un non terminale; sia N il suo simbolo non terminale.
(2) scegliere una delle regole della grammatica che riscrive N. Nel caso ve ne siano più di una questo è un punto di scelta ed è un potenziale punto di backtracking dell'algoritmo (si veda il successivo passo (4)).
(3) estendere l'albero, in corrispondenza della foglia selezionata al punto (1), utilizzando la regola selezionata al passo (2). Questo passo aggiungerà uno o più nodi foglia come figli del nodo etichettato con N, che quindi non sarà più foglia.
(4) processare da sinistra verso destra le nuove foglie introdotte al passo (3) fino a che se ne incontra una etichettata da un non terminale. Per ogni foglia processata, la cui l'etichetta è un simbolo terminale, verificare che il prossimo lessema della

stringa coincida con tale simbolo. Se è così allora prepararsi a leggere il successivo lessema della stringa (spostando in avanti la testina di lettura) e passare a processare la prossima foglia. Se invece il simbolo non coincide con il lessema attivare il backtracking riprendendo dall'ultimo punto di scelta che presenta una alternativa disponibile (ovvero una diversa scelta della regola della grammatica). Effettuando il backtracking sarà necessario disfare la costruzione (dell'ultima parte) dell'albero e riposizionare la testina di lettura.
Se non vi sono possibilità di backtracking allora dichiarare fallito il riconoscimento della stringa: essa non fa parte del linguaggio.

(5) Se tutti i lessemi della stringa sono stati letti e non vi sono ulteriori foglie etichettate con simboli non terminali, allora il riconoscimento ha avuto successo: la stringa fa parte del linguaggio.

In Prolog è semplice scrivere un programma che faccia il parsing di una lista di lessemi rispetto ad una data grammatica, utilizzando l'algoritmo appena descritto. Il prossimo esempio illustra come per fare ciò si sfruttino opportunamente le difference list.

Esempio 11.2. Ecco un esempio di programma Prolog che realizza un parser per la grammatica dell'Esercizio 11.1:

```
expr(S-S0) :- fattore(S-S1), mulop(S1-S2), expr(S2-S0).
expr(S-S0) :- fattore(S-S0).
fattore(S-S0) :- addendo(S-S1), addop(S1-S2), fattore(S2-S0).
fattore(S-S0) :- addendo(S-S0).
addendo(S-S0) :- opar(S-S1), expr(S1-S2), cpar(S2-S0).
addendo(S-S0) :- numero(S-S0).
numero(S-S0) :- cifra(S-S0).
numero(S-S0) :- cifra(S-S1), numero(S1-S0).
opar(['('|S]-S).          cpar([')'|S]-S).
cifra([0|S]-S).           cifra([1|S]-S).
cifra([2|S]-S).           cifra([3|S]-S).
cifra([4|S]-S).           cifra([5|S]-S).
cifra([6|S]-S).           cifra([7|S]-S).
cifra([8|S]-S).           cifra([9|S]-S).
addop([+|S]-S).           addop([-|S]-S).
mulop([*|S]-S).           mulop([/|S]-S).
```

Per effettuare il parsing della espressione `22+5*3/8` tramite il predicato `expr` è sufficiente istanziarne l'argomento come segue:

```
?- expr([2,2,+,5,*,3,/,8]-[]).
    yes
```

(Chiaramente, in situazioni usuali, la lista dei caratteri che compongono l'espressione potrà essere prodotta spezzando in lessemi la stringa `22+5*3/8`, ottenuta ad esempio come input dall'utente.)

Come si nota, il vantaggio di utilizzare difference list invece di liste usuali, rende possibile effettuare il parsing di una sequenza di lessemi compiendo ad ogni passo una append in tempo costante (come illustrato nella sezione precedente).

Vista quindi l'importanza delle difference list nel parsing, il linguaggio Prolog mette a disposizione una notazione sintattica appositamente introdotta per scrivere regole grammaticali. In tale notazione le difference list vengono lasciate implicite.

Esempio 11.3. Ecco la grammatica dell'Esempio 11.2 (si veda anche l'Esempio 11.1) espressa nella sintassi delle DCG.

```
expr --> fattore, mulop, expr.
expr --> fattore.
fattore --> addendo, addop, fattore.
fattore --> addendo.
addendo --> opar, expr, cpar.
addendo --> numero.
numero --> cifra.
numero --> cifra, numero.
cifra --> [0].      cifra --> [1].      cifra --> [2].
cifra --> [3].      cifra --> [4].      cifra --> [5].
cifra --> [6].      cifra --> [7].      cifra --> [8].
cifra --> [9].
opar --> ['('].     addop --> [+].      mulop --> [*].
cpar --> [')'].     addop --> [-].      mulop --> [/].
```

Quando queste regole vengono consultate/compilate dall'interprete Prolog, esse vengono automaticamente tradotte in una forma equivalente a quella illustrata nell'Esempio 11.2:

```
expr(A,B) :- fattore(A,C), mulop(C,D), expr(D,B).
expr(A,B) :- fattore(A,B).
fattore(A,B) :- addendo(A,B).
fattore(A,B) :- addendo(A,C), addop(C,D), fattore(D,B).
addendo(A,B) :- opar(A,C), expr(C,D), cpar(D,B).
addendo(A,B) :- numero(A,B).
numero(A,B) :- cifra(A,B).
numero(A,B) :- cifra(A,C), numero(C,B).
cifra([0|A],A).          cifra([1|A],A).          cifra([2|A],A).
cifra([3|A],A).          cifra([4|A],A).          cifra([5|A],A).
cifra([6|A],A).          cifra([7|A],A).          cifra([8|A],A).
cifra([9|A],A).
opar(['('|A],A).         addop([+|A],A).          mulop([*|A],A).
cpar([')'|A],A).         addop([-|A],A).          mulop([/|A],A).
```

dove l'unica differenza risiede nel fatto che l'interprete Prolog non impiega un particolare operatore per accoppiare i due termini che definiscono una difference list. Si aggiungono invece ad ogni predicato due argomenti che congiuntamente rappresentano la difference list.

In generale, una volta consultate le regole che definiscono una grammatica, i predicati definiti dall'interprete Prolog consentono non solo di effettuare il parsing di una data espressione; ma, se le regole sono opportunamente formulate, gli stessi predicati possono essere impiegati per generare tutte le stringhe del linguaggio.

Il seguente esempio chiarirà questo punto.

ESEMPIO 11.4. Sia data la DCG così descritta:

```
frase --> soggetto, azione, modo  |  soggetto, transaz, oggetto.
soggetto --> articolo, cosaAnimata.
oggetto --> articolo, cosaInanimata.
articolo --> [la]  |  [una].
cosaAnimata --> [bimba]  |  [gatta].
cosaInanimata --> [palla]  |  [strada].
azione --> [salta]  |  [corre].
transaz --> [scaglia]  |  [percorre].
modo --> []  |  [agilmente]  |  [velocemente].
```

Notiamo innanzitutto la scrittura compatta, tramite l'operatore `|` , di più regole con lo stesso simbolo non terminale nella testa. Osserviamo altresì che un lessema, ad esempio `strada` o `palla`, non deve necessariamente essere costituito da un solo carattere. Una ultima osservazione riguarda le regole che nel corpo presentano il lessema vuoto `[]`. Tali regole sono solitamente dette ϵ-produzioni e, rifacendosi alla descrizione dell'algoritmo di parsing data in predecenza, non comportano la lettura di alcun lessema della stringa di input.

Per utilizzare la grammatica dell'esempio precedente come generatore delle stringhe del linguaggio è sufficiente sottoporre a Prolog il goal `?-frase(F,[]).` ed enumerare le possibili frasi (digitando "`;`"). Ecco ciò che si otterrebbe:

```
?-frase(F,[]).
    yes  F = [la, bimba, salta]  ;
    yes  F = [la, bimba, salta, agilmente]  ;
    yes  F = [la, bimba, salta, velocemente]  ;
    yes  F = [la, bimba, corre]  ;
    yes  F = [la, bimba, corre, agilmente]  ;
    yes  F = [la, bimba, corre, velocemente]  ;
    yes  F = [la, gatta, salta]  ;
    yes  F = [la, gatta, salta, agilmente]  ;
    yes  F = [la, gatta, salta, velocemente]  ;
    yes  F = [la, gatta, corre]  ;
⋮
```

Abbiamo detto che la grammatica può essere impiegata per generare le frasi del linguaggio solo se le sue regole sono "opportunamente formulate". Con ciò intendiamo che, qualora si voglia utilizzare le regole per generare le stringhe, è necessario fare attenzione al modo in cui si effettuano le dichiarazioni ricorsive. Infatti le particolari regole di selezione (di atomi e clausole, si veda Capitolo 5) adottate in Prolog, potrebbero innescare una computazione infinita (anche se né il linguaggio generato né le sue stringhe sono infiniti). Ecco due esempi problematici.

ESEMPIO 11.5. La semplice grammatica:

```
simbInit --> parteA, parteB.
parteA --> parteB, [a].
parteB --> parteA, [b].
```

descrive un linguaggio vuoto (ovvero nessuna stringa di soli terminali a, b appartiene ad esso). Tuttavia il programma Prolog corrispondente origina una ricorsione infinita per qualsiasi richiesta di riconoscimento di una stringa di input. Rifacendosi all'algoritmo di parsing descritto all'inizio, ciò accade perchè l'albero continua ad essere esteso (selezionando sempre la foglia non terminale più a sinistra), ma nessun lessema della stringa viene mai letto. Ad ogni passo si estende l'albero inserendo una nuova foglia non terminale che sarà selezionata al passo successivo.

Esempio 11.6. La semplice grammatica

```
simb --> parteA, simb  |  [a].
parteA --> [].
```

descrive un linguaggio finito costituito dalla unica stringa a. Tuttavia a causa della ϵ-produzione, l'algoritmo di discesa ricorsiva che abbiamo descritto non termina per alcuna richiesta di riconoscimento di una stringa di input.

Una possibile soluzione per evitare questi fenomeni consiste quindi nel riformulare opportunamente le regole della grammatica in modo che non sia possibile innescare una successione infinita estensioni dell'albero, senza che nessun lessema sia letto. La seguente è una riformulazione della grammatica per le espressioni aritmetiche presentata nell'Esempio 11.1:

$$
\begin{array}{rcl}
\langle expr \rangle & ::= & \langle addendo \rangle \ \langle addendi \rangle \\
\langle addendi \rangle & ::= & \langle sumop \rangle \ \langle addendo \rangle \ \langle addendi \rangle \\
\langle addendi \rangle & ::= & \langle difop \rangle \ \langle addendo \rangle \ \langle addendi \rangle \\
\langle addendi \rangle & ::= & \epsilon \\
\langle addendo \rangle & ::= & \langle fattore \rangle \ \langle fattori \rangle \\
\langle fattori \rangle & ::= & \langle mulop \rangle \ \langle fattore \rangle \ \langle fattori \rangle \\
\langle fattori \rangle & ::= & \langle divop \rangle \ \langle fattore \rangle \ \langle fattori \rangle \\
\langle fattori \rangle & ::= & \epsilon \\
\langle fattore \rangle & ::= & \langle cifra \rangle \ \langle cifre \rangle \ \mid \ (\ \langle expr \rangle \ ) \\
\langle cifre \rangle & ::= & \langle cifra \rangle \ \mid \ \epsilon \\
\langle cifra \rangle & ::= & 0|1|2|3|4|5|6|7|8|9 \\
\langle sumop \rangle & ::= & + \\
\langle difop \rangle & ::= & - \\
\langle mulop \rangle & ::= & * \\
\langle divop \rangle & ::= & /
\end{array}
$$

dove ϵ denota come detto la stringa vuota. Utilizzeremo questa grammatica nella prossima sezione.

Concludiamo questa sezione menzionando il fatto che Prolog metta a disposizione il predicato predefinito phrase utilizzabile nell'effettuare il parsing di una lista di lessemi, rispetto ad un simbolo non terminale, senza esplicitare le componenti della difference list. La sintassi base è:

```
phrase(NonTerminale, Lista)
```

Il mode è (+,?). L'atomo sarà vero se la lista di lessemi Lista può essere unificata con una frase del linguaggio generato dal simbolo non terminale NonTerminale (quest'ultimo deve essere fornito). Una versione alternativa del predicato phrase è invece la seguente:

```
phrase(NonTerminale, Lista, Resto)
```

In questo caso il mode è (+,?,?). Il predicato è verificato se un prefisso di `Lista` corrisponde a una frase del linguaggio e il resto di `Lista` unifica con `Resto`.

3. Alcune estensioni

Analizziamo ora alcune estensioni delle DCG. Lo faremo sfruttando i programmi descritti nella sezione precedente.

La prima estensione consiste nell'inserire degli argomenti ausiliari nelle regole della grammatica. Dopo la traduzione effettuata in fase di compilazione/consultazione dall'interprete Prolog, questi argomenti ausiliari figureranno nelle clausole Prolog prodotte dall'interprete, assieme ai due argomenti introdotti per modellare le difference list.

Supponiamo di voler costruire, durante il parsing, l'albero sintattico della stringa analizzata. Consideriamo a questo scopo la grammatica per le espressioni aritmetiche illustrata a pag.156. La sua formulazione tramite DCG è la seguente:

```
expr(esp(E)) --> addendo(Psn), addendi(Psn,E).
addendi(Psn,P) --> sumop, addendo(Pdx), addendi(sum(Psn,Pdx),P).
addendi(Psn,P) --> difop, addendo(Pdx), addendi(dif(Psn,Pdx),P).
addendi(P,P) --> [].
addendo(P) --> fattore(Psn), fattori(Psn,P).
fattori(Psn,P) --> mulop, fattore(Pdx), fattori(mul(Psn,Pdx),P).
fattori(Psn,P) --> divop, fattore(Pdx), fattori(div(Psn,Pdx),P).
fattori(P,P) --> [].
fattore(P) --> ['('], expr(P), [')'].
fattore(P) --> cifra(Psn), cifre(Psn,P).
cifre(Cs,P) --> cifra(C), cifre(cc(Cs,C),P).
cifre(P,P) --> [].
cifra(c(0)) --> [0].      cifra(c(1)) --> [1].      cifra(c(2)) --> [2].
cifra(c(3)) --> [3].      cifra(c(4)) --> [4].      cifra(c(5)) --> [5].
cifra(c(6)) --> [6].      cifra(c(7)) --> [7].      cifra(c(8)) --> [8].
cifra(c(9)) --> [9].
sumop --> [+].      difop --> [-].      mulop --> [*].      divop --> [/].
```

Si noti l'impiego di argomenti supplementari per gestire la costruzione del termine Prolog. Questa grammatica verrà convertita dall'interprete nelle clausole:

```
expr(e(A),B,C) :- addendo(D,B,E), addendi(D,A,E,C).
addendi(A,B,C,D) :- sumop(C,E), addendo(F,E,G), addendi(sum(A,F),B,G,D).
addendi(A,B,C,D) :- difop(C,E), addendo(F,E,G), addendi(dif(A,F),B,G,D).
addendi(A,A,B,B).
addendo(A,B,C) :- fattore(D,B,E), fattori(D,A,E,C).
⋮
```

Dove i due argomenti più a destra di ogni predicato corrispondono alle due componenti della difference list. Conseguentemente, al goal

```
?- expr(T, [2,2,+,5,*,'(',3,/,8,')'], []).
```

o, equivalentemente, al goal

```
?- phrase(expr(T), [2,2,+,5,*,'(',3,/,8,')'], []).
```

otterremo come risposta

```
yes    T = e(sum(cc(c(2), c(2)), mul(c(5), e(div(c(3), c(8))))))
```

Illustriamo ora una ulteriore funzionalità offerta da Prolog nel trattamento delle DCG. Essa consiste nella possibilità di inserire nel corpo delle regole grammaticali ordinari predicati Prolog. Così facendo, per poter riconoscere una sequenza di lessemi come appartenente al linguaggio, l'interprete Prolog non solo deve verificare che i lessemi rispettino le regole grammaticali, ma che anche i goal Prolog supplementari siano soddisfatti.

I goal Prolog sono inseriti nelle regole racchiudendoli tra parentesi graffe. L'ordine in cui gli atomi vengono processati durante la procedura di risoluzione è quello usuale del Prolog.

Questa possibilità di arricchire le regole grammaticali permette principalmente sia di valutare condizioni supplementari per il riconoscimento di una stringa, sia di effettuare una compilazione della stringa. Questa compilazione può consistere sia in una traduzione vera e propria sia, più semplicemente, in una valutazione come nel prossimo esempio.

Esempio 11.7. Ecco la grammatica sopra descritta arricchita con i goal necessari per valutare l'espressione aritmetica:

```
valuta(Val,Lessemi) :- expr(Val,Lessemi,[]),!.
expr(E) --> addendo(A), addendi(A,E).
addendi(V1,R) --> sumop, addendo(A), {S is V1 + A}, addendi(S,R).
addendi(V1,R) --> difop, addendo(A), {D is V1 - A}, addendi(D,R).
addendi(P,P) --> [].
addendo(V) --> fattore(V1), fattori(V1,V).
fattori(V1,R) --> mulop, fattore(V), {M is V1 * V}, fattori(M,R).
fattori(V1,R) --> divop, fattore(V), {Q is V1 / V}, fattori(Q,R).
fattori(P,P) --> [].
fattore(E) --> ['('], expr(E), [')'].
fattore(V) --> cifra(Psn), cifre(Psn,V).
cifre(Cs,P) --> cifra(C), {V is (Cs * 10) + C}, cifre(V,P).
cifre(P,P) --> [].
cifra(0) --> [0].      cifra(1) --> [1].      cifra(2) --> [2].
cifra(3) --> [3].      cifra(4) --> [4].      cifra(5) --> [5].
cifra(6) --> [6].      cifra(7) --> [7].      cifra(8) --> [8].
cifra(9) --> [9].
sumop --> [+].     difop --> [-].     mulop --> [*].     divop --> [/].
```

Quando queste regole vengono consultate/compilate dall'interprete Prolog, esse vengono tradotte come:

```
valuta(A,B) :- expr(A,B,[]), !.
expr(A,B,C) :- addendo(D,B,E), addendi(D,A,E,C).
addendi(A,B,C,D) :- sumop(C,E), addendo(F,E,G),
                    H is A+F, I=G, addendi(H,B,I,D).
addendi(A,B,C,D) :- difop(C,E), addendo(F,E,G),
                    H is A-F, I=G, addendi(H,B,I,D).
addendi(A,A,B,B).
addendo(A,B,C) :- fattore(D,B,E), fattori(D,A,E,C).
⋮
```

Dove, come in precedenza, i due argomenti più a destra di ogni predicato corrispondono alle due componenti della difference list. Se sottoponessimo il goal

```
?- valuta(Val,[7,+,5,*,'(','(',3,+,2,')',/,1,0,')']).
```

otterremmo la risposta

```
yes   Val = 9.5.
```

4. Esercizi

Esercizio 11.1. Si modifichi la grammatica dell'Esempio 11.7 in modo da utilizzare una sola regola per tutte le cifre, in luogo delle dieci regole elencate. [SUGGERIMENTO: si scriva una regola parametrizzata dalla cifra e si utilizzi un predicato Prolog che determini se un carattere è una cifra.]

Esercizio 11.2. Modificare la DCG dell'Esempio 11.7 in modo da utilizzare un solo simbolo non terminale ⟨*muldivop*⟩ (e quindi una sola regola) per le operazioni di prodotto e divisione. Analogamente si operi per la somma e la differenza. [OSSERVAZIONE: a seguito di questa modifica, dovranno restare solo due regole per il non terminale ⟨*fattori*⟩. La prima di esse gestirà entrambe le operazioni di prodotto e divisione (similmente per ⟨*addendi*⟩).]

Esercizio 11.3. Descrivere quali *frasi* (sequenze di lessemi) sono riconosciute da questa DCG dove il simbolo iniziale è `s`:

```
s --> ['('], s0, [')'].
s0 --> ['1'] | s1, ['+'], s1.
s1 --> ['1'], ['+'], s2.
s2 --> ['1'] | s1.
```

Esercizio 11.4. Completare il programma dell'Esercizio 11.3 in modo che durante il parsing di una stringa venga calcolato il valore della corrispondente espressione.

Esercizio 11.5. Scrivere una grammatica definita che riconosca i numeri interi con segno, ovvero le sequenze di cifre che non iniziano per '0' e che possono o meno essere precedute da un segno.

Esercizio 11.6. Completare il programma dell'Esercizio 11.5 in modo che durante il parsing di una stringa venga calcolato il numero di cifre che la compongono, trascurando l'eventuale segno.

CAPITOLO 12

Answer set programming

Abbiamo studiato nel Capitolo 8 diverse semantiche tradizionalmente introdotte per gestire la negazione nei programmi logici. Questo capitolo illustra un approccio alternativo, l'*answer set programming* (brevemente, ASP), non solo al trattamento della negazione, ma più in generale a tutta la programmazione dichiarativa. L'answer set programming, come vedremo, si differenzia in molteplici punti dalla programmazione Prolog.

Inizieremo la trattazione descrivendo una forma semplificata di ASP. Nonostante le semplificazioni che adotteremo, vedremo che questo stile di programmazione dichiarativa risulta sufficientemente espressivo da essere proficuamente utilizzabile in diversi contesti.

1. Regole e programmi ASP

Ci rifacciamo in quanto segue alle nozioni di alfabeto, termine, letterale, ecc. introdotte nel Capitolo 2.

Definizione 12.1. Una *regola (ASP)* è una clausola della seguente forma:

$$L_0 \leftarrow L_1, \ldots, L_m, not L_{m+1}, \ldots, not L_n.$$

dove ogni L_i è una formula atomica.
Un *vincolo (ASP)* è una clausola della seguente forma:

$$\leftarrow L_1, \ldots, L_m, not L_{m+1}, \ldots, not L_n.$$

la cui semantica logica è la disgiunzione: $\neg L_1 \vee \cdots \vee \neg L_m \vee L_{m+1} \vee \cdots \vee L_n$.
Un *programma* ASP è un insieme di regole ASP. Spesso considereremo insiemi di regole e di vincoli: un *codice* ASP è un insieme di regole e vincoli ASP.[1]

Nella precedente definizione il connettivo *not* denota la negazione per fallimento finito (si veda la Sezione 1.2 del Capitolo 8). Diremo che un letterale del tipo $not L$ è un *naf-literal*.

Nota 12.1. Un programma ASP ammette sempre un modello. Ad esempio, la base di Herbrand, rendendo vere tutte le teste delle regole, è un modello del programma.

Ciò non è sempre vero per un generico codice ASP. Si pensi ad esempio al codice ASP $\{p. \quad \leftarrow p\}$ costituito da una regola e un vincolo. Questo codice è logicamente equivalente alla congiunzione $p \wedge \neg p$ che è chiaramente insoddisfacibile.

Dalla Definizione 12.1 sembrerebbe che l'ASP non sia poi molto diverso dal Prolog. Tuttavia ciò che differenzia i due approcci è, da un lato il modo in cui viene assegnata

[1] In letteratura quelli che abbiamo appena chiamato codici sono detti semplicemente programmi ASP. La ragione starà nel fatto che, relativamente alla ricerca di modelli stabili, vincoli e regole sono equivalenti—si veda l'Esempio 12.5.

la semantica ad un programma, dall'altro la procedura impiegata per trovare le soluzioni. Vediamo in modo informale una serie di punti in cui i due approcci si differenziano:

- in una regola ASP l'ordine dei letterali non ha alcuna importanza. Ciò in contrasto con le convenzioni adottate in Prolog. Abbiamo infatti visto come il comportamento dell'interprete Prolog, a causa della sua regola di selezione, sia fortemente influenzato dall'ordine in cui i letterali occorrono nelle clausole. In ASP quindi il corpo di una regola si può considerare come un vero e proprio insieme di letterali.
- In Prolog l'esecuzione di un goal avviene in modo *top-down* e *goal-directed*: a partire dal goal si procede utilizzando le clausole e costruendo una derivazione che porta alla soluzione/risposta. Contrariamente a ciò, solitamente un interprete per l'ASP opera *bottom-up*, partendo dai fatti si procede verso le conclusioni che portano alla risposta.
- La procedura risolutiva solitamente implementata in Prolog (SLD-risoluzione), con le sue particolari scelte relative alle regole di selezione di letterali (leftmost) e clausole (dall'alto al basso), può causare la generazione di computazioni infinite (anche in assenza della negazione). Come vedremo ciò non accade per gli ASP-solver.
- Il costrutto extra-logico CUT non è presente in ASP.
- Problemi che si verificano in Prolog, quali il *floundering* (Definizione 8.3) o la generazione di computazioni infinite nella gestione della negazione per fallimento finito, non sono presenti in ASP.
- Come vedremo, i programmi logici trattabili con gli attuali ASP-solver devono rispettare delle restrizioni sull'impiego delle variabili.

2. Una semantica alternativa per la negazione

Le differenze tra Prolog e ASP nascono essenzialmente dal modo in cui viene assegnata la semantica ai programmi. Fin dall'inizio (Definizione 12.1) abbiamo ammesso la presenza nelle regole della negazione per fallimento finito. Possiamo dire che l'ASP sia una forma di programmazione dichiarativa in cui si fissa la semantica dei programmi ad essere la *answer set semantics*. Ciò in contrasto con lo stile di programmazione logica che abbiamo studiato nei capitolo precedenti, dove il significato di un programma (in particolare in presenza di negazione) può essere determinato in base a differenti semantiche (Capitolo 8).

Per descrivere la semantica di un ASP dobbiamo prima introdurre la nozione di *answer set*[2] Innanzitutto affrontiamo il caso più semplice dei programmi ground (ovvero di programmi privi di variabili).

Definizione 12.2. Se P è un programma ASP, allora $ground(P)$ è l'insieme di tutte le istanze ground di regole di P ($ground(P)$ si ottiene sostituendo, in tutti i modi possibili, alle variabili in P gli elementi dell'universo di Herbrand $\mathcal{H}_P$).

Innanzitutto, gli answer set di un generico programma ASP P sono gli stessi answer set di $ground(P)$. Pertanto, nel seguito, se non differentemente specificato, ci limiteremo a studiare il caso di programmi ground (i quali possono anche essere di lunghezza infinita). Gli *answer set* di un programma P vanno ricercati tra i sottoinsiemi *minimali* di $\mathcal{B}_P$ che siano modello di $ground(P)$.

[2] Si noti che la nozione di answer set coincide, per quanto riguarda gli scopi di questo testo, con quella di *stable model* (o *modello stabile*), frequentemente impiegata in relazione allo studio di ASP.

Un programma ASP privo di naf-literal è un programma definito e quindi ha un unico modello minimale (il modello minimo). Risulta naturale allora stabilire che un programma definito ha sempre un unico *answer set* che coincide con il suo modello minimo.

Abbiamo visto nel Capitolo 8 che in presenza di negazione non esiste in generale un unico modello minimo, ma più modelli minimali. Vedremo però che non tutti i modelli minimali sono answer set. Prima di giustificare formalmente questa affermazione, chiariamone intuitivamente il senso con un semplice esempio.

Esempio 12.1. Consideriamo il programma P

```
p :- not q.
```

Sappiamo che P ha due modelli minimali: $\{p\}$ e $\{q\}$. Tuttavia, osserviamo che il modello $\{q\}$ non riflette il significato intuitivo che attribuiamo al programma P (ovvero "se si dimostra che q è falso allora vale p"). Infatti in P non vi è nessuna informazione che fornisca una giustificazione a sostegno della verità di q.

Ricordiamo inoltre che non possiamo fare uso dell'operatore di conseguenza immediata. Esso (o meglio la sua naturale estensione ai programmi con negazione descritta nel Capitolo 8) infatti non è monotono in presenza di negazione. Si pensi all'esempio appena visto, per il quale $T_P(\emptyset) = \{p\}$, ma $T_P(T_P(\emptyset)) = T_P(\{q\}) = \emptyset$.

L'idea base per dichiarare che un insieme di atomi S è answer set di un programma P (quando P contiene naf-literal), consiste nel trasformare P in un programma P^S, ad esso affine ma privo di negazione, e dimostrare che S è il modello minimo (ovvero l'unico answer set) di P^S:

Definizione 12.3 (Answer Set). Sia P un programma ASP. Sia S un insieme di atomi. Definiamo il programma P^S, detto *ridotto di P rispetto a S*, come il programma ottenuto da P come segue:

(1) rimuovendo ogni regola il cui corpo contiene un naf-literal $notL$ tale che $L \in S$;
(2) rimuovendo tutti i naf-literal dai corpi delle restanti regole.

Per costruzione P^S non contiene alcun naf-literal. Quindi ha un unico answer set. Se tale answer set coincide con S, allora S è un answer set per P.

Si osservi che se P è un programma di clausole definite (ovvero privo di naf-literal) allora qualunque sia S, $P^S = P$. Pertanto, S è answer set di P se e solo se S è il modello minimo di P.

Esempio 12.2. Consideriamo il programma P

```
p :- a.
a :- not b.
b :- not a.
```

I candidati answer set per P sono certamente sottoinsiemi della base di Herbrand di P:

$$\mathcal{B}_P = \{a, b, p\}.$$

Analizziamoli uno ad uno:

$\emptyset$: Abbiamo che $P^\emptyset = \{p \leftarrow a. \;\; a. \;\; b.\}$ ma $\emptyset$ non è answer set per $P^\emptyset$. Quindi $\emptyset$ non è un answer set di P.

$\{a\}$: Abbiamo che $P^{\{a\}} = \{p \leftarrow a.\;\; a.\}$ ma $\{a\}$ non è answer set per $P^{\{a\}}$. Quindi neanche $\{a\}$ è un answer set di P.

$\{b\}$: Abbiamo che $P^{\{b\}} = \{p \leftarrow a.\;\; b.\}$ e $\{b\}$ è l'answer set di $P^{\{b\}}$ (ovvero, è il modello minimo). Quindi $\{b\}$ è un answer set di P.

$\{p\}$: Abbiamo che $P^{\{p\}} = \{p \leftarrow a.\;\; a.\;\; b.\}$ ma $\{p\}$ non è answer set per $P^{\{p\}}$. Quindi neanche $\{p\}$ è un answer set di P.

$\{p, a\}$: Abbiamo che $P^{\{p,a\}} = \{p \leftarrow a.\;\; a.\}$ e $\{p, a\}$ è l'answer set di $P^{\{p,a\}}$. Quindi $\{p, a\}$ è un answer set di P.

Gli insiemi $\{a, b\}$, $\{b, p\}$ e $\{a, b, p\}$, non sono answer set di P perchè includono propriamente un answer set (ad esempio $\{b\}$). Quindi P ha due answer set distinti: $\{b\}$ e $\{p, a\}$.

Esempio 12.3. Consideriamo il programma P

```
a :- not b.
b :- not c.
d.
```

L'insieme $S_1 = \{b, d\}$ è un answer set di P. Infatti $P^{S_1} = \{b.\;\; d.\}$ che ha S_1 come answer set. Invece l'insieme $S_2 = \{a, d\}$ non è un answer set di P. Infatti $P^{S_2} = \{a.\;\; b.\;\; d.\}$ che non ha S_2 come answer set.

Esempio 12.4. Si consideri il seguente programma P

```
p :- p.
q.
```

L'unico answer set di P è l'insieme $\{q\}$.

Il seguente esempio mette in luce una caratteristica fondamentale dell'ASP.

Esempio 12.5. Consideriamo il programma P

```
p :- not p,  d.
d.
```

Questo programma ammette il modello $\{p, d\}$. Tuttavia, non ha alcun answer set. Ci si può rendere conto di questo osservando che un qualsiasi modello di P deve contenere d. Quindi abbiamo due possibili candidati ad essere answer set:

- $S_1 = \{d\}$: allora $P^{S_1} = \{p \leftarrow d.\;\; d.\}$ ha come modello minimo $\{d, p\}$ che è diverso da S_1. Quindi S_1 non è answer set per P.
- $S_2 = \{d, p\}$: allora $P^{S_2} = \{\; d.\}$ ha come answer set $\{d\}$ che è diverso da S_2. Quindi S_2 non è answer set per P.

Pensando al corpo di una regola ASP come ad una giustificazione per supportare la verità della testa della regola, allora l'idea intuitiva impiegata nel decidere se un modello sia o meno un answer set è la seguente: *"Un qualsiasi p è presente nell'answer set solo se è forzato ad esserlo in quanto testa di una regola con corpo vero. Tuttavia, la verità della testa p di una regola non può essere giustificata in base alla verità di $not\, p$ nel corpo."*

Applichiamo questo criterio al programma precedente. La seconda regola impone la verità di d, inoltre l'unico modo per giustificare la presenza di p in un answer set è che il corpo della prima clausola sia vero, ma ciò accade solo se è vero $not\, p$. Quindi la prima regola non può

supportare la verità di p. (Si noti che la verità di p potrebbe comunque essere supportata da una altra regola, come accade nel programma P_2 che analizzeremo tra poco.)

Dal punto di vista della ricerca di answer set, pertanto, la regola

```
p :- not p, d.
```

è equivalente ad imporre che d sia falso. Vedremo nella Sezione 3.1 che ciò corrisponde ad utilizzare il vincolo

```
:- d
```

Un ragionamento analogo può essere fatto per il seguente programma P_1:

```
p :- not p,  d.
r :- not d.
d :- not r.
```

Se prendessimo in considerazione solamente le ultime due regole allora avremmo due answer set: $\{d\}$ e $\{r\}$. Tuttavia la presenza della prima regola ha come effetto di invalidare il primo di questi modelli. Quindi P_1 ha $\{r\}$ come unico answer set. Chiariamo ulteriormente questo fenomeno tramite il seguente programma P_2 simile al precedente:

```
p :- not p.
p :- not d.
r :- not d.
d :- not r.
```

Il solo answer set è $\{r, p\}$. Analizzando P_2 si può comprendere meglio come le due regole coinvolgenti p invalidino ogni candidato answer set che contenga $\{d\}$. Come detto, la presenza delle ultime due clausole forza uno (e uno solo) tra d e r ad appartenere ad ogni answer set. Consideriamo un qualsiasi insieme S di atomi candidato ad essere answer set per P_2, e tale che d non vi appartenga (quindi r vi appartiene). Il fatto che d sia falso in S forza p ad essere vero per la seconda regola di P_2. Quindi affinchè S sia answer set deve contenere anche p. Ciò, per la Definizione 12.3, rende inefficace la clausola $p \leftarrow \mathit{not}\, p$. nella costruzione del ridotto P_2^S. Quindi $\{r, p\}$ è answer set di P_2. Sia ora S' un qualsiasi insieme di atomi che contenga d. Per ogni insieme di questo tipo, a causa della presenza di d, la seconda regola di P_2 non darà contributo nella costruzione di $P_2^{S'}$. Sussistono due possibilità alternative: p appartiene o meno a S'. Se p appartiene a S' allora il ridotto $P_2^{S'}$ ha come answer set $\{d\} \neq S'$ (in questo caso S' è modello di P_2 ma non è minimale). Altrimenti, se p non appartiene a S', nel ridotto $P_2^{S'}$ compare la regola p.. Ciò impone, affinchè S' sia answer set, che anche p appartenga a S', contrariamente all'assunzione. Neanche in questo caso S' è answer set.

Dalla analisi di P_2 si evince quindi un modo per descrivere, tramite alcune regole del programma, una collezione di candidati answer set e invalidare una parte di questi tramite dei vincoli imposti da ulteriori regole. Studieremo meglio questa tecnica di programmazione nelle prossime sezioni.

Esercizio 12.1. Si trovino tutti gli answer set del programma composto dalle n regole della forma:

$$\mathtt{p}_i \ \texttt{:- not } \mathtt{p}_{i+1}.$$

con $1 \leqslant i \leqslant n$.

Esercizio 12.2. Si trovino tutti gli answer set del seguente programma:

```
p :- not q.
q :- not p.
r :- p.
r :- q.
```

Esercizio 12.3. Si trovino tutti gli answer set del seguente programma:

```
p :- not q.
q :- not p.
r :- not r.
r :- p.
```

Il prossimo esempio (tratto da [**Bar04**]) richiama un problema noto nella letteratura sull'intelligenza artificiale: come sia possibile modellare il fatto che gli uccelli volino e al contempo che i pinguini non siano in grado di farlo.

Esempio 12.6. Il programma P è costituito dalle seguenti regole di facile lettura dichiarativa:

```
vola(X) :- uccello(X), not anormale(X).
anormale(X) :- pinguino(X).
uccello(X) :- pinguino(X).
uccello(tweety).
pinguino(skippy).
```

Per determinare quale sia la semantica che viene associata a questo programma, determiniamo quali siano i suoi answer set. Per fare ciò dobbiamo prima costruire $P' = \mathit{ground}(P)$, tenendo presente che l'universo di Herbrand di P è {`tweety, skippy`}. P' è:

```
vola(tweety) :- uccello(tweety), not anormale(tweety).
vola(skippy) :- uccello(skippy), not anormale(skippy).
anormale(tweety) :- pinguino(tweety).
anormale(skippy) :- pinguino(skippy).
uccello(tweety) :- pinguino(tweety).
uccello(skippy) :- pinguino(skippy).
uccello(tweety).
pinguino(skippy).
```

Osservando P' notiamo che sia `uccello(tweety)` che `pinguino(skippy)` devono appartenere a qualsiasi answer set. Conseguentemente, anche `uccello(skippy)` e `anormale(skippy)` devono appartenere a qualsiasi answer set. Consideriamo ad esempio l'insieme

$$\begin{aligned} S \;=\; \{&\texttt{uccello(tweety), pinguino(skippy),}\\ &\texttt{uccello(skippy), anormale(skippy), vola(tweety)}\}. \end{aligned}$$

Verifichiamo che S è un answer set di P calcolando il minimo punto fisso dell'operatore T_{P^S}. Il ridotto P^S è:

```
vola(tweety) :- uccello(tweety).
anormale(tweety) :- pinguino(tweety).
anormale(skippy) :- pinguino(skippy).
uccello(tweety) :- pinguino(tweety).
uccello(skippy) :- pinguino(skippy).
uccello(tweety).
pinguino(skippy).
```

Utilizziamo ora l'operatore di conseguenza immediata. Dopo tre applicazioni otteniamo il punto fisso $T_{P^S}\uparrow\omega = T_{P^S}\uparrow 3 = S$. Quindi S è answer set di P.

I seguenti risultati correlano formalmente answer set e modelli (di Herbrand) minimali:

TEOREMA 12.1. *Dato un programma P, ogni answer set di P è un modello minimale di P.*

PROPOSIZIONE 12.1. *Sia P un programma e sia S un insieme di atomi. Allora S è answer set di P se e solo se*

- *S è modello di P; e*
- *per ogni S', se S' è modello di P^S allora $S \subseteq S'$.*

3. Tecniche di programmazione in ASP

In questa sezione illustriamo come l'uso di particolari regole ASP permetta di implementare semplici funzionalità. Le tecniche illustrate potranno essere poi impiegate nella realizzazione di programmi/codici più complessi.

Abbiamo visto (Sezione 2, pag. 165) che regole di particolare forma possono essere impiegate per escludere dei modelli dall'insieme degli answer set di un programma. Come vedremo ora, questa è una vera e propria tecnica di programmazione.

3.1. Vincoli di integrità. I vincoli di integrità (o integrity constraint) sono regole prive di testa:

$$\leftarrow L_1, \ldots, L_m, not L_{m+1}, \ldots, not L_n.$$

Una tale regola ha l'effetto di invalidare ogni answer set che ne soddisfi il corpo, ovvero ogni insieme di atomi che contenga tutti gli atomi $L_1, \ldots, L_m$ e nessuno degli $L_{m+1}, \ldots, L_n$.

ESEMPIO 12.7. Si consideri il programma

```
a :- not b.
b :- not a.
:- a.
```

Dei due candidati answer set determinati dalle prime due regole, $\{a\}$ e $\{b\}$, ma la terza regola invalida ogni answer set che contenga a (nel caso, $\{a\}$).

Ricordando anche quanto osservato nell'Esempio 12.5, sappiamo quindi che l'aggiungere il vincolo `:- a.` equivale ad aggiungere una regola del tipo `p :- not p, a.` (dove `p` è un nuovo predicato). Possiamo quindi asserire che, relativamente alla ricerca di answer set, i vincoli sono un'estensione puramente sintattica del linguaggio. Tuttavia, è importante

osservare che scrivendo un programma ASP risulta oltremodo naturale utilizzare sia regole che vincoli (ottenendo dunque *codici* anziché *programmi*).

I vincoli, come vedremo, permettono di descrivere in modo compatto proprietà ricorsive. D'altro canto, un codice che non sia programma potrebbe non ammettere alcun answer set (si pensi al semplice codice: {`p.    :-p`}). In virtù di questa osservazione, nel resto del testo parleremo in generale di programmi ASP identificando con questo termine i concetti di programmi ASP e codici ASP.

3.2. Enumerazione finita. Consideriamo il seguente programma

```
  a :- not n_a.
n_a :- not a.
  b :- not n_b.
n_b :- not b.
  c :- not n_c.
n_c :- not c.
  d :- not n_d.
n_d :- not d.
```

Gli answer set di questo programma rappresentano tutti i modi possibili di scegliere una e una sola alternativa tra `a` e `n_a`, una tra `b` e `n_b`, una tra `c` e `n_c`, e una tra `d` e `n_d`. Ovviamente, la situazione può essere generalizzata ad un numero arbitrario di predicati.

Questa tecnica può essere usata per mostrare una delle proprietà più importanti dell'Answer Set Programming:

TEOREMA 12.2. *Il problema di stabilire se un programma (o un codice) ground P ammette modelli stabili è NP-completo.*

DIM. Dato P ground, un possibile modello stabile S conterrà necessariamente solo atomi presenti in P. Dunque $|S| \leq |P|$. Per verificare se S sia o meno modello stabile è sufficiente calcolare P^S e il modello minimo dello stesso. Entrambe le operazioni si effettuano in tempo polinomiale su $|P|$. Dunque il problema appartiene ad NP.

Per mostrare la completezza, si consideri un'istanza di 3SAT, ad esempio:

$$\underbrace{(A \vee \neg B \vee C)}_{\texttt{c1}} \wedge \underbrace{(\neg A \vee B \vee \neg C)}_{\texttt{c2}}$$

e si costruisca il seguente programma:

```
a :- not na.
na :- not a.
b :- not nb.
nb :- not b.
c :- not nc.
nc :- not c.
```

Queste clausole garantiscono la presenza di esattamente uno tra `a` (A) e `na` ($\neg A$) in ogni modello stabile. Similmente per B e C. Sono inserite pertanto due clausole ogni variabile dell'istanza di SAT. Per ogni clausola dell'istanza di 3SAT si aggiungono le 3 clausole che la definiscono:

```
c1 :- a.
c1 :- nb.
c1 :- c.
c2 :- na.
c2 :- b.
c2 :- nc.
```

Un ultimo passo si compie richiedendo che tutte le clausole dell'istanza di 3SAT (in questo caso `c1` e `c2`) siano verificate. Ciò può essere fatto tramite due vincoli (che invalidano tutti gli answer set in cui almeno uno tra `c1` e `c2` è falso):

```
:- not c1.
:- not c2.
```

E' immediato verificare che il programma ottenuto ha un modello stabile se e solo se l'istanza di 3SAT è soddisfacibile. □

3.3. Enumerazione generica. Supponiamo di voler realizzare un programma i cui answer set contengano almeno un atomo (o più di uno) scelto/i tra un determinato insieme di possibilità. Il seguente schema di programma può essere impiegato per questo scopo:

```
scelto(X) :- possibile(X), not non_scelto(X).
non_scelto(X) :- possibile(X), not scelto(X).
p :- scelto(X).
:- not p.
```

Se ad esempio, al precedente programma aggiungiamo le regole

```
possibile(a).
possibile(b).
possibile(c).
```

otteniamo un programma i cui answer set sono:

$S \cup$ {scelto(a), scelto(b), scelto(c)}
$S \cup$ {scelto(a), scelto(b), non_scelto(c)}
$S \cup$ {scelto(a), scelto(c), non_scelto(b)}
$S \cup$ {scelto(b), scelto(c), non_scelto(a)}
$S \cup$ {scelto(a), non_scelto(b), non_scelto(c)}
$S \cup$ {scelto(b), non_scelto(a), non_scelto(c)}
$S \cup$ {scelto(c), non_scelto(a), non_scelto(b)}

dove $S =${p, possibile(a), possibile(b), possibile(c)}.

3.4. Scelta univoca. Una variazione del programma precedente permette di selezionare solamente gli answer set in cui una ed una sola delle possibili alternative è vera. Lo schema base è dato dal seguente programma P:[3]

[3]Come vedremo quando nella Sezione 4 descriveremo l'ASP-solver Smodels (ma lo stesso dicasi per tutti gli ASP-solver che utilizzano Lparse come front-end), "`!=`" denota la disuguaglianza sintattica. L'uguaglianza sintattica è invece denotata, sempre in Lparse/Smodels, con "`==`". Il simbolo "`=`" è invece riservato per una forma di assegnamento (si veda la descrizione delle funzioni built-in riportata a pag. 179).

```
differente_da_scelto(X) :- scelto(Y), X!=Y.
scelto(X) :- possibile(X), not differente_da_scelto(X).
```

Le seguenti regole elencano le varie alternative:

```
possibile(a).
possibile(b).
possibile(c).
```

Gli answer set sono quindi:

$S \cup$ {scelto(a), differente_da_scelto(b), differente_da_scelto(c)}
$S \cup$ {scelto(b), differente_da_scelto(a), differente_da_scelto(c)}
$S \cup$ {scelto(c), differente_da_scelto(a), differente_da_scelto(b)}

dove $S =$ {possibile(a), possibile(b), possibile(c)}.

3.5. Enumerazione vincolata. Il seguente schema di programma generalizza lo schema di scelta univoca appena illustrato.

```
non_scelto(X,Y) :- scelto(Z,Y), Z!=X.
non_scelto(X,Y) :- scelto(X,Z), Z!=Y.
scelto(X,Y) :- possibile_1(X), possibile_2(Y), not non_scelto(X,Y).
```

In questo caso, aggiungendo opportune regole per i predicati `possibile_1` e `possibile_1`, ogni answer set conterrà un insieme di atomi `scelto(a,b)` tale che ogni valore `X`, per cui `possibile_1(X)`, comparirà una ed una sola volta come primo argomento e similmente per `possibile_2(Y)`. Ad esempio completando il programma con

```
possibile(1).
possibile(2).
possibile(3).
possibile1(X):-possibile(X).
possibile2(X):-possibile(X).
```

otterremo sei diversi answer set:

```
{scelto(1,3), scelto(2,2), scelto(3,1), possibile(1), ...}
{scelto(1,3), scelto(3,2), scelto(2,1), possibile(1), ...}
{scelto(2,3), scelto(1,2), scelto(3,1), possibile(1), ...}
{scelto(2,3), scelto(3,2), scelto(1,1), possibile(1), ...}
{scelto(3,3), scelto(1,2), scelto(2,1), possibile(1), ...}
{scelto(3,3), scelto(2,2), scelto(1,1), possibile(1), ...}
```

Ovviamente, la tecnica può essere generalizzata ad un numero arbitrario di argomenti.

3.6. Operare con ordinamenti. Supponiamo sia dato un insieme di oggetti ed un ordine lineare su esso. Assumiamo che l'ordine sia rappresentato dal predicato `minore`. Il seguente programma può essere impiegato per determinare il minimo e il massimo elemento dell'insieme. Inoltre è possibile anche determinare per ogni oggetto chi sia il prossimo oggetto nell'ordine.

```
non_il_piu_piccolo(X) :- oggetto(X), oggetto(Y), minore(Y,X).
il_piu_piccolo(X) :- oggetto(X), not non_il_piu_piccolo(X).

non_il_piu_grande(X) :- oggetto(X), oggetto(Y), minore(X,Y).
il_piu_grande(X) :- oggetto(X), not non_il_piu_grande(X).

non_il_prossimo(X,Y) :- X==Y.
non_il_prossimo(X,Y) :- minore(Y,X).
non_il_prossimo(X,Y) :- oggetto(X), oggetto(Y), oggetto(Z),
                        minore(X,Z), minore(Z,Y).
il_prossimo(X,Y) :- oggetto(X), oggetto(Y),
                    not non_il_prossimo(Z,Y).
```

Esercizio 12.4. Nel programma precedente si assume che l'ordine descritto dal predicato `minore` sia lineare ed in particolare totale. Si modifichi il programma in modo che operi correttamente anche nel caso in cui l'ordine è parziale.

4. ASP-solver

In questa sezione illustreremo principalmente un ASP-solver, ovvero un sistema in grado di calcolare gli answer set di un programma ASP. L'ASP-solver che tratteremo è Smodels, ma i concetti trattati in questa sezione si applicano anche ad altri ASP-solver, quali ad esempio Cmodels [**LM04**].[4] Forniremo anche alcune informazioni basilari relative all'ASP-solver DLV, per lo più mirate a cogliere le differenze tra DLV e Smodels. Ulteriori informazioni su alcuni altri ASP-solver, resi disponibili da vari istituti di ricerca, si trovano nella Sezione 2 dell'Appendice B.

4.1. Smodels. Smodels è un tool in grado di calcolare gli answer set di programmi ASP. È stato sviluppato presso la Helsinki University of Technology ed è possibile, al sito web `http://www.tcs.hut.fi/Software/smodels`, sia ottenere il software (sotto GNU Public Licence) sia accedere alla documentazione relativa. Il sistema Smodels è in realtà composto da due strumenti: lparse e smodels. Il primo funge da front-end ed accetta in input un programma ASP P (in cui occorrono eventualmente delle variabili). Successivamente ad una fase di analisi sintattica del programma, lparse effettua il cosiddetto *grounding*. Questa trasformazione ha lo scopo di produrre una forma ottimizzata, e secondo una rappresentazione interna, del programma $ground(P)$. L'output di lparse viene poi processato da smodels che calcola effettivamente gli answer set del programma grounded (e quindi di P).

4.2. Il grounding. Come accennato, il grounding è il processo attraverso il quale dato un programma P contenente variabili, si ottiene il programma $ground(P)$. Allo scopo di rendere finito il processo di grounding e di poter generare un programma $ground(P)$ composto da un numero finito di regole, il programma P deve rispettare alcuni vincoli. Innanzitutto, è consentito solamente un uso limitato dei simboli di funzione (si veda pag. 179); inoltre il

[4]Dal punto di vista del programmatore non ci sono sostanziali differenze tra Smodels e Cmodels. Entrambi utilizzano la stessa sintassi esterna e lo stesso front-end (lparse). Differiscono nel modo in cui è implementato il solver.

programma deve essere *strongly range restricted*. Intuitivamente, l'idea base è che il programma P deve essere strutturato in modo che sia possibile, per ogni variabile di una regola, stabilire l'insieme di valori che essa può assumere; tale insieme deve essere inoltre finito. Al fine di definire il concetto di programma strongly range restricted, dobbiamo introdurre la nozione preliminare di *dependency graph* di un programma ASP.

DEFINIZIONE 12.4. Sia P un programma. Il *dependency graph* D_P di P è un grafo etichettato così definito:

- i nodi di D_P corrispondono ai simboli di predicato presenti in P;
- vi è un arco diretto etichettato $\langle p_i, p_j, \ell \rangle$ tra i due nodi p_i e p_j se in P esiste una regola in cui p_i occorre nella testa e p_j occorre nel corpo. L'etichetta ℓ può essere uno o entrambi i simboli $+$, $-$ a seconda che il simbolo p_j occorra in un letterale positivo o negativo nel corpo della regola. (Si noti che una etichetta può essere anche la coppia $+, -$.)

Un ciclo nel grafo D_P si dice *ciclo negativo* se almeno una delle sue etichette contiene $-$.

Un altro concetto utile:

DEFINIZIONE 12.5. Sia P un programma e sia D_P il suo dependency graph. Un predicato p che occorre in P si dice *predicato di dominio* se e solo se in D_P ogni cammino che parte dal nodo p non contiene cicli negativi.

DEFINIZIONE 12.6. Una regola ρ si dice *strongly range restricted* se ogni variabile che occorre in ρ occorre anche negli argomenti di un predicato di dominio nel corpo di ρ.

Un programma P è *strongly range restricted* se tutte le sue regole sono strongly range restricted.

Si può catturare intuitivamente l'idea soggiacente a questi concetti pensando ai predicati di dominio come a dei predicati per i quali tutto è noto, ovvero predicati veri in un numero finito di istanze ground tutte in qualche modo esplicite nel programma (solitamente, ma non necessariamente, sono predicati descritti in P in modo estensionale). Conseguentemente, se una regola ρ è strongly range restricted, allora i predicati di dominio determinano un numero finito di possibili valori per tutte le variabili di ρ. Quindi, esisteranno solo un numero finito di istanze ground di ρ.

ESEMPIO 12.8. Si consideri il programma P presentato nella Sezione 3.4 e qui di seguito riportato:

```
differente_da_scelto(X) :- scelto(Y), X!=Y.
scelto(X) :- possibile(X), not differente_da_scelto(X).
possibile(a).
possibile(b).
possibile(c).
```

Il dependency graph D_P di P è illustrato in Figura 12.1. Analizzando D_P si vede chiaramente che l'unico predicato di dominio è `possibile`. Dato che nella prima regola del programma la variabile `X` non occorre come argomento di alcun predicato di dominio, ne consegue che P non è strongly range restricted e per questo non è processabile da lparse. È sufficiente, comunque, modificare marginalmente P affinché lparse possa effettuarne il grounding. Consideriamo

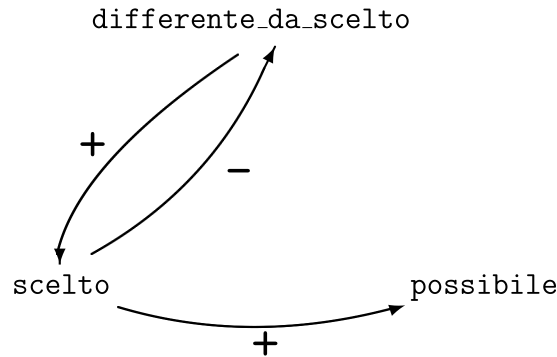


FIGURA 12.1. Il dependency graph D_P relativo al programma P dell'Esempio 12.8

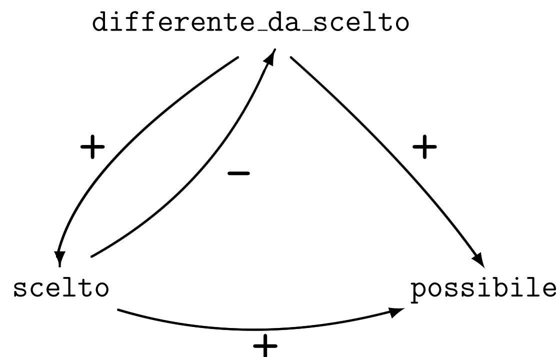


FIGURA 12.2. Il dependency graph $D_{P'}$ relativo al programma P' dell'Esempio 12.8

infatti il programma P' ottenuto da P aggiungendo dei predicati di dominio nella prima regola:

```
differente_da_scelto(X) :- possibile(X), possibile(Y), scelto(Y), X!=Y.
scelto(X) :- possibile(X), not differente_da_scelto(X).
possibile(a).
possibile(b).
possibile(c).
```

Il dependency graph $D_{P'}$ di P' è illustrato in Figura 12.2. P' è strongly range restricted.

4.3. Uso di lparse e smodels. Usualmente un programma ASP per Smodels viene scritto in un file di testo, diciamo `asp.lp`. Tale file viene processato dalla coppia lparse/smodels

invocando il comando

lparse asp.lp | smodels n

Il parametro opzionale n è un numero intero che indica quanti answer set (al massimo) smodels debba produrre. In assenza di questo parametro verrà prodotto (se esiste) un solo answer set. Se $n = 0$ allora verranno prodotti tutti gli answer set del programma. È possibile scrivere un programma ASP spezzandolo in più file. Questa possibilità può risultare utile, ad esempio, quando un file (o più file) contiene il vero e proprio programma in grado di risolvere un problema, mentre un altro file (o più file) contiene la descrizione di una specifica istanza del problema. In questo caso il comando da utilizzare sarà:

lparse asp1.lp asp2.lp asp3.lp ... | smodels n

Esistono diverse opzioni che permettono di controllare sia il comportamento di lparse che di smodels. Per una loro descrizione esaustiva si rimanda a [**Syr01**].

4.4. Estensioni del linguaggio offerte da Lparse. In questa sezione elenchiamo le principali estensioni dell'answer set programming offerte dal front-end lparse. Illustreremo prima la parte relativa a programmi ground. Successivamente affronteremo i programmi con variabili (per una descrizione esaustiva si rimanda a [**Bar04**] o a [**Syr01**]).

4.4.1. *Ground cardinality constraint.* È un constraint C della forma

$$n\{\mathtt{L}_1,\ldots,\mathtt{L}_h,\ \mathtt{not}\ \mathtt{H}_1,\ldots,\ \mathtt{not}\ \mathtt{H}_k\}m$$

dove $\mathtt{L}_1,\ldots,\mathtt{L}_h,\mathtt{H}_1,\ldots,\mathtt{H}_k$ sono atomi e n e m sono numeri interi (uno o entrambi possono essere assenti). Definiamo, relativamente ad un insieme di atomi S e ad un cardinality constraint C il valore $val(C,S)$ come

$$val(C,S) = \mid S \cap \{\mathtt{L}_1,\ldots,\mathtt{L}_h\} \mid + (k - \mid S \cap \{\mathtt{H}_1,\ldots,\mathtt{H}_k\} \mid).$$

Diremo che C è vero in S se $n \leqslant val(C,S) \leqslant m$. Si confronti questo tipo di regola con il programma di Sezione 3.4.

4.4.2. *Ground weight constraint.* È un constraint C della forma

$$n[\mathtt{L}_1 = w_{\mathtt{L}_1},\ldots,\mathtt{L}_h = w_{\mathtt{L}_h},\ \mathtt{not}\ \mathtt{H}_1 = w_{\mathtt{H}_1},\ldots,\ \mathtt{not}\ \mathtt{H}_k = w_{\mathtt{H}_k}]m$$

dove $\mathtt{L}_1,\ldots,\mathtt{L}_h,\mathtt{H}_1,\ldots,\mathtt{H}_k$ sono atomi e n, m e gli w_x sono numeri interi. I valori $w_{\mathtt{L}_i}$ sono pesi assegnati ad ogni atomo vero, mentre i valori $w_{\mathtt{H}_i}$ sono pesi assegnati agli atomi falsi. I pesi unitari possono essere omessi. Definiamo, relativamente ad un insieme di atomi S e ad un weight constraint C il valore $val'(C,S)$ come

$$val'(C,S) = \sum_{\mathtt{L}_i \in S, 1 \leqslant i \leqslant h} w_{\mathtt{L}_i} + \sum_{\mathtt{H}_j \notin S, 1 \leqslant j \leqslant k} w_{\mathtt{H}_j}$$

Diremo che C è vero in S se $n \leqslant val'(C,S) \leqslant m$.

4.4.3. *Le regole ground di Lparse.* Una regola ground ammessa da Lparse è una regola della forma:

$$\mathtt{L}_0\ \texttt{:-}\ \mathtt{L}_1,\ldots,\mathtt{L}_k$$

dove $\mathtt{L}_0$ è un atomo ground, oppure un ground cardinality constraint, oppure un ground weight constraint. Ogni $\mathtt{L}_i$, per $1 \leqslant i \leqslant k$, è invece un letterale ground, oppure un ground cardinality constraint, oppure un ground weight constraint.

Esempio 12.9. Consideriamo il programma P

```
1{a,b,c}2 :- p.
p.
```

P ha come answer set tutti gli insiemi di atomi che contengono p e che contengono uno o due tra gli atomi a, b, e c. Ad esempio $\{a, p\}$ e $\{b, c, p\}$ sono due answer set. Al contrario $\{a, b, c, p\}$ non è un answer set di P.

Si noti come l'impiego del cardinality constraint nel precedente programma si possa simulare tramite le seguenti regole:

```
p.
a :- p, not na.
na :- not a.
b :- p, not nb.
nb :- not b.
c :- p, not nc.
nc :- not c.
:- not na, not nb, not nc.
:- not a, not b, not c.
```

Infatti, dato che le prime sette regole impongono una scelta (Sezione 3.2) tra a e na, tra b e nb, e tra c e nc, tramite il primo dei due vincoli si impone indirettamente che uno tra $\{a, b, c\}$ debba essere vero. Il secondo vincolo impone invece che uno tra $\{a, b, c\}$ debba essere falso.

Osserviamo quindi che è sempre possibile effettuare una sorta di compilazione delle estensioni del linguaggio offerte da Lparse+Smodels in programmi ASP costituiti da regole del tipo introdotto nella Sezione 1.

Tuttavia, nelle implementazioni reali, queste estensioni vengono tradotte nelle seguenti regole ground primitive, che vengono poi trattate con la semantica opportuna dai solver quali Smodels:

choice rule:

$$\{h_1, \dots, h_k\} \leftarrow a_1, \dots, a_m, \text{ not } b_1, \dots, \text{ not } b_n.$$

In questo caso in ogni answer set vengono scelti zero o più (eventualmente tutti) gli atomi h_i. Ciò corrisponde ad utilizzare il cardinality constraint $0\{h_1, \dots, h_k\}k$.

primitive choice rule:

$$h \leftarrow \alpha\{a_1, \dots, a_m, \text{ not } b_1, \dots, \text{ not } b_n\}.$$

primitive weight rule:

$$h \leftarrow \{a_1 = W_1, \dots, a_m = W_m, \text{ not } b_1 = W'_1, \dots, \text{ not } b_n = W'_n\} \geq \alpha.$$

dove $h, h_1, \dots, h_k, a_1, \dots, a_m, b_1, \dots, b_n$ sono atomi ground, $\alpha, W_1, \dots, W_m, W'_1, \dots, W'_n$ sono numeri naturali.

Come per i programmi ASP descritti nella Sezione 1 anche per queste estensioni offerte da Smodels è possibile definire le nozioni di programma ridotto e conseguentemente conferire a programmi Smodels una semantica in modo del tutto analogo a quanto visto nelle sezioni precedenti. Per un trattamento formale e dettagliato si rimanda ad esempio a [**Bar04**].

4.4.4. *L'istruzioni* `compute` *e* `optimize` *(caso ground).* L'istruzione di lparse `compute` ha la seguente sintassi:

$$\texttt{compute } n\{\texttt{L}_1\texttt{,L}_2\texttt{,...,L}_h\texttt{, not H}_1\texttt{, not H}_2\texttt{,..., not H}_k\}.$$

Una istruzione `compute` rappresenta una sorta di filtro per answer set: tutti gli answer set che non contengono tutti gli $\mathtt{L}_i$ o che contengono almeno uno degli $\mathtt{H}_j$ vengono scartati. Il parametro opzionale n indica quanti (al massimo) answer set produrre (ovviamente tra quelli che soddisfano il filtro). In assenza di tale parametro verrà prodotto un solo answer set. Se $n = 0$ verranno prodotti tutti gli answer set. In un programma ASP è possibile non inserire alcuna istruzione `compute`, ciò corrisponde a non imporre alcun filtro: tutti gli answer set saranno prodotti.

L'istruzione di lparse `optimize` ha invece quattro forme alternative:

- $\texttt{maximize } \{\texttt{L}_1\texttt{,...,L}_h\texttt{, not H}_1\texttt{,..., not H}_k\}.$
- $\texttt{minimize } \{\texttt{L}_1\texttt{,...,L}_h\texttt{, not H}_1\texttt{,..., not H}_k\}.$
- $\texttt{maximize [L}_1 = w_{\mathtt{L}_1}\texttt{,...,L}_h = w_{\mathtt{L}_h}\texttt{, not H}_1 = w_{\mathtt{H}_1}\texttt{,..., not H}_k = w_{\mathtt{H}_k}\texttt{].}$
- $\texttt{minimize [L}_1 = w_{\mathtt{L}_1}\texttt{,...,L}_h = w_{\mathtt{L}_h}\texttt{, not H}_1 = w_{\mathtt{H}_1}\texttt{,..., not H}_k = w_{\mathtt{H}_k}\texttt{].}$

Qualora un programma contenga una sola istruzione `maximize` (o `minimize`) viene prodotto solamente l'answer set che realizza il valore ottimo (massimo o minimo, a seconda della istruzione). Più precisamente il comportamento di smodels è il seguente: viene inizialmente prodotto un primo answer set; dopo di ciò la elaborazione continua ricercando ulteriori answer set, ma solo quelli che realizzano un valore migliore verranno prodotti.

Se il programma invece contiene molteplici istruzioni `optimize` allora queste vengono considerate in ordine inverso a quello in cui compaiono nel programma (ovvero dall'ultima alla prima): un answer set è dichiarato migliore di un altro se realizza un valore migliore nell'ultimo `optimize` del programma. In tal caso non vengono valutati gli altri `optimize`. Se invece due answer set realizzano lo stesso valore relativamente all'ultimo `optimize` del programma, si passa a considerare il penultimo `optimize`, e così via.

ESEMPIO 12.10. Si consideri il programma costituito dalla sola regola

```
1{a,b,c,d}4.
```

Chiaramente smodels fornirà 15 diversi answer set. Se al programma aggiungiamo una regola

```
1{a,b,c,d}4.
minimize {a,b,c,d}.
```

allora verranno prodotti solamente answer set di cardinalità unitaria. Se inoltre aggiungessimo anche una ulteriore regola ottenendo il programma:

```
1{a,b,c,d}4.
maximize [a=2,b=1,c=2,d=1].
minimize {a,b,c,d}.
```

verrà prodotto solamente un answer set tra $\{c\}$ e $\{a\}$ (il primo dei due ad essere trovato).

4.4.5. *Letterali condizionali e uso delle variabili.* Al fine di rendere più compatti i programmi, lparse ammette l'impiego di variabili e di *conditional literal* nella forma:

$$\texttt{p(X}_1\texttt{,...,X}_n\texttt{) : p}_1\texttt{(X}_{i_1}\texttt{) : ... : p}_m\texttt{(X}_{i_m}\texttt{)}$$

con $\{i_1, \ldots, i_m\} \subseteq \{1, \ldots, n\}$. Il predicato `p` è detto *enumerated predicate* mentre ogni $\mathtt{p}_i$ è detto *condition on* `p`.

Utilizzando i letterali condizionali è possibile generalizzare la sintassi dei cardinality e weight constraint ammettendo variabili. In generale quindi un cardinality constraint può avere la forma di un ground cardinality constraint oppure un la forma seguente:

$$k\{Cond_Lit\}h$$

dove *Cond_Lit* è un letterale condizionale. Un weight constraint invece può avere la forma di un ground weight constraint oppure un la forma seguente:

$$k\,[Cond_Lit]\,h$$

Lparse impone una restrizione sull'impiego di questo tipo di constraint, al fine di permettere il grounding. In pratica tutte le regole che contengono letterali condizionali devono essere *domain restricted*. Ciò significa che ogni variabile in queste regole deve occorrere come argomento o di un predicato di dominio, o di una delle condition della regola.

Il prossimo esempio illustra un semplice impiego di questi constraint.

Esempio 12.11. Supponiamo sia dato un grafo con n nodi e si voglia assegnare un unico colore ad ogni nodo. Il seguente programma risolve questo problema: ogni answer set è un diverso assegnamento di colori.

```
nodo(a1).
nodo(a2).
⋮
nodo(an).
colore(c1).
colore(c2).
⋮
colore(ck).
1{assegna(X,C) : colore(C)}1 :- nodo(X).
```

L'ultima clausola utilizza un insieme definito intensionalmente. E' come se fosse scritto:

```
1{assegna(X,c1), assegna(X,c2),..., assegna(X,ck) }1 :- nodo(X).
```

che dopo il grounding diventa:

```
1{assegna(a1,c1), assegna(a1,c2),..., assegna(a1,ck) }1.
1{assegna(a2,c1), assegna(a2,c2),..., assegna(a2,ck) }1.
...
1{assegna(an,c1), assegna(an,c2),..., assegna(an,ck) }1.
```

4.4.6. *Altri costrutti utili.* Vengono di seguito descritti brevemente diversi costrutti e funzionalià offerti da lparse. Per un trattamento esaustivo di questi ed altre caratteristiche di Smodels si rimanda a [**Syr01**].

Intervalli: lparse permette di scrivere in modo compatto una lista di atomi. Ad esempio, in luogo delle regole

```
val(4).
val(5).
val(6).
val(7).
```

è possibile scrivere la singola regola `val(4..7).` Similmente invece di scrivere la regola

```
p:- q(6), q(7), q(8).
```

è possibile scrivere in modo compatto la regola

```
p:- q(6..8).
```

Argomenti multipli: similmente agli intervalli appena descritti, lparse permette un altro genere di scrittura compatta delle regole. Ad esempio invece di scrivere

```
val(c).
val(f).
val(a).
```

possiamo scrivere

```
val(c;f;a).
```

Analogamente, la regola

```
p:- q(6;a;2;8).
```

è equivalente a

```
p:- q(4), q(a), q(2), q(8).
```

Hide e show: la dichiarazione `#hide p(X,Y).` indica a smodels di non includere gli atomi della forma `p(X,Y)` nella presentazione degli answer set (ovvero, nell'output testuale prodotto). La dichiarazione `#hide.` si applica a tutti i predicati (quindi in questo caso verrà fornita solo una risposta positiva o negativa a seconda che esista un answer set o meno, ma questo non verrà stampato. Dato che la dichiarazione `#hide.` inibisce la stampa di tutti gli atomi, in presenza di tale dichiarazione è possibile indicare esplicitamente quali atomi stampare. Ciò tramite dichiarazioni `#show p(X,Y).`. La dichiarazione `#hide.` ha l'ovvio significato atteso.

Costanti: è possibile, tramite una dichiarazione del tipo

`#const` *identificatore* `=` *expr*.

dichiarare una costante il cui valore sarà il valore della espressione numerica *expr*. Qualora si ometta *expr* il valore deve essere fornito sulla linea di comando, come segue:

`lparse -c` *identificatore*`=`*valore* `asp.lp | smodels` n

Esempio 12.12. Il seguente programma Smodels corrisponde al programma ASP descritto nell'Esercizio 12.1. Si noti come l'uso dell'intervallo e delle variabili consenta di scrivere una sola regola in luogo di n.

```
indice(1..n).
p(X) :- not p(X+1), indice(X).
```

Si ricordi che il valore di `n` deve essere fornito sulla linea di comando, ad esempio:

```
lparse -c n=10 file.lp | smodels
```

Dichiarazioni di weight: abbiamo precedentemente descritto le istruzioni `compute` e `optimize` ed abbiamo visto come in tali istruzioni sia possibile assegnare dei pesi (weight) agli atomi. Similmente alle dichiarazioni di costanti, è possibile dichiarare il peso di un atomo una volta per tutte. La dichiarazione in questione ha due formati possibili:

```
#weight atomo = espressione.
#weight atomo1 = atomo2.
```

dove l'espressione può anche coinvolgere variabili che occorrono nell'atomo, come ad esempio in:

```
#weight p(X) = X + 7.
```

Il secondo formato della dichiarazioni di weight è utilizzabile per rendere uguali i pesi di due atomi. Dovrà quindi essere presente nel programma una altra dichiarazione relativa al peso di `atomo2`.

FUNZIONI BUILT-IN: lparse mette a disposizione del programmatore un minimo insieme di funzioni built-in. Tra esse ricordiamo `plus`, `minus`, `times`, `div`, `mod`, `lt`, `gt`, `le`, `ge`, `neq`, `abs`, `and`, `or`, ecc. con gli intuibili significati. È da tenere presente il fatto che tali funzioni vengono valutate durante il processo di grounding. Quindi in tale momento gli argomenti delle funzioni devono essere disponibili. Una distinzione va fatta tra l'operatore di confronto "`==`" e quello di assegnamento "`=`". Un atomo della forma `X=`***`espressione`*** assegna alla variabile `X` il valore di ***`espressione`***.

È anche possibile per il programmatore definire proprie funzioni tramite dei programmi C o C++. Lparse mette a disposizione allo scopo un limitato ma utile meccanismo di interfacciamento con programmi C o C++ (si veda [**Syr01**] per i dettagli).

COMMENTI: analogamente al linguaggio Prolog le linee di commento nei file di input per lparse iniziano con il carattere %.

4.5. Un semplice algoritmo per calcolare un answer set. Descriviamo in questa sezione un semplice algoritmo (ed i risultati teorici su cui esso si basa) che può essere implementato per calcolare l'answer set di un programma ASP, nel caso esso sia unico. Trattiamo quindi solamente il caso semplificato di programmi, senza negazione esplicita (si veda in merito la Sezione 6) e senza disgiunzione, per i quali sia noto che esiste un unico answer set. (Se questa assunzione cade, allora l'algoritmo che vedremo non può esser impiegato. Esistono ovviamente algoritmi applicabili al caso generale, si veda [**Sim00, SNS02, Bar04**].)

Il seguente risultato giustifica l'algoritmo che presenteremo: Dato un programma Q denotiamo con $as(Q)$ il suo answer set.

TEOREMA 12.3. *Sia S l'answer set del programma P. Siano L e U due insiemi di atomi. Se $L \subseteq S \subseteq U$ allora $L \cup as(P^U) \subseteq S \subseteq U \cap as(P^L)$.*

Il teorema fornisce un metodo iterativo per cercare l'answer set S (nel caso ve ne sia al più uno). Poichè ovviamente

$$\emptyset \subseteq S \subseteq \mathcal{B}_P$$

ponendo $L_0 = \emptyset$ e $U_0 = \mathcal{B}_P$ il teorema ci dice che

$$\emptyset \cup as(P^{\mathcal{B}_P}) = as(P^{\mathcal{B}_P}) \subseteq S \subseteq \mathcal{B}_P \cap as(P^{\emptyset}) = as(P^{\emptyset})$$

Se questi due insiemi sono uguali, abbiamo determinato S, altrimenti continuo assegnando $L_1 = as(P^{\mathcal{B}_P})$ e $U_1 = as(P^{\emptyset})$.

Se ad un certo passo scopro che $L_i = U_i$, allora ho trovato l'answer set S. Se invece si verifica che $L_i \not\subseteq U_i$, allora non esiste alcun answer set.

5. Cenni al solver Cmodels

La tecnica su cui si basa Cmodels per calcolare gli answer sets di un programma è radicalmente diversa da quella appena vista per il solver Smodels. L'idea di base è quella di trasformare il programma P in una formula logica proposizionale e valutarne poi la soddisfacibilità. Tale formula viene costruita in modo che i suoi modelli siano tutti e soli gli answer sets del programma di partenza. La ricerca delle soluzioni viene dunque delegata ad un SAT-solver. Per una particolare classe di programmi, detti *tight*, la traduzione è molto semplice e ottenuta direttamente dal completamento del programma (si veda il capitolo 8). Per i programmi *non tight* è necessario considerare, unitamente al completamento del programma, anche un insieme (potenzialmente molto grande) di particolari formule, dette *loop-formule*.

Forniremo ora qualche dettaglio sulla traduzione menzionata. Per semplicità, considereremo solo programmi P ground (il caso di programmi non ground viene gestito analogamente, premettendo una fase di grounding). Paggiungeertanto tutti i letterali presenti sono di fatto letterali proposizionali. Dato un codice ASP P (sono ammessi quindi dei constraint), il programma $Comp(P)$ è definito nel modo seguente:

(1) Sia p definito dalle regole $p \leftarrow G_1, \ldots, p \leftarrow G_n$, allora $Comp(P)$ comprenderà la formula $p \leftrightarrow G_1 \vee \cdots \vee G_n$.
(2) Se invece l'atomo q non compare come testa di alcuna regola, $Comp(P)$ comprenderà la formula: $\neg q$.
(3) Se in P è presente il vincolo $\leftarrow B_1, \ldots, B_n$ allora viene aggiunta a $Comp(P)$ la formula $\neg(B_1 \wedge \cdots \wedge B_n)$.

Si ricorda che il vincolo $\leftarrow B_1, \ldots, B_n$ è equivalente, dal punto di vista dei modelli stabili, alla regola $r \leftarrow not\, r, B_1, \ldots, B_n$ con r simbolo predicativo mai usato altrove. Si osservi pertanto che la formula ottenuta con la regola (3) e la formula che si otterrebbe applicando la regola (1) a $r \leftarrow not\, r, B_1, \ldots, B_n$ sono equivalenti.

Vale il seguente Teorema:

Teorema 12.4. *Ogni modello stabile di P è modello di $Comp(P)$.*

Il viceversa non è sempre vero. Si pensi ad esempio al programma $P = \{p \leftarrow q. \quad q \leftarrow p.\}$ che ha l'unico modello stabile $\emptyset$. Tuttavia, $Comp(P) = p \leftrightarrow q$ ammette sia il modello vuoto che il modello $\{p, q\}$.

Vediamo come si possa ovviare a questo problema e trovare una formula i cui modelli coincidano con gli answer set del programma anche nel caso quest'ultimo non sia tight. Si costruisca un grafo con un nodo associato a ogni simbolo di predicato presente nel programma. Per ogni regola

$$p \leftarrow q_1, \ldots, q_m, not\, r_1, \ldots, not\, r_n$$

per ogni q_i, si aggiunga un arco uscente dal nodo q_i ed entrante nel nodo p (si noti che, non si considerano i naf-literals della regola). Se il grafo ottenuto è aciclico, allora il programma è assolutamente tight (per brevità diremo *tight*). Vale in seguente risultato.

Teorema 12.5. *Se P è programma tight, allora modelli stabili di P e modelli di $Comp(P)$ coincidono.*

Pertanto i modelli stabili per programmi tight si possono ottenere sfruttando un SAT solver operante sul completamento del programma.

Se invece il programma non è tight (come ad esempio $\{p \leftarrow q. \quad q \leftarrow p.\}$) si consideri uno alla volta tutti i cicli nel grafo. Sia L l'insieme dei letterali presenti in un ciclo. Definiamo

$$R(L) = \{p \leftarrow G \in P \,:\, p \in L, \text{per ogni atomo non negato } q \text{ in } G,\, q \notin L\}$$

Sia $L = \{p_1, \dots, p_n\}$ e siano

$$\begin{array}{ccc} p_1 \leftarrow G_1^1 & \cdots & p_1 \leftarrow G_{s_1}^1 \\ & \vdots & \\ p_n \leftarrow G_1^n & \cdots & p_n \leftarrow G_{s_n}^n \end{array}$$

le clausole definenti gli atomi di L in $R(L)$. Si aggiunga al completamento di P la formula (detta loop formula):

$$\neg(G_1^1 \vee \cdots \vee G_{s_1}^1 \vee G_1^n \vee \cdots \vee G_{s_n}^n) \rightarrow (\neg p_1 \wedge \cdots \wedge \neg p_n)$$

Nel caso del precedente esempio $\{p \leftarrow q. \quad q \leftarrow p.\}$, l'unico loop è $L = \{p, q\}$. Pertanto $R(L) = \emptyset$. La formula è:

$$\neg\,\mathsf{false} \rightarrow \neg p \wedge \neg q$$

equivalente a $\neg p \wedge \neg q$ che non è modellata dall'interpretazione $\{p, q\}$. L'unico modello della formula è quello vuoto, che è anche l'answer set del programma di partenza.

6. La negazione esplicita in ASP

In questa sezione affronteremo brevemente il problema di come sia possibile utilizzare e inferire informazione negativa in programmi ASP. Chiariamo subito considerando l'Esempio 12.6, che tipo di informazione negativa ci interessa trattare. Anticipiamo subito che non si tratta della informazione catturata dai naf-literal.

Supponiamo che `tweety` sia ferito e non possa volare. Ci chiediamo: *"Come aggiungiamo l'informazione che* `tweety` *non può volare nel programma dell'Esempio 12.6?"* Non possiamo utilizzare dei naf-literal perchè l'operatore *not* significa "falso perchè non dimostrabile". Ciò non è proprio il concetto di negazione che ci interessa. Ci interessa una negazione, detta usualmente *negazione esplicita* che invece è molto simile alla negazione della logica classica. Il successivo esempio illustra meglio la differenza tra negazione per fallimento e negazione esplicita.

Esempio 12.13. Supponiamo di voler rappresentare, con delle regole ASP, il fatto che è sicuro attraversare a nuoto un fiume se non ci sono piraña che vi nuotano. Se disponiamo solo della negazione per fallimento, il meglio che possiamo fare è scrivere la regola

```
attraversa :- not ci_sono_piraña.
```

Vi fidereste di questo programma ASP? Per rispondere riflettiamo su quale sia il significato di soddisfare il naf-literal `not ci_sono_piraña`. Ciò significa che, in base a quanto conosciamo, non siamo in grado di dimostrare che l'atomo `ci_sono_piraña` sia vero. Per fare un parallelo con la realtà: pensate di essere in prossimità della riva di un fangoso fiume dell'Amazzonia. Non siete quindi in grado di intravedere attraverso l'acqua se dei piraña nuotino o meno nel fiume (anche se l'acqua fosse limpida non potreste che osservare una piccola porzione del fiume e non potreste essere certi che vi non siano piraña in nessun tratto del fiume). Attraversereste a nuoto il fiume?

La regola precedente, dato che usa proprio la negazione per fallimento, vi direbbe che è sicuro attraversare se non riuscite a sapere che ci sono piraña. Evidentemente questa regola non è ciò che desideriamo. Vorremmo invece una regola come la seguente:

```
attraversa :- -ci_sono_piraña.
```

Essa asserisce "Attraversa se non ci sono piraña" e non pericolosamente "Attraversa se non riesci a dimostrare che ci sono piraña".

Per superare quindi questi limiti si estende la classe dei programmi ASP di interesse. Le regole di questa nuova classe di programmi hanno la stessa forma delle regole introdotte nella Definizione 12.1, con l'unica differenza che ogni L_i non è vincolato ad essere una semplice formula atomica, ma può essere un letterale (positivo o negativo). Tali programmi sono solitamente detti *programmi logici estesi (extended logic programs).*

Le regole di un programma esteso contengono quindi due tipi di negazioni, la negazione esplicita e la negazione per fallimento finito. Tutti i letterali non naf-literal (positivi o negati dalla negazione esplicita) vengono considerati alla stessa stregua. L'unico vincolo che si impone è che un answer set di un programma sia sempre consistente, ovvero non contenga sia `p` che `-p`.[5]

ESEMPIO 12.14. Ecco alcuni esempi di programma estesi e dei rispettivi answer set:

P_1: `{p :- q., -p :- r., q.}`	S_1: `{q, p}`
P_2: `{p :- q., -p :- r., r.}`	S_2: `{r, -p}`
P_3: `{p :- q., -p :- r.}`	S_3: `{}`
P_4: `{p :- q., -p :- r., q., r.}`	`---`

Il prossimo esempio illustra il fatto che la negazione estesa non ha le stesse proprietà della negazione della logica classica.

ESEMPIO 12.15. Consideriamo i due programmi:

P_1: `{-p.,    p :- -q.}`
P_2: `{-p.,    q :- -p.}`

Se la negazione fosse quella classica P_1 e P_2 sarebbero costituiti dalle stesse clausole, e quindi sarebbero equivalenti. Considerando invece la negazione esplicita, scopriamo che i due programmi hanno entrambi un answer set, ma i due answer set sono diversi. L'answer set di P_1 è `{-p}`, mentre quello di P_2 è `{-p, q}`. Quindi per ASP P_1 e P_2 hanno diversa semantica.

Un altro esempio:

ESEMPIO 12.16. Consideriamo il programma di una sola regola:

```
-q :- not p.
```

Esso asserisce "`q` è falso se non c'è evidenza del fatto che `p` sia vero" (cioè se "non si riesce a dimostrare che `p` sia vero"). Esiste un unico answer set: `{-q}`.

Riprendiamo il problema menzionato all'inizio della sezione: `tweety` è ferito e non vola:

[5]Alcuni testi e alcuni ASP-solver adottano la convenzione che un answer set è ammesso come risposta se è consistente oppure se è l'insieme $\{p \mid p$ è atomo del programma$\} \cup \{-p \mid p$ è atomo del programma$\}$ (cioè l'insieme di letterali contraddittorio più grande).

Esempio 12.17. Il seguente programma ground è una alternativa al programma dell'Esempio 12.6 che utilizza i due tipi di negazione:

```
vola(tweety) :- uccello(tweety), not -vola(tweety).
vola(skippy) :- uccello(skippy), not -vola(skippy).
-vola(tweety) :- pinguino(tweety).
-vola(skippy) :- pinguino(skippy).
uccello(tweety).
uccello(skippy).
pinguino(skippy).
```

Il suo answer set è:

{ `uccello(tweety), uccello(skippy), pinguino(skippy),`
`vola(tweety), -vola(skippy)` }

Importante caratteristica di questo programma è che se ad un certo punto scopriamo che `tweety` è ferito e non vola, è sufficiente aggiungere il fatto `-vola(tweety)` al programma. Il nuovo programma avrà una diversa semantica che rifletterà il fatto che `tweety` non vola. Il suo answer set sarà infatti:

{ `uccello(tweety), uccello(skippy), pinguino(skippy),`
`-vola(tweety), -vola(skippy)` }

Anche gli answer set dei programmi estesi godono di proprietà di minimalità:

Teorema 12.6. *Sia dato un programma esteso P. Non possono esistere due answer set di P tali che uno sia strettamente incluso nell'altro.*

6.1. Negazione esplicita in Smodels. Lparse è in grado di processare programmi ASP in cui la negazione esplicita (indicata con il simbolo '`-`') occorre nel corpo delle regole. Il modo in cui un letterale `-p` viene trattato è aggiungendo al programma una nuova regola, più precisamente il constraint:

```
:- p,  -p.
```

e trattando il letterale `-p` alla stessa stregua di un letterale positivo. La nuova regola assicura che in nessun answer set del programma potranno essere soddisfatti sia `p` che `-p`.

Per utilizzare la negazione esplicita in un programma è necessario comunicare questa intenzione a lparse tramite la opzione `--true-negation`:

```
lparse --true-negation file.lp | smodels
```

Se ad esempio il file `file.lp` contenesse le regole

```
a :- not -b.
-b :- not a.
q :- a.
-q :- a.
```

otterremmo un'unica soluzione: {`-b`}.

7. La disgiunzione in ASP

In questa sezione accenneremo brevemente ad una altra estensione dell'ASP proposta in letteratura. Introdurremo un costrutto `or` per rappresentare informazione disgiuntiva nella testa delle regole. Menzioniamo subito che la semantica di `or` non coincide con la semantica che usualmente si assegna alla disgiunzione $\vee$ nella logica classica. La semantica di `or` è definita in termini di answer set. Per fissare le idee potremmo affermare che la clausola $A \vee B$ significa "A è vero oppure B è vero", mentre è più corretto assegnare alla regola `A or B.` il significato "A è creduto vero oppure B è creduto vero". Conseguentemente, mentre la clausola $A \vee \neg A$ è sempre vera, si ha che `A or -A.` potrebbe non essere vera.

Non presenteremo una trattazione formale di programmi con teste disgiuntive. Illustriamo solo alcuni esempi, menzionando il fatto che (al momento in cui queste righe vengono scritte) solo alcuni ASP-solver, tra cui DLV (si veda la Sezione 7.1), sono in grado di trattare questi programmi. Recentemente è stata anche proposta una estensione di Cmodels in grado di gestire programmi disgiuntivi (si veda in merito [**Lie05**]).

Esempio 12.18. Il seguente programma

```
a or b.
a :- b.
b :- a.
```

ha un unico answer set: {`a, b`}. Lo si confronti con

```
a :- not b.
b :- not a.
a :- b.
b :- a.
```

che non ha alcun answer set.

Esempio 12.19. Il seguente programma

```
a or b.
a or c.
:- a, not b, not c.
:- not a, b, c.
```

non ha alcun answer set, in quanto gli answer set del sotto programma fatto dalle prime due regole sono {`a`} e {`b, c`}, ma ognuno di essi contraddice una delle restanti regole. Siamo in presenza di un programma con cui l'intuizione può portarci facilmente alla soluzione errata. Un approccio sbagliato infatti potrebbe consistere nel calcolare i modelli delle prime due regole. Essi sono {`a`}, {`a, c`}, {`a, b`}, {`b, c`} e {`a, b, c`}. Poi potremmo scegliere quelli minimali tra coloro che non violano le ultime due regole. Otterremo così erroneamente {`a, c`} e {`a, b`}.

Per un trattamento formale dei programmi disgiuntivi, della loro semantica e degli algoritmi esistenti per calcolarne gli answer set, si rimanda a [**Bar04**], tra gli altri.

Ci limitiamo qui ad osservare che in alcuni casi un programma con teste disgiuntive può essere trasformato in un programma privo di disgiunzioni ma che possiede gli stessi answer set. Ecco un esempio di tali programmi:

Esempio 12.20. Il programma P costituito dalla sola regola (priva di corpo)

```
a or b or c.
```

ha tre answer set: {a}, {b}, e {c}. Possiamo sostituire l'unica regola di P con le tre seguenti:

```
a :- not b, not c.
b :- not a, not c.
c :- not a, not b.
```

Il programma così ottenuto ha evidentemente gli stessi answer set.

7.1. Cenni al sistema DLV. Il solver DLV è sviluppato presso la TU Wien ed è disponibile unitamente alla relativa documentazione, al sito

`www.dbai.tuwien.ac.at/proj/dlv`.

In questa sezione descriveremo solamente le principali caratteristiche che differenziano DLV da Smodels. Per maggiori dettagli sulle numerose funzionalità di DVL si rimanda alla documentazione disponibile al sito di DLV, anche in virtù del fatto che DLV è continuamente oggetto di migliorie ed estensioni.

Dal punto di vista del programmatore, una delle principali differenze tra DLV e Smodels consiste nel fatto che DLV è in grado di trattare programmi disgiuntivi. Contrariamente a ciò Smodels garantisce solamente una limitata forma di disgiunzione (si veda [**Syr01**]).

Ecco altre differenze interessanti:

- In luogo di weight constraint, cardinality constraint e conditional literal, DLV offre *weak constraint* e differenti front-end per specifici contesti applicativi quali planning, sistemi di diagnosi ed SQL3.
- Differente trattamento delle funzioni numeriche; DLV impone la definizione di una costante `#maxint` che indichi il massimo intero che possa essere impiegato nel programma. Questo è un approccio che evita l'impiego di predicati di dominio per restringere l'insieme dei valori ammissibili per argomenti di funzioni numeriche.
- DLV ammette variabili anonime, al contrario di Lparse.
- DLV, in virtù delle sue funzionalità orientate alla basi di dati relazionali, possiede interfacce sperimentali verso DBMS relazionali quali Oracle o Objectivity.
- DLV, al contrario di Lparse, non offre la possibilità di interfacciamento con funzioni C o C++.

8. Esercizi

Esercizio 12.5. Si scelga un numero naturale n. Scrivere un programma ASP che abbia almeno n modelli stabili diversi.

Esercizio 12.6. Si scrivano tutti gli answer set (se ne esistono) del programma:

```
c :- not c.
c :- not a.
a :- not b.
b :- not a.
```

Esercizio 12.7. Verificare se {p} sia o meno un answer set per il seguente programma ASP:

```
p :- not q.
q :- not r.
```

Esercizio 12.8. Si scrivano tutti i modelli stabili (se ne esistono) del programma:

```
c :- not c, not a.
a :- not b.
b :- not a.
d :- a.
d :- b.
```

Esercizio 12.9. Si scrivano tutti i modelli stabili (se ne esistono) del programma:

```
c :- not c, not -a.
-a :- not b.
b :- not -a.
```

Esercizio 12.10 (Knights and knaves). Su un'isola vi sono due tipi di abitanti. Gli onesti dicono sempre la verità mentre i bugiardi mentono sempre. Appena sbarcati sull'isola incontriamo due abitanti, che indichiamo con A e B. Appena A ci vede ci dice subito che sia lui che B sono bugiardi. Scrivere un programma ASP che determini chi è onesto e chi è bugiardo tra A e B.

Esercizio 12.11 (Ancora knights and knaves). Su un'isola vi sono due tipi di abitanti. Gli onesti dicono sempre la verità mentre i bugiardi mentono sempre. Appena sbarcati sull'isola incontriamo tre abitanti, che indichiamo con A, B, e C. Essi ci accolgono così: A dice che sia B che C sono onesti. B dice che A è bugiardo mentre C è onesto. Scrivere un programma ASP che determini chi è onesto e chi è bugiardo tra A, B, e C.

Esercizio 12.12 (Il club dei marziani e venusiani). Su Ganimede c'è un club noto come *"Il club dei marziani e venusiani"*, ciò perchè è prevalentemente frequentato da marziani e venusiani, anche se sono ammessi altri alieni. Un giorno un terrestre entra nel club e si accorge di non saper distinguere tra venusiani e marziani e neanche tra maschi e femmine, perchè le due razze sono molto simili e inoltre i due sessi vestono in modo indistinguibile. Tuttavia il terrestre ha delle informazioni: i marziani maschi dicono sempre la verità mentre le marziane mentono sempre. Inoltre i venusiani maschi mentono sempre mentre le venusiane dicono sempre la verità. Al bancone il terrestre incontra due membri del club: Ork e Bog che gli dicono:

(1) Ork dice: "Bog viene da Venere."
(2) Bog dice: "Ork viene da Marte."
(3) Ork dice: "Bog è maschio."
(4) Bog dice: "Ork è femmina."

Il terrestre è alquanto confuso! Non sa dire né chi è maschio o femmina né chi è marziano o venusiano. Una volta tornato a casa continua a pensare alla vicenda e decide di risolvere il dilemma. Essendo esperto di ASP scrive un programma che determini sesso e razza di Ork e Bog. Come ha fatto?

Esercizio 12.13 (Ancora il club dei marziani e venusiani). Il nostro amico terrestre ha risolto il sui dilemmi su Ork e Bog e contento. Torna quindi al club sicuro che non si troverà più in imbarazzo. Tuttavia viene a sapere che tra marziani e venusiani sono frequenti le coppie miste. Mentre sorseggia il suo drink viene in contatto proprio con una coppia di alieni D ed F, ma non sapendo distinguere le razze chiede ai due da dove vengano. D gli dice

"Io vengo da Marte." Subito F replica "Non è vero!" Il terrestre è ancora più imbarazzato di prima. Aiutatelo scrivendo un programma ASP che determini se la coppia è multirazziale o meno.

CAPITOLO 13

Soluzione di problemi con ASP

In ASP risulta estremamente naturale utilizzare la tecnica di programmazione generate-and-test per risolvere un dato problema.

L'idea chiave è quella di descrivere (dichiarativamente) un problema caratterizzandone le soluzioni in modo che ogni answer set del programma descriva una soluzione del problema. Sarà quindi l'ASP-solver a trovare una (o tutte) le soluzioni, nel momento stesso in cui calcola gli answer set de programma.

Solitamente quindi, nello scrivere un programma ASP si opera in due fasi: prima si descrive tramite delle regole un insieme di soluzioni candidate (in altre parole, un insieme di potenziali answer set). Poi si introducono dei vincoli (regole senza testa) che invalidano gli answer set che non sono realmente accettabili come soluzioni del problema.

Vediamo in questo capitolo una serie di problemi risolti con questa tecnica.

1. Il *marriage problem*

Questo problema è molto semplice: si conoscono le preferenze di alcuni ragazzi/ragazze. Possiamo rappresentarle tramite questi fatti:

```
likes(andrea,diana).
likes(andrea,federica).
likes(bruno,diana).
likes(bruno,elena).
likes(bruno,federica).
likes(carlo,elena).
likes(carlo,federica).
```

L'obiettivo consiste nel trovare uno o più accoppiamenti delle persone in modo che tutti abbiano un/a compagno/a gradito/a. Considerando che ci sono in tutto tre coppie, potremmo immaginare di risolvere il problema scrivendo un programma Prolog. Infatti il problema può essere risolto dal semplice programma ottenuto aggiungendo ai sette fatti sopra elencati la seguente clausola:

```
coppie(andrea-A,bruno-B,carlo-C) :- likes(andrea,A),
                                    likes(bruno,B),
                                    likes(carlo,C),
                                    A \= B,
                                    A \= C,
                                    B \= C.
```

Conseguentemente, le risposte al goal:

```
?- coppie(X,Y,Z).
```

sarebbero:

```
yes     X = andrea-diana Y = bruno-elena Z = carlo-federica ;
yes     X = andrea-diana Y = bruno-federica Z = carlo-elena ;
yes     X = andrea-federica Y = bruno-diana Z = carlo-elena ;
No
```

Risolviamo ora lo stesso problema scrivendo un programma ASP. È sufficiente utilizzare lo schema di programma della Sezione 3.5 del Capitolo 12, aggiungendo nel corpo delle regole opportuni letterali di dominio (nel caso, `likes`):

```
bigamia(X,Y) :- likes(X,Y), likes(X,Y1),
                coppia(X,Y), coppia(X,Y1), Y!=Y1.
bigamia(X,Y) :- likes(X,Y), likes(X1,Y),
                coppia(X1,Y), X!=X1.
coppia(X,Y) :- likes(X,Y), not bigamia(X,Y).
#hide.
#show coppia(X,Y).
```

Il comando per calcolare gli answer set di questo programma è:

```
lparse marriage.lp | smodels 0
```

Tramite il parametro `0` fornito a smodels richiediamo di trovare tutti gli answer set. Smodels risponderà con tre diversi answer set, ognuno corrispondente ad un accoppiamento ammissibile:[1]

```
smodels version 2.28.  Reading...done
Answer: 1
Stable Model: coppia(carlo,federica) coppia(bruno,elena) coppia(andrea,diana)
Answer: 2
Stable Model: coppia(carlo,elena) coppia(bruno,diana) coppia(andrea,federica)
Answer: 3
Stable Model: coppia(carlo,elena) coppia(bruno,federica) coppia(andrea,diana)
False
Duration:  0.002
Number of choice points:  2
Number of wrong choices:  2
Number of atoms:  21
Number of rules:  34
Number of picked atoms:  12
Number of forced atoms:  0
Number of truth assignments:  130
Size of searchspace (removed):  7 (0)
```

La stampa di `False` indica che non vi sono altri answer set. In chiusura smodels stampa anche delle statistiche relative alla computazione effettuata.

[1] Si noti che, per tradizione, smodels indica gli answer set con la dicitura "Stable Model". Considereremo, per quanto ci riguarda, "answer set" e "modello stabile" come sinonimi.

Si noti che il programma ASP appare più complesso della singola clausola Prolog, tuttavia il programma ASP descritto sopra può essere impiegato senza alcuna modifica per tutte le istanze possibili del problema (immaginate di aggiungere altri fatti `likes` relativi ad altre persone). Al contrario se il numero di coppie cambia (o anche se solo i nomi dei ragazzi/e cambiano) si dovrà modificare il programma Prolog.

2. Il problema delle *N regine*

Riprendiamo il problema delle N regine affrontato nella Sezione 2.1 del Capitolo 9. Il seguente è un programma Smodels che lo risolve:

```
numero(1..n).
1{queen(I,J) : numero(I)}1 :- numero(J).
:- queen(I,J), queen(I,J1), numero(I;J;J1), J<J1.
:- queen(I,J), queen(I1,J1), numero(I;I1;J;J1), J<J1, abs(I1-I)==J1-J.
#hide numero(X).
```

La prima regola dichiara il predicato di dominio `numero`. La regola è parametrica, quindi il valore della costante `n` dovrà essere fornita a lparse sulla linea di comando. Tramite l'atomo `queen(I,J)` rappresentiamo il fatto che vi è una regina nella posizione (I,J) della scacchiera (riga `I` e colonna `J`). La seconda regola impone che per ogni numero `I` (intuitivamente, per ogni colonna della scacchiera) esista un solo valore `J` tale che l'atomo `queen(I,J)` sia vero. Così facendo abbiamo quindi specificato che siamo interessati a soluzioni in cui in ogni colonna vi è una sola regina. Questo è un modo di descrivere un insieme di soluzioni candidate.

Tramite i due constraint (la terza e la quarta regola) invalidiamo parte delle soluzioni descritte dalla seconda regola. Infatti il primo constraint impone che non sia accettabile un answer set in cui, per un qualsiasi `I`, siano contemporaneamente veri gli atomi `queen(I,J)` e `queen(I,J1)` e valga `J<J1`. È un modo per asserire che su una riga non possono esserci due regine. È facile verificare che il secondo constraint elimina ogni answer set in cui vi siano due regine sulla stessa diagonale della scacchiera. L'ultima riga del programma serve semplicemente a evitare la stampa di atomi della forma `numero(X)`, per rendere più leggibili gli answer set.

Il comando per calcolare gli answer set di questo programma è:

```
lparse -c n=4 queens.lp | smodels 0
```

L'output fornito da lparse+smodels sarà invece della forma:

```
Answer: 1
Stable Model: queen(3,1) queen(1,2) queen(4,3) queen(2,4)
Answer: 2
Stable Model: queen(2,1) queen(4,2) queen(1,3) queen(3,4)
False
Duration: 0.003
...
```

Osserviamo che avendo chiesto di produrre tutti gli answer set del programma, otteniamo in risposta due diverse soluzioni. Ognuna di esse corrisponde ad un diverso modo di posizionare $n = 4$ regine su una scacchiera 4×4.

Esercizio 13.1. Si esegua il programma descritto precedentemente per risolvere il problema delle n regine per diversi valori di n.

Si modifichi il programma in modo che vengano calcolate solo le soluzioni in cui una delle regine viene posizionata nella casella $(1, 1)$. Si osservi come varia il numero delle soluzioni (gli answer set) trovate per vari valori di n.

Si modifichi ancora il programma in modo che nessuna regina possa apparire sulla diagonale principale della scacchiera (cioè nessuna regina deve essere in posizione (i, i)).

3. Il problema della *zebra evasa*

Il problema che vogliamo risolvere è il seguente: Vi sono cinque case di cinque colori diversi: rosso, verde, avorio, blu, giallo. Gli inquilini delle case hanno nazionalità diversa. Essi provengono da Giappone, Inghilterra, Norvegia, Ucraina, e Spagna. Ogni inquilino possiede un animale. Gli animali sono: cavallo, chiocciola, zebra, volpe, e cane. Inoltre sappiamo che ogni inquilino beve usualmente una sola delle seguenti bevande: acqua, caffè, tea, latte, aranciata e guida una (sola) auto di una delle seguenti marche: Fiat, Lancia, BMW, Skoda, Audi. Sono note inoltre le seguenti informazioni:

(1) l'inglese vive nella casa rossa;
(2) lo spagnolo ha il cane;
(3) il norvegese vive nella prima casa a sinistra;
(4) nel garage della casa gialla c'è una Skoda;
(5) chi guida la BMW vive nella casa vicina a chi possiede la volpe;
(6) il norvegese vive in una casa vicino alla casa blu;
(7) chi possiede la Lancia possiede anche la chiocciola;
(8) chi guida la Fiat beve aranciata;
(9) l'ucraino beve tea;
(10) il giapponese guida la Audi;
(11) la Skoda è parcheggiata nel garage di una casa vicina alla casa dove c'è il cavallo;
(12) nella casa verde si beve caffè;
(13) la casa verde è immediatamente a destra della casa avorio;
(14) il latte si beve nella casa di mezzo (la terza).

La domanda è "È stata ritrovata una zebra in mezzo alla strada. Chi la ha lasciata scappare?"

Mostriamo in quanto segue una possibile soluzione del problema che prevede di modellare la associazione di una informazione ad una casa. Ogni casa viene rappresentata da un numero da 1 a 5, procedendo da sinistra a destra. Useremo il predicato binario `ha_colore(Num,Colore)` per asserire che la casa numero `Num` ha il colore `Colore`. Similmente faremo con `ha_auto`, `ha_bevanda`, `ha_animale` e `ha_nazione`.
Iniziamo col descrivere i predicati di dominio: Per le case sceglieмо di usare i numeri interi tra 1 e 5:

```
casa(1..5).
```

Per le altre entità usiamo delle costanti:

```
colore(rosso).  colore(verde).  colore(avorio).
colore(blu).  colore(giallo).
nazione(giappone).  nazione(inghilterra).  nazione(norvegia).
nazione(ucraina).  nazione(spagna).
animale(cavallo).  animale(chiocciola).  animale(zebra).
animale(volpe).  animale(cane).
bevanda(acqua).  bevanda(caffe).  bevanda(tea).
bevanda(latte).  bevanda(aranciata).
auto(fiat).  auto(lancia).  auto(bmw).  auto(skoda).  auto(audi).
```

Definiamo poi il concetto di vicinanza tra le case:

```
a_destra(X,X+1) :- casa(X), casa(X+1).
a_sinistra(X+1,X) :- casa(X), casa(X+1).
vicino(X,Y) :- a_destra(X,Y).
vicino(X,Y) :- a_sinistra(X,Y).
```

Passiamo ora ad enumerare le soluzioni candidate. Ciò viene fatto caratterizzando gli answer set nei quali ogni casa ha un solo colore e che ogni colore sia associato ad una sola casa:

```
altro_colore(Casa,Colore) :- casa(Casa), colore(Colore), colore(C),
                             ha_colore(Casa,C), C!=Colore.
altro_colore(Casa,Colore) :- casa(Casa), colore(Colore), casa(C1),
                             ha_colore(C1,Colore), C1!=Casa.
ha_colore(Casa,Colore) :- casa(Casa), colore(Colore),
                          not altro_colore(Casa,Colore).
```

Una caratterizzazione analoga avviene per le auto:

```
altra_auto(Casa,Auto) :- casa(Casa), auto(Auto), auto(A),
                         ha_auto(Casa,A), A!=Auto.
altra_auto(Casa,Auto) :- casa(Casa), auto(Auto), casa(C1),
                         ha_auto(C1,Auto), C1!=Casa.
ha_auto(Casa,Auto) :- casa(Casa), auto(Auto),
                      not altra_auto(Casa,Auto).
```

Per le bevande:

```
altra_bevanda(Casa,Bevanda) :- casa(Casa), bevanda(Bevanda), bevanda(B),
                               ha_bevanda(Casa,B), B!=Bevanda.
altra_bevanda(Casa,Bevanda) :- casa(Casa), bevanda(Bevanda), casa(C1),
                               ha_bevanda(C1,Bevanda), C1!=Casa.
ha_bevanda(Casa,Bevanda) :- casa(Casa), bevanda(Bevanda),
                            not altra_bevanda(Casa,Bevanda).
```

Per le nazioni:

```
altra_nazione(Casa,Nazione) :- casa(Casa), nazione(Nazione), nazione(N),
                               ha_nazione(Casa,N), N!=Nazione.
altra_nazione(Casa,Nazione) :- casa(Casa), nazione(Nazione), casa(C1),
                               ha_nazione(C1,Nazione), C1!=Casa.
ha_nazione(Casa,Nazione) :- casa(Casa), nazione(Nazione),
                            not altra_nazione(Casa,Nazione).
```

Per gli animali:

```
altro_animale(Casa,Animale) :- casa(Casa), animale(Animale), animale(A),
                               ha_animale(Casa,A), A!=Animale.
altro_animale(Casa,Animale) :- casa(Casa), animale(Animale), casa(C1),
                               ha_animale(C1,Animale), C1!=Casa.
ha_animale(Casa,Animale) :- casa(Casa), animale(Animale),
                            not altro_animale(Casa,Animale).
```

Così facendo abbiamo descritto un sopra-insieme delle soluzioni. Il prossimo passo consiste nel eliminare tutti gli answer set che non rappresentano una vera soluzione. Ciò viene fatto asserendo dei constraint. Essi corrispondono proprio alle informazioni che abbiamo elencato nel testo del problema.

```
s1 :- casa(C), ha_nazione(C,inghilterra), ha_colore(C,rosso).
s2 :- casa(C), ha_animale(C,cane), ha_nazione(C,spagna).
s3 :- ha_nazione(1,norvegia).
s4 :- casa(C), ha_auto(C,skoda), ha_colore(C,giallo).
s5 :- casa(C), casa(C1), vicino(C,C1), ha_auto(C,bmw), ha_animale(C1,volpe).
s6 :- casa(C), casa(C1), vicino(C,C1), ha_colore(C,blu),
      ha_nazione(C1,norvegia).
s7 :- casa(C), ha_auto(C,lancia), ha_animale(C,chiocciola).
s8 :- casa(C), ha_auto(C,fiat), ha_bevanda(C,aranciata).
s9 :- casa(C), ha_nazione(C,ucraina), ha_bevanda(C,tea).
s10 :- casa(C), ha_nazione(C,giappone), ha_auto(C,audi).
s11 :- casa(C), casa(C1), vicino(C,C1), ha_auto(C,skoda),
       ha_animale(C1,cavallo).
s12 :- casa(C), ha_colore(C,verde), ha_bevanda(C,caffe).
s13 :- casa(C), casa(C1), ha_colore(C,avorio), ha_colore(C1,verde),
       a_destra(C,C1).
s14 :- ha_bevanda(3,latte).
```

In modo semplice di imporre questi vincoli consiste nell'includere nel programma le due regole:

```
vincoli :- i1, i2, i3, i4 ,i5, i6, i7, i8, i9, i10, i11, i12, i13, i14.
:- not vincoli.
```

Queste ultime istruzioni controllano l'output di smodels:

```
#hide.
#show ha_nazione(X,Y).
#show ha_animale(X,Y).
#show ha_auto(X,Y).
#show ha_bevanda(X,Y).
#show ha_colore(X,Y).
```

Il seguente è l'unico answer set del programma, prodotto da smodels:

```
smodels version 2.28. Reading...done
Answer: 1
Stable Model: ha_colore(1,giallo) ha_colore(2,blu) ha_colore(4,avorio)
ha_colore(5,verde) ha_colore(3,rosso) ha_bevanda(4,aranciata)
ha_bevanda(3,latte) ha_bevanda(2,tea) ha_bevanda(5,caffe)
ha_bevanda(1,acqua) ha_auto(5,audi) ha_auto(1,skoda) ha_auto(2,bmw)
ha_auto(3,lancia) ha_auto(4,fiat) ha_animale(4,cane) ha_animale(1,volpe)
ha_animale(5,zebra) ha_animale(3,chiocciola) ha_animale(2,cavallo)
ha_nazione(4,spagna) ha_nazione(2,ucraina) ha_nazione(1,norvegia)
ha_nazione(3,inghilterra) ha_nazione(5,giappone)
False
...
```

ESERCIZIO 13.2. Si rispova il problema della zebra evasa modellando il problema e le soluzioni tramite un predicato

`associati(NumCasa,Auto,Nazione,Colore,Animale,Bevanda)`

che associa le informazioni relative ad ogni inquilino.

4. Il problema del *map coloring*

Nella Sezione 2.2 del Capitolo 9 abbiamo presentato il problema del map coloring ed abbiamo illustrato come sia possibile risolverlo in Prolog. Affrontiamo ora lo stesso problema con ASP. Assumiamo che la regola `colore(1..n)` descriva tutti i colori disponibili. Fatti della forma `nodo(R)` indicheranno le regioni. Inoltre assumiamo che il grafo sia descritto da un insieme di fatti della forma `arco(R1,R2)` indicanti che le regioni `R1` e `R2` confinano. Il seguente è un programma che trova tutte le n-colorazioni ammesse.

```
colore(1..n).
1{colorato(R,Colore) : colore(Colore)}1 :- nodo(R).
:- arco(R1,R2), colore(Colore), colorato(R1,Colore), colorato(R2,Colore).
#hide.
#show colorato(X,Y).
```

(Ovviamente il programma andrà completato con la descrizione del grafo tramite una serie di fatti ground delle forme `nodo(a)` e `arco(a,b)`.) La prima regola definisce i colori in funzione di un parametro fornito sulla linea di comando (opzione `-c` di lparse). La seconda regola descrive le soluzioni candidate: ci interessano solo le colorazioni in cui ad ogni regione viene assegnato uno ed un solo colore. La terza regola (è un constraint) elimina tutte le soluzioni che assegnerebbero lo stesso colore a due regioni confinanti.

ESERCIZIO 13.3. Si individuino dei grafi inventati o tratti da carte geografiche reali. Si esegua il programma precedente per diversi valori di n, per i vari grafi ottenuti. Si ripetano le stesse prove utilizzando il programma Prolog descritto nella Sezione 2.2 del Capitolo 9 (od un programma Prolog scritto da voi) e si valuti il diverso comportamento dei due risolutori anche in termini di efficienza.

5. Il problema del *circuito hamiltoniano*

Risolvere il problema del circuito hamiltoniano consiste nel trovare (se esiste) un circuito che passi per tutti i nodi di un dato grafo esattamente una volta. Come fatto in precedenza rappresenteremo un grafo tramite dei fatti del tipo: `nodo(a)` e `arco(a,b)`. Un esempio di istanza di questo problema è:

```
nodo(1..9).
arco(1,2).      arco(1,5).      arco(1,9).
arco(3,1).      arco(3,4).      arco(3,8).
arco(5,4).      arco(6,3).      arco(6,7).
arco(2,5).      arco(4,7).      arco(7,1).
arco(8,2).      arco(9,5).      arco(9,6).
```

Il seguente è un semplice programma ASP che risolve il problema del circuito hamiltoniano:

```
1 {hamilton(X,Y) : arco(X,Y)} 1 :- nodo(X).
1 {hamilton(Y,X) : arco(Y,X)} 1 :- nodo(X).
raggiungibile(X) :- nodo(X), hamilton(1,X).
raggiungibile(Y) :- nodo(X), nodo(Y), raggiungibile(X), hamilton(X,Y).
:- not raggiungibile(X), nodo(X).
#hide.
#show hamilton(X,Y).
```

Il circuito viene rappresentato da fatti del tipo `hamilton(X,Y)`, ad indicare che l'arco `arco(X,Y)` è parte della soluzione. Le prime due regole dichiarano che siamo interessati ad answer set in cui per ogni nodo `X` è vero uno ed un solo fatto `hamilton(X,Y)`, ed uno ed un solo fatto `hamilton(Y,X)`. Ovvero, vogliamo che il nostro circuito arrivi (e parta) una sola volta da ogni nodo. Il predicato `raggiungibile(X)` sarà invece soddisfatto se il nodo `X` risulta raggiungibile partendo dal nodo `1` e seguendo un cammino hamiltoniano (dato che cerchiamo un circuito, potremmo scegliere di iniziare da uno qualsiasi dei nodi). Infine, il constraint ha lo scopo di imporre che nella soluzione non esistano nodi che non siano raggiungibili.

Per il semplice grafo riportato sopra la (unica) soluzione ottenuta da smodels è:

```
Answer:  1
Stable Model:  hamilton(1,9) hamilton(2,5) hamilton(3,8) hamilton(4,7)
hamilton(5,4) hamilton(6,3) hamilton(7,1) hamilton(8,2) hamilton(9,6)
False
...
```

Esercizio 13.4. Modificare il precedente programma affinché calcoli i cammini hamiltoniani.

Esercizio 13.5. Si scriva un programma che calcoli i cammini euleriani di un grafo.

6. Il problema della k-*clicca*

Dato un grafo G il problema consiste nel trovare (se esiste) una clicca (un sotto grafo completo) di k nodi. Di nuovo, rappresenteremo un grafo diretto tramite dei fatti del tipo: `nodo(a)` e `arco(a,b)`. Supponiamo di voler operare sulla versione non diretta del grafo. Possiamo allora definire gli archi non diretti come segue:

```
und_arco(X,Y) :- arco(X,Y).
und_arco(X,Y) :- arco(Y,X).
```

La costruzione della clicca procede assegnando ad ogni nodo del grafo una etichetta. La relazione tra etichette diverse da 0 e nodi sarà biiettiva. Definiremo $k+1$ etichette con una regola compatta: `etichetta(0..k)`. Le prime due regole del programma impongono che la etichettatura sia univoca:

```
assegna_etichetta(Nodo,E) :- nodo(Nodo), etichetta(E),
                             not altra_etichetta(Nodo,E).
altra_etichetta(Nodo,E) :- nodo(Nodo), etichetta(E), etichetta(E1),
                           assegna_etichetta(Nodo,E1), E!=E1.
```

Il seguente constraint impone che non ci siano due nodi che abbiano la stessa etichetta positiva. Se due nodi hanno la stessa etichetta questa deve essere 0.

```
:- nodo(N1;N2), etichetta(E1), assegna_etichetta(N1,E1),
   assegna_etichetta(N2,E1), N1!=N2, E1!=0.
```

La etichettatura deve utilizzare tutte le k etichette non nulle:

```
etichetta_assegnata(E) :- nodo(Nodo), etichetta(E), assegna_etichetta(Nodo,E).
:- etichetta(E), E!=0, not etichetta_assegnata(E).
```

Resta da definire quali sono le vere soluzioni del problema: Con le regole precedenti vengono caratterizzati tutti gli answer set in cui le k etichette diverse da 0 sono assegnate a k nodi distinti del grafo. Con il seguente constraint invalidiamo tutti gli answer set in cui tali k nodi non formano un sotto grafo completo:

```
:- nodo(N1;N2), etichetta(E1;E2), assegna_etichetta(N1,E1),
   assegna_etichetta(N2,E2), E1!=0, E2!=0, N1!=N2, not und_arco(N1,N2).
```

Una ultima regola facilita l'individuazione della clicca:

```
clicca(N) :- nodo(N), etichetta(E), assegna_etichetta(N,E), E!=0.
#hide.
#show clicca(N).
```

Un esempio di soluzione calcolata per il grafo visto alla sezione precedente con $k = 3$:

```
smodels version 2.28.  Reading...done
Answer:  1
Stable Model:  clicca(1) clicca(2) clicca(5)
Answer:  2
Stable Model:  clicca(1) clicca(5) clicca(9)
...
```

7. Il problema del *vertex covering*

Il problema del vertex covering viene solitamente formulato come segue: dato un grafo ed un intero k si vuole stabilire se esista un sottoinsieme di k nodi del grafo tale che ogni arco del grafo incida in almeno uno dei k nodi. Il problema può essere risolto semplicemente da un programma ASP che utilizza un'approccio simile a quello adottato per il problema della k-clicca. Il programma seguente infatti si ottiene con una minima modifica da quello illustrato alla Sezione 6:

```
und_arco(X,Y) :- arco(X,Y).
und_arco(X,Y) :- arco(Y,X).
assegna_etichetta(Nodo,E) :- nodo(Nodo), etichetta(E),
                             not altra_etichetta(Nodo,E).
altra_etichetta(Nodo,E) :- nodo(Nodo), etichetta(E), etichetta(E1),
                           assegna_etichetta(Nodo,E1), E!=E1.
:- nodo(N1;N2), etichetta(E1), assegna_etichetta(N1,E1),
   assegna_etichetta(N2,E1), N1!=N2, E1!=0.
etichetta_assegnata(E) :- nodo(Nodo), etichetta(E), assegna_etichetta(Nodo,E).
:- etichetta(E), E!=0, not etichetta_assegnata(E).
:- und_arco(N1,N2), assegna_etichetta(N1,E1),
   assegna_etichetta(N2,E2), E1=0, E2=0.
vertex_cover(N) :- nodo(N), etichetta(E), assegna_etichetta(N,E), E!=0.
#hide.
#show vertex_cover(N).
```

Analogamente al programma della clicca l'algoritmo prevede che vengono assegnate k etichette non nulle a k nodi distinti. Questo insieme di k nodi rappresenta un candidato covering. L'unica differenza rispetto al programma per la clicca risiede nell'ultima regola. Essa invalida tutti i potenziali answer set in cui esiste un arco che non ha nessun estremo tra i nodi etichettati con etichetta non nulla.

Esercizio 13.6. Costruire delle istanze del problema del vertex covering e risolverle utilizzando il programma appena illustrato.

Esercizio 13.7. Scrivere un programma ASP che dato un grafo calcola il valore del minimo k per cui esiste un vertex covering di cardinalità k.

8. Il problema della *allocazione di compiti*

In un'officina artigianale in cui lavorano 4 dipendenti $w1$, $w2$, $w3$ e $w4$, vengono fabbricati 4 diversi prodotti: $p1$, $p2$, $p3$ e $p4$. La seguente tabella indica i differenti profitti associati ad ogni combinazione lavoratore/prodotto.

	p_1	p_2	p_3	p_4
w_1	7	1	3	4
w_2	8	2	5	1
w_3	4	3	7	2
w_4	3	1	6	3

Si vuole determinare il modo migliore di assegnare ogni lavoro ad un operaio, ovvero l'assegnamento che garantisce il maggior profitto globale.

Il seguente programma ASP risolve il problema:

```
lavoro(1..4).
operaio(1..4).

1{ass(W,J) : lavoro(J)}1 :- operaio(W).
:- lavoro(J), operaio(W1;W2), ass(W1,J), ass(W2,J), W1!=W2.

maximize [ ass(1,1)=7, ass(1,2)=1, ass(1,3)=3, ass(1,4)=4,
           ass(2,1)=8, ass(2,2)=2, ass(2,3)=5, ass(2,4)=1,
           ass(3,1)=4, ass(3,2)=3, ass(3,3)=7, ass(3,4)=2,
           ass(4,1)=3, ass(4,2)=1, ass(4,3)=6, ass(4,4)=3 ].
#hide.
#show ass(W,J).
```

dove il generico atomo `ass(W,J)` indica l'assegnamento del lavoro `J` all'operaio `W`.
Come abbiamo detto, smodels procede individuando un primo answer set e stampandolo. Ad esso corrisponde una prima soluzione (ovvero un valore del profitto globale). Si procede poi ricercando altri answer set che migliorino la soluzione, essi vengono stampati solamente se sono effettivamente delle soluzioni migliori dell'ultima ottenuta. Nel caso specifico, ecco un possibile output di smodels:

```
smodels version 2.28.  Reading...done
Answer:  1
Stable Model:  ass(1,4) ass(2,3) ass(3,1) ass(4,2)
{ not ass(1,1), not ass(1,2), ...   ..., not ass(4,4) } min = 46
Answer:  2
Stable Model:  ass(1,4) ass(2,3) ass(3,2) ass(4,1)
{ not ass(1,1), not ass(1,2), ...   ..., not ass(4,4) } min = 45
Answer:  3
Stable Model:  ass(1,4) ass(2,1) ass(3,2) ass(4,3)
{ not ass(1,1), not ass(1,2), ...   ..., not ass(4,4) } min = 39
False
...
```

Si noti come smodels valuti la "bontà" di una soluzione: per la prima soluzione calcolata il "peso" degli atomi falsi è 46 (corrispondentemente il guadagno globale di questa soluzione è 14). Viene poi trovato un answer set che migliora la soluzione: per il secondo answer set il valore degli atomi falsi è 45 mentre il guadagno è 15. Infine un successivo answer set realizza la soluzione ottima con un guadagno globale pari a 21. (Anche in questo caso `False` indica che non vi sono ulteriori soluzioni migliori dell'ultima prodotta.)

9. Il problema del *knapsack*

Il noto problena del knapsack consiste nel massimizzare il valore degli oggetti che si inseriscono in uno zaino con il vincolo che il peso totale degli oggetti non superi la capacità dello zaino. Presentiamo ora un programma per Smodels che risolve questo problema. Possiamo supporre che il programma sia diviso in due file. Un primo file contiene le regole che risolvono il problema:

```
nello_zaino(X) :- oggetto(X), not fuori_dallo_zaino(X).
fuori_dallo_zaino(X) :- oggetto(X), not nello_zaino(X).

valore(X) :- oggetto(X), nello_zaino(X).
peso(X) :- oggetto(X), nello_zaino(X).

non_eccede_capacita :- [ peso(X) : oggetto(X) ] N, capacita_zaino(N).
:- not non_eccede_capacita.

maximize [ valore(X) : oggetto(X) ].

#hide.
#show nello_zaino(X).
```

Le prime due regole descrivono l'insieme dei candidati answer set (ovvero lo spazio delle soluzioni): ci interessano answer set in cui ogni oggetto può essere nello zaino o fuori dallo zaino, ma non entrambe le cose. Le successive due regole definiscono semplicemente i predicati `valore(X)` e `peso(X)`, per gli oggetti nello zaino. La quinta e la sesta regola vincolano la soluzione a non eccedere la capacità dello zaino. Si noti l'uso di un weight constraint, il weight di ogni atomo `peso(X)` sarà definito nel file che descrive la specifica istanza del problema. L'ultima regola impone di cercare l'answer set che massimizza il valore degli oggetti nello zaino (anche in questo caso il weight degli atomi `valore(X)` dipende dalla specifica istanza del problema).

Una particolare istanza del problema sarà descritta in un secondo file. La descrizione dovrà indicare quale è la capacità dello zaino e quali sono gli oggetti (con il loro peso e valore), ad esempio:

```
oggetto(1..6).

capacita_zaino(12).

#weight valore(1) = 2.
#weight valore(2) = 6.
#weight valore(3) = 3.
#weight valore(4) = 8.
#weight valore(5) = 5.
#weight valore(6) = 6.

#weight peso(1) = 4.
#weight peso(2) = 5.
#weight peso(3) = 6.
#weight peso(4) = 5.
#weight peso(5) = 3.
#weight peso(6) = 4.
```

In questo frammento di codice il peso e il valore degli oggetti è stato indicato tramite delle definizioni di weight (si veda pag. 178). Tali definizioni hanno influenza sulla valutazione

della quinta e della settima regola del programma.
Ecco il risultato prodotto da smodels per questa particolare istanza:

```
smodels version 2.28.  Reading...done
Answer:  1
Stable Model:  nello_zaino(1) nello_zaino(4) nello_zaino(5)
{ } min = 15
Answer:  2
Stable Model:  nello_zaino(4) nello_zaino(5) nello_zaino(6)
{ } min = 11
False
...
```

dove si può osservare che ad una prima soluzione in cui l'oggetto 1 viene inserito nello zaino, si preferisce una altra soluzione che sostituisce l'oggetto 1 con il 6.

Vediamo ora come risolvere una versione decisionale del problema del knapsack generalizzato. La generalità consiste nel fatto che in questo caso abbiamo ancora oggetti di vario tipo, ma per ogni tipo vi è a disposizione un numero arbitrario di oggetti uguali ed è possibile inserire nello zaino anche più oggetti dello stesso tipo. Data una capacità dello zaino e un limite inferiore al valore (desiderato) del contenuto dello zaino, il problema consiste nel determinare se sia possibile riempire lo zaino in modo che il valore del contenuto dello zaino raggiunga il minimo richiesto. Come prima ogni oggetto ha un peso e il peso totale non può eccedere la capacità dello zaino. La soluzione che vedremo non fa uso delle definizioni `#weight`, ma sfrutta il fatto che pesi, valori e occorrenze sono tutti numeri interi.

Dovendo descrivere ogni oggetto utilizziamo un atomo `tipo_oggetto(TipoOggetto, Peso, Valore)`, ad esempio come segue:

```
tipo_oggetto(1,  2,  2).     tipo_oggetto( 2,   4,   5).
tipo_oggetto(3,  8, 11).     tipo_oggetto( 4,  16,  23).
tipo_oggetto(5, 32, 47).     tipo_oggetto( 6,  64,  95).
tipo_oggetto(7,128,191).     tipo_oggetto( 8, 256, 383).
tipo_oggetto(9,512,767).     tipo_oggetto(10,1024,1535).
```

Ecco il programma ASP che risolve il knapsack generalizzato:

```
occorrenze(0..capacita_zaino).

occ_oggetto(I,Num_occorrenze,W,C) :- tipo_oggetto(I,W,C),
                                     occorrenze(O), Num_occorrenze = O/W.

1{ in_zaino(I,IO,W,C): occ_oggetto(I,IO,W,C) }1 :- tipo_oggetto(I,W,C).

cond_valore :- valore_min [in_zaino(I,IO,W,C) : occ_oggetto(I,IO,W,C) = IO*C].
:- not cond_valore.

non_eccede_capacita :- [in_zaino(I,IO,W,C) :
                         occ_oggetto(I,IO,W,C) = IO*W] capacita_zaino.
:- not non_eccede_capacita.
```

Il primo fatto fissa (in modo molto permissivo) il dominio per il numeri di occorrenze di ogni oggetto. La seconda regola fissa il numero di occorrenze che ogni oggetto ha nello zaino. La terza regola asserisce che per ogni tipo `I` di oggetto, c'è un solo fatto della forma `in_zaino(I,IO,W,C)` nella soluzione. Questo fatto rappresenta in numero `IO` di oggetti di tipo `I` inseriti nello zaino. Similmente alla precedente soluzione, le ultime regole impongono vincoli sul massimo peso e sul minimo valore del contenuto dello zaino. I valori per due costanti `capacita_zaino` e `valore_min` saranno forniti a `lparse` tramite due opzioni "`-c`" sulla linea di comando.

10. Il problema dei *numeri di Schur*

In questa sezione mostreremo come risolvere in ASP il problema computazionale (molto difficile) del calcolo dei numeri di Schur [**Wei05**].

Vediamo prima come questi numeri sono definiti. Un insieme S di numeri naturali è detto *sum-free* se l'intersezione di S con l'insieme S' così definito $S' = \{x + y : x \in S, y \in S\}$ è vuota. In altre parole un insieme S è sum-free se presi due qualsiasi suoi elementi (anche uguali) S non contiene la loro somma. Il *numero di Schur* $S(P)$ è il più grande numero intero N tale che l'insieme $\{1, \ldots, N\}$ possa essere partizionato in P sottoinsiemi tutti sum-free. Per esempio, l'insieme $\{1, 2, 3, 4\}$ può essere partizionato in due sottoinsiemi: $S_1 = \{1, 4\}$ e $S_2 = \{2, 3\}$. Entrambi sono sum-free: $S'_1 = \{2, 5, 8\}$ e $S'_2 = \{4, 5, 6\}$. Invece è facile verificare che per partizionare l'insieme $\{1, 2, 3, 4, 5\}$ in parti sum-free è necessario dividerlo in almeno tre parti. Quindi abbiamo che $S(2) = 4$. Ad oggi solamente quattro numeri di Schur sono stati calcolati: $S(1) = 1$, $S(2) = 4$, $S(3) = 13$, e $S(4) = 44$. Inoltre solamente delle limitazioni sono note per i numeri di Schur successivi, ad esempio si sa che $160 \leq S(5) \leq 315$.

Il problema che vogliamo risolvere è il problema decisionale: dati P ed N, è vero o meno che $S(P) \geq N$?

In pratica stiamo cercando una funzione $B : \{1, \ldots, N\} \to \{1, \ldots, P\}$ che assegni ogni intero tra 1 e N ad una delle P partizioni in modo che: $(\forall I \in \{1, \ldots, N\})(\forall J \in \{I, \ldots, N\})(B(I) = B(J) \to B(I + J) \neq B(I))$.

Per rendere la funzione B utilizziamo il predicato `inpart(X,P)`. Esso rappresenta il fatto che il numero X è assegnato alla parte P:

```
numero(1..n).
part(1..p).
1{ inpart(X,P) : part(P) }1 :- numero(X).
:- numero(X;Y), part(P), X<=Y, inpart(X,P), inpart(Y,P), inpart(X+Y,P).
```

La regola della terza riga impone che `inpart` sia una funzione dai numeri alle perti (ovvero: per ogni numero vi può essere solo una parte ad esso assegnata). Il vincolo invece impone che per ogni X e Y (non necessariamente diversi), i tre numeri X, Y, e $X + Y$ non possono essere assegnati alla stessa parte.

11. Il problema della *protein structure prediction*

In questa sezione affronteremo una versione semplificata del problema di predizione della struttura di una proteina. Il problema, nella sua generalità prevede di determinare

quale sia la disposizione spaziale di equilibrio raggiunta da una sequenza di N aminoacidi. Tale posizione è determinata dalla configurazione che realizza la minore energia. A sua volta la energia di una particolare disposizione degli aminoacidi è determinata dalle forze attrattive/repulsive che ogni aminoacido esercita sui restanti aminoacidi.

Rispetto alla realtà faremo le seguenti semplificazioni (si veda anche [**CB01**]):

- assumiamo lo spazio essere bidimensionale invece che tridimensionale;
- assumiamo altresì che gli aminoacidi possano disporsi solo in posizioni determinate da una griglia $2N \times 2N$ e che il primo aminoacido sia fissato in posizione (N, N);
- supponiamo che esistano solo due tipi di aminoacidi: h e p (abbreviazioni di *hydrophobic* e *polar*, rispettivamente). Inoltre una coppia di aminoacidi di tipo h che si trovino in due posizioni adiacenti, nella griglia, si attraggono e danno un contributo unitario alla energia globale. Un aminoacido di tipo p invece non interagisce mai con alcun altro aminoacido della sequenza.

Abbiamo quindi una sequenza $S = s_1 \cdots s_n$, con $s_i \in \{h, p\}$. Il problema consiste nel determinare un mapping (detto *folding*) $\omega : \{1, \ldots, n\} \to \mathbb{N}^2$ tale che:

$$(\forall i \in [1, n-1])\ \mathsf{next}(\omega(i), \omega(i+1))$$

e

$$(\forall i, j \in [1, n])\ (i \neq j \to \omega(i) \neq \omega(j))$$

e tale da minimizzare la seguente funzione energia:[2]

$$\sum_{\substack{1 \leq i \leq n-2 \\ i+2 \leq j \leq n}} \mathsf{Pot}(s_i, s_j) \cdot \mathsf{next}(\omega(i), \omega(j))$$

dove $\mathsf{Pot}(s_i, s_j) \in \{0, -1\}$ e $\mathsf{Pot} = -1$ se e solo se $s_i = s_j = h$. Inoltre la condizione $\mathsf{next}(\langle X_1, Y_1 \rangle, \langle X_2, Y_2 \rangle)$ utilizzata nella espressione precedente vale solo tra due posizioni adiacenti della griglia se e solo se $|X_1 - X_2| + |Y_1 - Y_2| = 1$.

Intuitivamente cerchiamo un cammino che passi per N nodi della griglia, partente dalla posizione (N, N) e tale da non passare più volte sulla stessa posizione. Tale cammino deve massimizzare il numero di coppie di aminoacidi h che si trovano in posizioni adiacenti. La Figura 13.1 mostra due possibili configurazioni della sequenza $S = hhphhhph$, una delle quali è la disposizione minimale.

Nella nostra soluzione non conteremo le occorrenze di coppie di h che sono consecutive nella sequenza. Il loro contributo sarà infatti sempre lo stesso per qualsiasi disposizione (ad esempio per la sequenza $S = hhphhhph$ di Figura 13.1, il fatto di avere due sotto sequenze hh e hhh contribuisce sempre con una quota di energia pari a -3). Si noti che una semplice osservazione ci permetterà di includere una immediata ottimizzazione: due aminoacidi contribuiscono alla energia globale solo se si trovano ad una distanza dispari l'uno dall'altro; solo in questo caso infatti possono essere piazzati in due posizioni adiacenti nella griglia. Ci riferiremo a questa proprietà con il termine *odd property*.

Procediamo scegliendo di rappresentare una specifica istanza del problema (la sequenza di h e p) con N fatti, come ad esempio in:

[2] Per comodità, invece di minimizzare una quantità negativa, nel successivo programma massimizzeremo una quantità positiva.

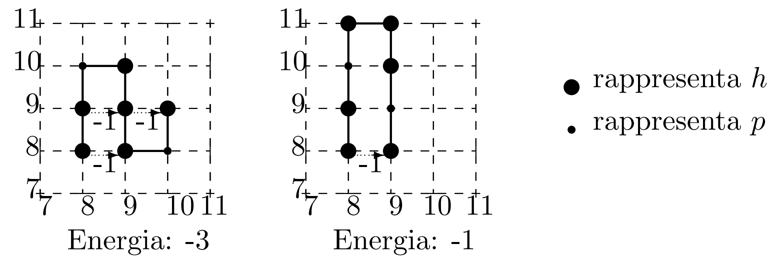


FIGURA 13.1. Due possibili folding di $S = hhphhhph$ ($N = 8$). Quello di sinistra è minimale.

```
prot(1,h).  prot(2,p).  prot(3,p).  prot(4,h).  prot(5,p).
prot(6,p).  prot(7,h).  prot(8,p).  prot(9,p).  prot(10,h).
```

che rappresenta la proteina `hpphpphpph`. Rappresenteremo il fatto che l'i−esimo aminoacido sia in posizione (n, m) con l'atomo `sol(`i`,`n`,`m`)`. Il programma ASP per risolvere il problema del protein folding è il seguente:

```
size(N). %%% dove N è la lunghezza dell'istanza
range(1..2*N). %%% le coordinate nella griglia
1 { sol(I,X,Y) : range(X;Y) } 1 :- prot(I,Amino).
sol(1,N,N) :- size(N).
:- prot(I1,A1), prot(I2,A2), I1<I2, sol(I1,X,Y), sol(I2,X,Y), range(X;Y).
:- prot(I1,A1), prot(I2,A2), I2>1, I1==I2-1, not next(I1,I2).
next(I1,I2) :- prot(I1,A1), prot(I2,A2), I1<I2,
               sol(I1,X1,Y1), sol(I2,X2,Y2), range(X1;Y1;X2;Y2),
               1==abs(Y1-Y2)+abs(X2-X1).
energy_pair(I1,I2) :- prot(I1,h), prot(I2,h),
                      next(I1,I2), I1+2<I2, 1==(I2-I1) mod 2.
maximize{ energy_pair(I1,I2) :  prot(I1,h) :  prot(I2,h) }.
```

Le prime due regole, unitamente al predicato `prot`, definiscono i domini. La terza regola implementa la fase "generate": asserisce che ogni aminoacido occupa una sola posizione della griglia. La quarta regola fissa la posizione del primo aminoacido (in (N, N)). I due successivi vincoli impongono che il cammino non generi cicli e che valga la proprietà `next` tra ogni coppia di aminoacidi consecutivi. Questa proprietà è caratterizzata dalla successiva regola in cui si gestisce anche la *odd property.* Tramite il predicato `energy_pair` si definisce il contributo di energia apportato da una generica coppia di aminoacidi. La massimizzazione della funzione obiettivo viene effettuata dalla ultima istruzione del programma che impone la ricerca dell'answer set che massimizza l'energia globale. (Si noti che essendo ogni contributo unitario, ciò corrosponde a massimizzare il numero di fatti `energy_pair(h,h)` veri nell'answer set).

ESERCIZIO 13.8. Cosa accadrebbe se al programma appena descritto si aggiungesse la seguente regola?

```
sol(2,N,N+1) :- size(N).
```

e se, sapendo che l'istanza è lunga $N = 10$, si sostituisse `range(7..13).` con `range(`x`..`y`).` dove $x = N - \sqrt{N}$ e $y = N + \sqrt{N}$?

12. Esercizi

Esercizio 13.9. Dire cosa accadrebbe se al programma che risolve il problema dei numeri di Schur illustrato nella Sezione 10, si aggiungesse il vincolo

```
1 { inpart(X,P) : numero(X) } n :- part(P).
```

Ci sono differenze tra gli answer set dei due programmi? e nella efficienza dei due programmi? Se sì perché?

Esercizio 13.10. Cosa accadrebbe se invece della modifica proposta nell'Esercizio 13.9 si aggiungessero al programma illustrato nella Sezione 10 le due seguenti regole:

```
:- numero(X), part(P;P1), inpart(X,P), P1<P, not occupata(X,P1).
occupata(X,P) :- numero(X;Y), part(P), Y<X, inpart(Y,P).
```

Dire che significato hanno e che effetto si otterrebbe sugli answer set del programma.

Esercizio 13.11. Scrivere un programma ASP che stabilisca se un numero N è primo.

Esercizio 13.12. Si consideri un grafo rappresentato con fatti del tipo `arco(X,Y)` e `nodo(X)`. Scrivere un programma ASP che determini se il grafo è o meno connesso.

Esercizio 13.13. Scrivere un programma ASP che, dato un grafo diretto $G(\mathit{Nodi}, \mathit{Archi})$, trovi (se esiste) il kernel del grafo (ovvero un insieme di nodi $N \subseteq \mathit{Nodi}$ tale nessuna coppia di nodi in N è connessa da un arco e per ogni nodo in $\mathit{Nodi} \setminus N$ esiste un arco che lo connette ad un nodi in N.

Esercizio 13.14. Scrivere un programma ASP che, dato un grafo non pesato, risolva il problema del *massimo taglio*.

Esercizio 13.15. Scrivere un programma ASP che, dato un grafo e un intero k (non maggiore del numero dei nodi), risolva il problema dell'*independent set*.

Esercizio 13.16. Scrivere un programma ASP che, dato un grafo e un intero k (non maggiore del numero degli archi), risolva il problema del *maximal matching*.

Esercizio 13.17. Si modifichi il programma ASP per il protein structure prediction affinchè possa essere utilizzato per risolvere la versione decisionale del problema. Ovvero, data una sequenza S ed una energia `en`, il programma deve rispondere alla domanda "Può la sequenza S disporsi in modo da raggiungere una energia pari almeno a `en`?"

CAPITOLO 14

Planning

Nei capitolo precedenti abbiamo studiato come rappresentare la conoscenza sul mondo in modo dichiarativo e come effettuare dei ragionamenti basati su tale rappresentazione al fine di risolvere dei problemi.

In quanto studiato non abbiamo però mai affrontato direttamente gli aspetti dinamici che generalmente un dominio del discorso possiede.

Introdurre la dimensione temporale nella descrizione del mondo comporta ovviamente una maggiore difficoltà, tuttavia vi sono situazioni in cui siamo proprio interessati a effettuare dei ragionamenti e inferenze su aspetti e proprietà dinamiche.

In generale quindi possiamo figurare una situazione in cui la descrizione del mondo, lo *stato*, è soggetta a cambiamenti dovuti ad *azioni* che vengono eseguite proprio allo scopo di far evolvere tale rappresentazione. Il problema più tipico che ci si può porre in questo contesto consiste nel chiedersi quale sequenza di azioni sia in grado di far evolvere il mondo (o meglio, la sua rappresentazione) da uno stato iniziale ad uno stato finale. Questo è detto *problema di planning*.

Esiste una vasta letteratura sul planning e molteplici sono le tecniche e gli approcci sviluppati per trattarlo. In questo capitolo ci limiteremo a introdurre i concetti elementari e a studiare come l'answer set programming possa essere un utile strumento per risolvere problemi di planning. Inoltre ci limiteremo al trattamento di situazioni semplificate in cui, ad esempio, la conoscenza sul mondo, e sugli effetti delle azioni, non è incompleta. I concetti che studieremo si possono comunque, debitamente raffinati, trasporre ad un contesto affetto da incompletezza della conoscenza. Rimandiamo per lo studio di questi aspetti alla letteratura sull'argomento (si veda ad esempio [**PMG98, RN03**], ove è possibile trovare anche ulteriori indicazioni bibliografiche).

1. Azioni e loro rappresentazione

In questa sezione descriviamo un linguaggio, $\mathcal{A}$, introdotto in [**GL92**] (si veda anche [**Bar04**, Capitolo 5]) allo scopo di descrivere dichiarativamente le proprietà del mondo (o dominio), delle azioni che nel dominio possono essere compiute, e degli effetti che queste azioni causano.

In particolare, tramite questo formalismo predicheremo su due generi di concetti: i *fluenti* e le *azioni*. Un fluente esprime una proprietà di un oggetto del mondo. Tramite i fluenti vengono descritte le proprietà salienti della realtà modellata. Solitamente il valore (logico) di un fluente dipende dal tempo. Nel nostro approccio dichiarativo, tali proprietà vengono descritte tramite dei letterali (*fluent literal*). Un letterale può essere un fluente o la negazione di un fluente. Lo stato del mondo viene quindi rappresentato dall'insieme di tutti i fluenti. Conseguentemente diremo che un fluente f vale nello stato σ se $f \in \sigma$. Viceversa, diremo

che $\neg f$ vale nello stato σ se $f \notin \sigma$. Le azioni, quando eseguite, hanno l'effetto di modificare lo stato del mondo, ovvero, modificando l'insieme dei fluenti veri, portano il mondo da uno stato ad un altro.

Una *situazione* è la rappresentazione della sequenza delle azioni compiute a partire da uno stato iniziale del mondo. Rappresentando la situazione priva di azioni (quella iniziale) con la lista vuota, allora la scrittura $[a_n, \ldots, a_1]$ denota la situazione ottenuta compiendo le azioni $a_1, \ldots, a_n$, nell'ordine, a partire dallo stato iniziale.

Per descrivere la transizione da uno stato ad il successivo, per effetto dell'esecuzione di una azione usiamo una *effect proposition* della forma:

$$a \ \textbf{causes} \ f \ \textbf{if} \ p_1, \ldots, p_n, \neg q_1, \ldots, \neg q_r$$

dove a è una azione, f un fluent literal e i p_i e i q_j sono fluenti. Questa proposizione asserisce che se i fluenti $p_1, \ldots, p_n, \neg q_1, \ldots, \neg q_r$ sono veri in uno stato σ, o meglio: in una situazione che corrisponde ad uno stato σ, allora estendere tale situazione con la esecuzione della azione a porta ad uno stato in cui sarà vero il fluente f. Un caso particolare si ha quando non vi sono premesse:

$$a \ \textbf{causes} \ f.$$

Per indicare effetti multipli introduciamo la notazione abbreviata:

$$a \ \textbf{causes} \ f_1, \ldots, f_m.$$

Esempio 14.1. Consideriamo un mondo dei blocchi in cui esistono i tre blocchi $\mathtt{b}_1$, $\mathtt{b}_2$ e $\mathtt{b}_3$. Tutti i blocchi sono appoggiati sul tavolo. L'effetto della azione `prendi(X)` può essere espressa dalla seguente effect proposition schematica:

$$\texttt{prendi}(X) \ \textbf{causes} \ \neg\texttt{sul_tavolo}(X), \neg\texttt{mano_vuota}, \neg\texttt{libero}(X), \texttt{afferrato}(X) \tag{1.1}$$

dove, ad esempio, il letterale $\texttt{sul_tavolo}(X)$ denota la condizione di X di essere appoggiato sul tavolo, mentre $\texttt{libero}(X)$ asserisce che su X è possibile appoggiare un altro blocco, ecc.

L'esempio precedente illustra uno *schema* di effect proposition, in quanto l'impiego delle variabili permette di scrivere in modo succinto un insieme di effect proposition ground: la (1.1) è una scrittura abbreviata per le tre effect proposition seguenti:

$$\begin{array}{l}
\texttt{prendi}(b_1) \ \textbf{causes} \ \neg\texttt{sul_tavolo}(b_1), \neg\texttt{mano_vuota}, \neg\texttt{libero}(b_1), \texttt{afferrato}(b_1) \\
\texttt{prendi}(b_2) \ \textbf{causes} \ \neg\texttt{sul_tavolo}(b_2), \neg\texttt{mano_vuota}, \neg\texttt{libero}(b_2), \texttt{afferrato}(b_2) \\
\texttt{prendi}(b_3) \ \textbf{causes} \ \neg\texttt{sul_tavolo}(b_3), \neg\texttt{mano_vuota}, \neg\texttt{libero}(b_3), \texttt{afferrato}(b_3)
\end{array}$$

Osserviamo che un insieme di effect proposition, al fine di descrivere correttamente una funzione di transizione che porta da uno stato al successivo, deve essere consistente. Con ciò vogliamo escludere situazioni in cui sia possibile derivare che un fluente e la sua negazione sono al contempo veri nella stessa situazione. Un esempio di inconsistenza è dato dalle seguenti effect proposition:

$$\begin{array}{c}
a \ \textbf{causes} \ f \\
a \ \textbf{causes} \ \neg f
\end{array}$$

Oltre a descrivere la evoluzione delle situazioni vogliamo anche un mezzo per descrivere delle osservazioni sul mondo modellato. Ovvero un mezzo per asserire che (o chiedere se)

una certa proprietà sia o meno vera in uno stato/situazione. Ciò viene fatto tramite delle scritture delle forme:

$$\textbf{initially } f$$
$$f' \textbf{ after } a_1, \ldots, a_m$$

a indicare, rispettivamente, che il fluente f è vero nella situazione iniziale, e che il fluente f' è vero nella situazione $[a_m, \ldots, a_1]$.

Si noti che in corrispondenza di un insieme di osservazioni possono esistere diversi stati con esse compatibili.

ESEMPIO 14.2. Consideriamo la seguente coppia D di effect proposition:

`carica` **causes** `fucile_carico`
`spara` **causes** ¬`tacchino_vivo` **if** `fucile_carico`

Supponiamo di considerare l'insieme di osservazioni $O = \{$ **initially** `tacchino_vivo`$\}$, allora vi sono due possibili stati iniziali compatibili con O e con le due effect proposition in D. Essi sono $\sigma_0 = \{\texttt{tacchino_vivo}\}$ e $\sigma'_0 = \{\texttt{tacchino_vivo}, \texttt{fucile_carico}\}$.

Utilizzando lo stesso genere di scrittura possiamo anche esprimere delle interrogazioni. Vediamo un esempio.

ESEMPIO 14.3. Considerando la descrizione D del dominio e la osservazione O dell'esempio precedente, ci possiamo chiedere se

$$\neg\texttt{tacchino_vivo} \textbf{ after } \texttt{spara}$$

sia o meno conseguenza della conoscenza che possediamo. La risposta è negativa, in quanto le proposizioni in D non permettono di concludere, in modo indipendentemente dallo stato iniziale, che dopo l'esecuzione della azione `spara` il tacchino non sia vivo. Infatti, in corrispondenza dello stato iniziale σ_0, l'esecuzione di tale azione non porta alla verità del fluente ¬`tacchino_vivo`, perchè il fluente `fucile_carico` non è vero in σ_0.
Considerando invece di O le osservazioni

$$O' = \{ \textbf{ initially } \texttt{tacchino_vivo}, \textbf{ initially } \texttt{fucile_carico}\},$$

e la interrogazione ¬`tacchino_vivo` **after** `carica`, `spara` otteniamo che quest'ultima segue da O' e da D.

Utilizzando questo linguaggio possiamo formalizzare differenti generi di ragionamento sulle azioni. Ad esempio il predire il futuro in base a informazioni sulle azioni e sullo stato iniziale. Oppure dedurre delle informazioni sullo stato iniziale conoscendo la situazione attuale. Più in dettaglio, possiamo parlare di:

Proiezione temporale: in questo tipo di ragionamento assumendo solo delle informazioni della forma **initially** f, siamo interessati solamente a inferire informazioni sul futuro (ipotetico) che si può realizzare a seguito di determinate azioni.

Ragionamento sulla situazione iniziale: questo caso differisce dal precedente in quanto, possedendo informazioni su delle situazioni, vogliamo rispondere a delle interrogazioni che riguardano la sola situazione iniziale.

Assimilazione di osservazioni: questa forma di ragionamento combina le due precedenti: sono previste osservazioni e interrogazioni sia sulla situazione iniziale che sulle successive.

Planning: In questa la forma di ragionamento si assumono una descrizione del mondo D, delle osservazioni O sullo stato iniziale, e una collezione di fluenti $G = \{g_1, \ldots, g_\ell\}$ detta *goal*. Il problema che ci poniamo consiste nel determinare (se esiste) una sequenza $a_1, \ldots, a_n$ di azioni tale che (per ogni j) si abbia g_j **after** $a_1, \ldots, a_n$. Tale sequenza è detta *piano (plan)* per raggiungere il goal G dati D e O.

Esempio 14.4. Consideriamo la descrizione D del dominio e le due osservazioni $O = \{$ **initially** `tacchino_vivo`$\}$ e $O' = \{$ **initially** `tacchino_vivo`, **initially** `fucile_carico`$\}$, degli esempi precedenti. Sia $G = \{\neg$`tacchino_vivo`$\}$ il goal. Allora `spara` è un piano per G rispetto a D e O', ma non rispetto a D e O. Tuttavia, `carica`; `spara` è un piano per G rispetto a D e O.

2. Proiezione temporale e calcolo delle situazioni in ASP

In questa sezione illustreremo una semplice formalizzazione del problema della proiezione temporale in answer set programming.

Tratteremo tre diverse entità: situazioni, azioni e fluenti. Per questa trattazione assumeremo implicitamente che gli oggetti denotati dai simboli `s`, `a` e `f` (o anche `p` e `q`) siano rispettivamente situazioni, azioni e fluenti; similmente, per le variabili utilizzeremo `S`, `A` e `F`. Inoltre, per semplicità, talvolta indicheremo con `res(a,s)` la situazione che si ottienne dalla situazione `s` compiendo l'azione `a`.

Assumiamo innanzitutto di fissare una descrizione del mondo D ed un insieme di osservazioni O. Supponiamo siano espresse nel formalismo introdotto nella sezione precedente. Vediamo quindi come tradurre la conoscenza così espressa in un programma ASP (nel seguito denoteremo un programma ottenuto in questo modo con il simbolo π_1):

- ogni effect proposition della forma

  $$a \textbf{ causes } f \textbf{ if } p_1, \ldots, p_n, \neg q_1, \ldots, \neg q_r$$

  dove f è un fluente, viene resa dalla seguente regola:

  ```
  holds(f,res(a,S)) :- holds(p1,S),...,holds(pn,S),
                       not holds(q1,S),..., not holds(qr,S).     (π1.1)
  ```

  Se invece f è un letterale negativo, ad esempio $\neg g$, allora la effect proposition viene tradotta in:

  ```
  ab(g,a,S) :- holds(p1,S),...,holds(pn,S),
               not holds(q1,S),..., not holds(qr,S).     (π1.2)
  ```

- La traduzione di una osservazione **initially** f sullo stato iniziale, qualora f sia un fluente positivo sarà ottenuta dal solo fatto:

  ```
  holds(f,s0).     (π1.3)
  ```

  Se invece il fluent literal è negativo ($f \equiv \neg g$) nessuna regola viene aggiunta al programma ASP.

- Regole di *inerzia*. Queste regole hanno lo scopo di determinare cosa non sia influenzato dalla esecuzione di una azione. Si adotta la assunzione che un fluente non cambia il suo valore di verità se questo non è esplicitamente stabilito da una effect proposition. Quindi ciò che è vero in uno stato e non è reso falso da una azione sarà vero anche nello stato successivo:

```
holds(F,res(A,S)) :- holds(F,S), not ab(F,A,S).
```
$(\pi_{1.4})$

Si noti l'impiego delle variabili per schematizzare una proprietà di tutti i fluenti.

ESEMPIO 14.5. Riconsideriamo gli esempi della sezione precedente. Supponiamo ora che D sia ancora:

carica **causes** fucile_carico
spara **causes** ¬tacchino_vivo **if** fucile_carico

mentre le osservazioni siano $O'' = \{$ **initially** tacchino_vivo; **initially** ¬fucile_carico$\}$. Ecco il programma ASP corrispondente:

```
holds(fucile_carico, res(carica, S)).
ab(tacchino_vivo, spara, S) :- holds(fucile_carico, S).
holds(tacchino_vivo, s0).
holds(F,res(A,S)) :- holds(F,S), not ab(F,A,S).
```

Si noti che in tutti gli answer set di questo programma occorrono i fatti
holds(tacchino_vivo,[carica]) e holds(fucile_carico,[carica]).
Mentre in nessuno di essi occorre holds(tacchino_vivo,[spara,carica]), e quindi possiamo dire che a seguito delle azioni carica e spara si otterrà che il fluente tacchino_vivo non è vero.

Nella (ri-)formulazione appena descritta viene sempre prodotto un programma che non impiega la negazione esplicita. Dato che gli answer set di questa classe di programmi sono insiemi di atomi essi non potranno contenere fluenti negati.

Introduciamo ora una diversa formulazione del problema di proiezione temporale che impiega la negazione esplicita.

Questa traduzione opera similmente alla precedente (nel seguito denoteremo un programma ottenuto in questo modo con il simbolo π_2):

- ogni effect proposition della forma
$$a \textbf{ causes } f \textbf{ if } p_1, \ldots, p_n, \neg q_1, \ldots, \neg q_r$$
dove f è un fluente, viene resa dalle seguenti due regole:
```
holds(f,res(a,S)) :- holds(p1,S),...,holds(pn,S),
                     -holds(q1,S),...,-holds(qr,S).
ab(f,a,S) :- holds(p1,S),...,holds(pn,S),
             -holds(q1,S),...,-holds(qr,S).
```
$(\pi_{2.1})$

Se invece f è un letterale negativo, sia $\neg g$, allora la effect proposition viene tradotta nella coppia di regole:
```
-holds(g,res(a,S)) :- holds(p1,S),...,holds(pn,S),
                      -holds(q1,S),...,-holds(qr,S).
ab(g,a,S) :- holds(p1,S),...,holds(pn,S),
             -holds(q1,S),...,-holds(qr,S).
```
$(\pi_{2.2})$
- La traduzione di una osservazione **initially** f sullo stato iniziale, qualora f sia un fluente positivo sarà ottenuta dal solo fatto:
```
holds(f,s0).
```
$(\pi_{2.3})$

Mentre se il fluent literal è negativo ($f \equiv \neg g$) la regola utilizzata sarà:

```
-holds(g,s0).                                                    (π2.4)
```

- Regole di inerzia. In questa traduzione le regole di inerzia utilizzate sono due:

```
holds(F,res(A,S)) :- holds(F,S), not ab(F,A,S).
-holds(F,res(A,S)) :- -holds(F,S), not ab(F,A,S).                (π2.5)
```

Riconsideriamo il nostro esempio di lavoro.

Esempio 14.6. Se D è ancora:

carica **causes** fucile_carico
spara **causes** ¬tacchino_vivo **if** fucile_carico

e le osservazioni sono $O'' = \{$ **initially** tacchino_vivo; **initially** ¬fucile_carico$\}$. Ecco il programma ASP nella nuova traduzione:

```
holds(fucile_carico, res(carica, S)).
ab(fucile_carico, carica, S).
-holds(tacchino_vivo, res(spara, S)) :- holds(fucile_carico, S).
ab(tacchino_vivo, spara, S) :- holds(fucile_carico, S).
holds(tacchino_vivo, s0).
-holds(carico, s0).
holds(F,res(A,S)) :- holds(F,S), not ab(F,A,S).
-holds(F,res(A,S)) :- -holds(F,S), not ab(F,A,S).
```

Ora, dato che gli answer set di programmi con negazione esplicita sono insiemi di letterali, avremo che in tutti gli answer set di questo programma occorrono sia gli atomi

holds(tacchino_vivo,[carica]) e holds(fucile_carico,[carica])

che il letterale -holds(tacchino_vivo,[spara,carica]).

È possibile dimostrare [**Bar04**] che i programmi ottenuti applicando entrambe le traduzioni che abbiamo illustrato, godono delle proprietà di correttezza a completezza semantica rispetto alla formulazione nel linguaggio $\mathcal{A}$.

3. Planning e calcolo degli eventi in ASP

Come abbiamo visto, l'approccio presentato nella sezione precedente per trattare il problema di proiezione temporale è basato sulla descrizione di azioni e situazioni (viste come risultato della esecuzione di una sequenza di azioni a partire dallo stato iniziale). Questo approccio risulta adeguato nel caso si voglia verificare la adeguatezza di un dato piano, rispetto ad uno stato iniziale ed a un fluente che si vuole vero nello stato finale. La attività di planning tuttavia consiste nel determinare il piano (o i piani) che permette di realizzare un goal. Nel fare ciò risulta opportuno far entrare esplicitament in gioco una ulteriore dimensione del mondo o della sua descrizione: il tempo.

Il calcolo degli eventi è un formalismo, alternativo al calcolo delle situazioni, che permette di formulare il problema di proiezione temporale esplicitanto lo scorrere del tempo (in modo discreto o continuo). Ciò permette di ragionare su fluenti il cui valore di verità varia nel tempo. In questa sezione adotteremo questo approccio per formalizzare in ASP il problema

del planning. Vedremo che una volta formulato un problema di planning tramite un programma ASP, ognuno dei suoi answer set denoterà un diverso piano adeguato a raggiungere il goal. Questo approccio è solitamente detto *answer set planning* [**Lif99**].

Iniziamo quindi a formulare il problema di proiezione temporale esplicitando il tempo. Successivamente estenderemo la trattazione per affrontare il planning. Partendo da una descrizione del mondo D ed un insieme di osservazioni O (espresse nel formalismo $\mathcal{A}$), ecco come tradurre la conoscenza così espressa in un programma ASP che coinvolga la variabile temporale (nel seguito denoteremo un programma ottenuto con la seguente traduzione con il simbolo π_3):

- ogni effect proposition della forma

  $$a \textbf{ causes } f \textbf{ if } p_1, \ldots, p_n, \neg q_1, \ldots, \neg q_r$$

  dove f è un fluente, viene resa dalla seguente regola:

  ```
  holds(f,T+1) :- occurs(a,T), holds(p1,T),...,holds(pn,T),
                       not_holds(q1,T),..., not_holds(qr,T).
  ab(f,a,T) :- occurs(a,T), holds(p1,T),...,holds(pn,T),          (π3.1)
                  not_holds(q1,T),...,not_holds(qr,T).
  ```

  Se invece f è un letterale negativo, sia $\neg g$, allora la effect proposition viene tradotta nella coppia di regole:

  ```
  not_holds(g,T+1) :- occurs(a,T), holds(p1,T),...,holds(pn,T),
                        not_holds(q1,T),...,not_holds(qr,T).
  ab(g,a,T) :- occurs(a,T), holds(p1,T),...,holds(pn,T),          (π3.2)
                  not_holds(q1,T),...,not_holds(qr,T).
  ```

- La traduzione di una osservazione **initially** f sullo stato iniziale, qualora f sia un fluente positivo sarà ottenuta dal solo fatto:

  ```
  holds(f,1).                                                     (π3.3)
  ```

  Mentre se il fluent literal è negativo ($f \equiv \neg g$) la regola utilizzata sarà:

  ```
  not_holds(g,1).                                                 (π3.4)
  ```

- Regole di inerzia. Anche in questa traduzione le regole di inerzia utilizzate sono due:

  ```
  holds(F,T+1) :- occurs(A,T), holds(F,T), not ab(F,A,T).
  not_holds(F,T+1) :- occurs(A,T), not_holds(F,T), not ab(F,A,T).  (π3.5)
  ```

Si noti che il programma risultante dalla traduzione non contiene negazione esplicita. Abbiamo infatti utilizzato a tal fine il predicato ausiliario `not_holds`. Questo approccio può essere impiegato per compiere proiezione temporale nello stile del calcolo degli eventi. In realtà tale approccio permette anche di effettuare ragionamenti su fluenti lungo tutta la linea temporale.

Vediamo ora come modificare la traduzione precedente per poterla utilizzare per il planning. L'ingrediente chiave è introduzione della lunghezza ℓ del piano. Conseguentemente focalizzeremo l'attenzione sulle sequenze di azioni di tale lunghezza. Per ottenere ciò introdurremo dei vincoli che escluderanno tutte le soluzioni (ovvero, gli answer set) che comportano l'esecuzione di azioni oltre il limite temporale ℓ.

La nuova traduzione sarà ottenuta dalla precedente aggiungendo quindi opportune regole. Le prime due impongono che ad ogni istante di tempo una ed una sola azione venga eseguita:

```
not_occurs(A,T) :- occurs(B,T), A!=B.
occurs(A,T) :- T <= ℓ, not not_occurs(A,T).
```
$(\pi_{3.6})$

La seguente regola (si noti che è un constraint) invece traduce il generico fluente `h` del goal che si vuole raggiungere dopo l'esecuzione di ℓ azioni (ovvero, al tempo $\ell + 1$):

```
:- not holds(h,ℓ+1).
```
$(\pi_{3.7})$

Nota 14.1. Osserviamo che con la formulazione presentata possiamo accettare come validi solamente piani di lunghezza esattamente ℓ. Per ovviare a questa restrizione possiamo prevedere l'esistenza di una azione particolare, denotiamola con `no-op`, che non ha alcun effetto sullo stato del mondo, ma che permette semplicemente lo scorrere del tempo senza che alcuna azione significativa venga eseguita.

4. Una estensione: la esecuzione condizionata

Abbiamo visto come tramire le effect proposition vengono descritti gli effetti delle azioni. Abbiamo anche visto che è possibile enunciare quali requisiti devono essere soddisfatti all'atto della esecuzione della azione affinché tali effetti si producano. Il formalismo $\mathcal{A}$ tuttavia non permette di imporre delle condizioni sulla eseguibilità delle azioni. Descriviamo ora una estensione di $\mathcal{A}$ che permette ciò grazie alla asserzione di opportune *condizioni di eseguibilità* della seguente forma:

$$\textbf{executable } a \textbf{ if } p_1, \ldots, p_n, \neg q_1, \ldots, \neg q_r$$

dove a è una azione e ogni p_i e q_j è un fluente. Tale condizione asserisce che la azione a può essere compiuta solo se tutti i fluenti indicati sono veri. A seguito di questa estensione del formalismo $\mathcal{A}$ avremo che la descrizione del mondo sarà data in termini di un insieme di effect proposition e di condizioni di eseguibilità. Tale descrizione definirà, come in precedenza, una funzione di transizione tra gli stati del mondo.

Esempio 14.7. Consideriamo la seguente descrizione:

`guidare_fino_aeroporto` **causes** `in_aeroporto`
executable `guidare_fino_aeroporto` **if** `possiedi_auto`
`compra_auto` **causes** `possiedi_auto`

Se lo stato corrente fosse $\sigma_1 = \{\}$, allora la azione `guidare_fino_aeroporto` non sarebbe eseguibile. Viceversa, se lo stato corrente fosse $\sigma_2 = \{$`possiedi_auto`$\}$, allora tale azione sarebbe eseguibile.

Descriviamo ora le modifiche da apportare alle tre traduzioni π_1, π_2 e π_3 necessarie per trattare le condizioni di eseguibilità:

π_1: Al programma π_1, per ogni condizioni di eseguibilità aggiungiamo la regola

```
executable(A,S) :- holds(p1,S),...,holds(pn,S),
                   not holds(q1,S),..., not holds(qr,S).
```

inoltre aggiungiamo il letterale `executable(a,S)` al corpo di ogni regola del tipo $(\pi_{1.1})$ e $(\pi_{1.2})$, mentre al corpo di ogni regola del tipo $(\pi_{1.4})$ aggiungiamo il letterale `executable(A,S)`.

π_2: Al programma π_2, per ogni condizioni di eseguibilità aggiungiamo la regola

```
executable(A,S) :- holds(p1,S),...,holds(pn,S),
                   -holds(q1,S),..., -holds(qr,S).
```

inoltre aggiungiamo il letterale `executable(a,S)` al corpo di ogni regola del tipo ($\pi_{2.1}$) e ($\pi_{2.2}$), mentre al corpo di ogni regola del tipo ($\pi_{2.5}$) aggiungiamo il letterale `executable(A,S)`.

π_3: Al programma π_3, per ogni condizioni di eseguibilità aggiungiamo la regola

```
executable(A,T) :- holds(p1,T),...,holds(pn,T),
                   not_holds(q1,T),..., not_holds(qr,T).
```

inoltre aggiungiamo il seguente constraint:

```
:- occurs(A,T), not executable(A,T).
```

L'impiego di condizioni di eseguibilità del genere appena descritto, solitamente si accompagna alla assunzione implicita che le uniche azioni eseguibili in un determinato stato sono tutte e sole quelle che risultano abilitate da qualche condizione di eseguibilità. Viceversa, se una azione non è esplicitamente abilitata da (almeno) una condizione allora non può essere eseguita. Tale approccio è adottato anche nel linguaggio STRIPS [**PMG98**].

Qualora si voglia adottare un approccio complementare, ovvero in cui tutte le azioni risultano abilitate se non diversamente specificato, si impiega una asserzione del tipo:

$$\textbf{impossible } a \textbf{ if } p_1, \ldots, p_n, \neg q_1, \ldots, \neg q_r$$

con l'ovvio significato.

5. Esempi di problemi di planning

5.1. Il mondo dei blocchi: un esempio da STRIPS a ASP. Analizziamo ora in maggior dettaglio un classico esempio di planning: il mondo dei blocchi (si veda anche l'Esempio 14.1). Daremo prima una trattazione utilizzando per semplicità il linguaggio $\mathcal{A}$, tuttavia si noti che tale presentazione può essere immediatamente tradotta nel linguaggio di STRIPS. Successivamente presenteremo il corrispondente programma ASP.

Le entità di cui parleremo sono i blocchi. Il problema consiste nel determinare un piano che, se realizzato, trasformi lo stato iniziale del mondo in uno stato che soddisfi i requisiti specificati nel goal. Le azioni possibili sono quattro. Esse permettono di afferrare un blocco posizionato sul tavolo oppure toglierlo dalla cima di una pila di blocchi, oppure di appoggiare un blocco sul tavolo o sopra un altro blocco.

Utilizzeremo il predicato `blocco`(X) per descrivere l'insieme dei blocchi. In quanto segue useremo le variabili per denotare in generalità i blocchi.

I fluenti che entreranno nella descrizione sono: `sopra`(X,Y), `sul_tavolo`(X), `libero`(X), `mano_vuota` e `afferrato`(X).

Le condizioni di eseguibilità sono le seguenti:

executable `prendi`(X) **if** `libero`(X), `sul_tavolo`(X), `mano_vuota`
executable `appoggia`(X) **if** `afferrato`(X)
executable `impila`(X,Y) **if** `afferrato`(X), `libero`(Y)
executable `prendi_da_pila`(X,Y) **if** `sopra`(X,Y), `libero`(X), `mano_vuota`

Le effect proposition sono le seguenti:

$\texttt{prendi}(X)$ **causes** $\neg\texttt{libero}(X), \neg\texttt{sul_tavolo}(X), \neg\texttt{mano_vuota}, \texttt{afferrato}(X)$
$\texttt{appoggia}(X)$ **causes** $\neg\texttt{afferrato}(X), \texttt{libero}(X), \texttt{sul_tavolo}(X), \texttt{mano_vuota}$
$\texttt{impila}(X, Y)$ **causes** $\neg\texttt{afferrato}(X), \neg\texttt{libero}(Y), \texttt{libero}(X),$
$\texttt{sopra}(X, Y), \texttt{mano_vuota}$
$\texttt{prendi_da_pila}(X, Y)$ **causes** $\texttt{afferrato}(X), \neg\texttt{sopra}(X, Y), \texttt{libero}(Y),$
$\neg\texttt{libero}(X), \neg\texttt{mano_vuota}$

Descriviamo ora lo stato iniziale del mondo in cui sono presenti i blocchi `a`, `b`, `c` e `d`:

initially $\texttt{libero}(\texttt{c})$
initially $\texttt{libero}(\texttt{a})$
initially $\texttt{sul_tavolo}(\texttt{c})$
initially $\texttt{sul_tavolo}(\texttt{b})$
initially $\texttt{sopra}(\texttt{a}, \texttt{b})$
initially $\texttt{mano_vuota}$

con la ulteriore assunzione che solo i fluenti veri sono specificati: gli altri sono da intendersi falsi. Il goal: $\{\texttt{sopra}(\texttt{a}, \texttt{c}), \texttt{sopra}(\texttt{c}, \texttt{b})\}$. In Figura 14.1 sono illustrati lo stato iniziale e lo stato finale relativi a questa istanza del problema dei blocchi.

FIGURA 14.1.

Prima di affrontare la formalizzazione del problema in ASP, a titolo di esempio presentiamo qui di seguito una sua formulazione nello stile di STRIPS. Sarà così immediato rendersi conto, al di là delle differenze sintattiche, delle minime diversità concettuali tra la rappresentazione nel linguaggio di STRIPS e nel formalismo $\mathcal{A}$.

Nella rappresentazione di STRIPS le azioni vengono descritte indicando (in una unica asserzione) una lista di precondizioni (alla eseguibilità), i fluenti che diventeranno veri e quelli che non saranno più veri dopo la esecuzione della azione. Un esempio di linguaggio STRIPS-like è il seguente [**PMG98**]:

$\texttt{prendi}(X)$: **preconditions** $[\texttt{libero}(X), \texttt{sul_tavolo}(X), \texttt{mano_vuota}]$
delete list $[\texttt{libero}(X), \texttt{sul_tavolo}(X), \texttt{mano_vuota}]$
add list $[\texttt{afferrato}(X)]$
$\texttt{appoggia}(X)$: **preconditions** $[\texttt{afferrato}(X)]$
delete list $[\texttt{afferrato}(X)]$
add list $[\texttt{libero}(X), \texttt{sul_tavolo}(X), \texttt{mano_vuota}]$
$\texttt{impila}(X)$: **preconditions** $[\texttt{afferrato}(X), \texttt{libero}(Y)]$
delete list $[\texttt{afferrato}(X), \texttt{libero}(Y)]$
add list $[\texttt{libero}(X), \texttt{sopra}(X, Y), \texttt{mano_vuota}]$

`prendi_da_pila`(X): **preconditions** [`sopra`(X, Y), `libero`(X), `mano_vuota`]
delete list [`sopra`(X, Y), `libero`(X), `mano_vuota`]
add list [`afferrato`(X), `libero`(Y)]

Nel modellare il problema di planning in ASP faremo uso delle tecniche di programmazione studiate nel Capitolo 12. In particolare, dato che eseguendo, in uno specifico istante di tempo, azioni diverse si generano piani (potenziali) diversi, useremo la tecnica di scelta per esplorare ogni possibile sequenza di azioni ammissibile. Ogni answer set del programma che scriveremo rappresenterà un particolare piano in grado di realizzare il goal.

Ecco la specifica del problema di planning in ASP. In quanto segue adotteremo un approccio che si richiama al calcolo degli eventi illustrato nella Sezione 3. Tuttavia al fine di conferire una maggiore generalità alla trattazione distingueremo tra aspetti legati alla specifica istanza del problema di planning, e aspetti indipendenti dal dominio del discorso. Inseriremo quindi una sorta di "indirezione" tra le regole ASP che surrogheranno il motore inferenziale di un ipotetico planner e le regole che descrivono la istanza del problema.

Regole dipendenti dal dominio.

- Le seguenti regole definiscono blocchi, fluenti e azioni:

```
blocco(a)
blocco(b)
blocco(c)
blocco(d)

fluente(on(X,Y)) :- blocco(X), blocco(Y).
fluente(sul_tavolo(X)) :- blocco(X).
fluente(libero(X)) :- blocco(X).
fluente(afferrato(X)) :- blocco(X).
fluente(mano_vuota).

azione(prendi(X)) :- blocco(X).
azione(appoggia(X)) :- blocco(X).
azione(impila(X,Y)) :- blocco(X), blocco(Y).
azione(prendi_da_pila(X,Y)) :- blocco(X), blocco(Y).
```

- Le condizioni di eseguibilità vengono espresse nel seguente modo. Si noti che ogni prerequisito alla eseguibilità di una azione figura in un singolo fatto. Vedremo in seguito come la parte indipendente dal dominio gestisce questi fatti.

```
exec(prendi(X), libero(X)).
exec(prendi(X), sul_tavolo(X)).
exec(prendi(X), mano_vuota).

exec(appoggia(X), afferrato(X)).

exec(impila(X,Y), afferrato(X)).
exec(impila(X,Y), libero(Y)).

exec(prendi_da_pila(X,Y), libero(X)).
exec(prendi_da_pila(X,Y), sopra(X,Y)).
exec(prendi_da_pila(X,Y), mano_vuota).
```

Le effect proposition vengono rese nel seguente modo. Si noti che questo approccio è adeguato solamente per effect proposition del tipo a **causes** f. Nel caso siano presenti delle proposition condizionate sarà necessaria una diversa traduzione (per una trattazione dettagliata si veda, ad esempio, [**Bar04**]).

```
causa(prendi(X),neg(sul_tavolo(X))).
causa(prendi(X),neg(libero(X))).
causa(prendi(X),afferrato(X)).
causa(prendi(X),neg(mano_vuota)).

causa(appoggia(X),sul_tavolo(X)).
causa(appoggia(X),libero(X)).
causa(appoggia(X),neg(afferrato(X))).
causa(appoggia(X),mano_vuota).

causa(impila(X,Y),neg(afferrato(X))).
causa(impila(X,Y),neg(libero(Y))).
causa(impila(X,Y),libero(X)).
causa(impila(X,Y),mano_vuota).
causa(impila(X,Y),sopra(X,Y)).

causa(prendi_da_pila(X,Y),afferrato(X)).
causa(prendi_da_pila(X,Y),libero(Y)).
causa(prendi_da_pila(X,Y),neg(libero(X))).
causa(prendi_da_pila(X,Y),neg(mano_vuota)).
causa(prendi_da_pila(X,Y),neg(sopra(X,Y))).
```

La descrizione dello stato iniziale consiste in quanto segue:

```
iniziale(mano_vuota).
iniziale(libero(a)).
iniziale(libero(c)).
iniziale(sul_tavolo(c)).
iniziale(sul_tavolo(b)).
iniziale(sopra(a,b)).
```

mentre il goal viene descritto da:

```
finale(sopra(a,c)).
finale(sopra(c,b)).
```

Regole indipendenti dal dominio. Il primo insieme di fatti non dipendenti dalla specifica istanza del problema caratterizza gli istanti temporali:

```
tempo(1).
  ⋮
tempo(ℓ).
```

Ove il parametro ℓ denota la lunghezza del piano. Ricordiamo che utilizzando le costanti in lparse possiamo riassumere i precedenti fatti con `tempo(1..l).` e fornire il valore di `l` sulla linea di comando. Il goal viene gestito dalle regole:

```
not_goal(T) :- tempo(T), finale(F), holds(F,T).
goal(T) :- tempo(T), not holds(F,T).
:- not goal(ℓ).
```

l'ultima delle quali (un constraint) elimina tutti gli answer set in cui al tempo ℓ almeno uno dei fluenti del goal non è vero. Le successive due regole mettono in relazione ogni fluente `f` con il termine `neg(f)` che ne denota la negazione nella precedente rappresentazione della istanza del problema:

```
opposto(F,neg(F)).
opposto(neg(F),F).
```

Le condizioni di eseguibilità e le effect proposition vengono gestite dal seguente frammento di programma:

```
not_eseguibile(A,T) :- exec(A,F), not holds(F,T).
eseguibile(A,T) :- T < ℓ, not not_eseguibile(A,T).
holds(F,T+1) :- T < ℓ, eseguibile(A,T), occurs(A,T), causa(A,F).
```

Ecco la regola che gestisce l'inerzia:

```
holds(F,T+1) :- opposto(F,G), T < ℓ, holds(F,T), not holds(G,T+1).
```

Le seguenti ulteriori regole asseriscono che ad ogni stante di tempo una sola azione viene eseguita. Inoltre ogni azione eseguibile viene perseguita in un (diverso) answer set e una azione non può essere eseguita se non è eseguibile.

```
occurs(A,T) :- azione(A), tempo(T), not goal(T), not not_occurs(A,T).
not_occurs(A,T) :- azione(A), azione(B), tempo(T), occurs(B,T), A!=B.
:- azione(A), tempo(T), occurs(A,T), not eseguibile(A,T).
```

Si osservi che non abbiamo fatto uso di alcuna azione fittizia "`no-op`" (si veda la Nota 14.1) per poter ottenere answer set che codifichino piano più corti di ℓ. Invece di adottare questo espediente è stato inserito il naf-literal `not goal(T)` che impedisce l'esecuzione di una azione se il goal è stato raggiunto.

Esercizio 14.1. Come abbiamo menzionato, il modo di rendere le effect proposition utilizzato nell'esempio del mondo dei blocchi non è sufficientemente generale da poter essere impiegato nella gestione di effect proposition condizionali. Si modifichi la tecnica illustrata nell'esempio del mondo dei blocchi in modo da poter gestire anche tale genere di effect proposition.

5.2. Il mondo dei blocchi in Prolog. Allo scopo di evidenziare ancora una volta sia le differenze che le similarità degli approcci alla programmazione dichiarativa basati su Prolog e su ASP, in quanto segue riportiamo una possibile soluzione in Prolog del problema di planning trattato nella sezione precedente.

È ormai chiaro che una delle principali differenze si riscontra nel genere di risposta che i due approcci forniscono. Mentre un ASP-solver produce uno o più answer set (cioè modelli stabili del programma logico), Prolog fornisce risposte basandosi sul meccanismo di istanziazione delle variabili del goal (Prolog).

Il programma Prolog che segue risolve il problema utilizzando una visita di uno spazio degli stati. Gli stati dello spazio di ricerca corrispondono appunto agli stati che nel mondo dei blocchi possono essere realizzati eseguendo le azioni. Trovare una soluzione consiste quindi

nel trovare un cammino dallo stato iniziale ad uno stato finale (si veda in merito anche il Capitolo 10).

Le entità in gioco saranno descritte mediante dei fatti:

```
blocco(a).          blocco(b).
blocco(c).          blocco(d).
```

Le rappresentazioni delle due configurazioni iniziale e finale sono rese tramite delle liste Prolog che indicano i fluenti veri:

```
iniziale([libero(a), libero(c), mano_vuota,
          sopra(a,b), sul_tavolo(b), sul_tavolo(c)]).
finale([sopra(a,c), sopra(c,b)]).
```

Il programma implementerà una ricerca depth-first. Realizzeremo quindi un *depth-first planner*. Al fine di controllare se la visita di uno stato è già stata effettuata (evitando così che la visita compia dei cicli) effettueremo un confronto tra i fluenti che descrivono il candidato ad essere il prossimo stato e quelli già visitati. Per facilitare il confronto manteniamo sempre ordinata la lista dei fluenti.

Ecco il motore dell'algoritmo:

```
df_plan_search(StatoIniziale,StatoFinale,Piano) :-
            ricerca(StatoIniziale,StatoFinale,[StatoIniziale],Piano).

ricerca(Goal,Goal,Visitati,[]).
ricerca(StatoAttuale,Goal,Visitati,[Azione|Azioni]) :-
            azione_legale(Azione,StatoAttuale),
            prossimo_stato(Azione,StatoAttuale,ProssimoStato),
            not member(ProssimoStato,Visitati),
            ricerca(ProssimoStato,Goal,[ProssimoStato|Visitati],Azioni).
```

Le condizioni di eseguibilità sono rese tramite `azione_legale`:

```
azione_legale(prendi(Blocco),Stato) :-
            verifica(sul_tavolo(Blocco),Stato),
            verifica(libero(Blocco),Stato),
            verifica(mano_vuota,Stato).
azione_legale(appoggia(Blocco),Stato) :-
            verifica(afferrato(Blocco),Stato).
azione_legale(impila(Blocco,Blocco2),Stato) :-
            verifica(afferrato(Blocco),Stato),
            verifica(libero(Blocco2),Stato).
azione_legale(prendi_da_pila(Blocco1,Blocco2),Stato) :-
            verifica(mano_vuota,Stato),
            verifica(sopra(Blocco1,Blocco2),Stato),
            verifica(libero(Blocco2),Stato).

verifica(Fluente,Stato) :- member(Fluente, Stato).
```

mentre `prossimo_stato` riflette le effect condition:

```
prossimo_stato(prendi(Blocco),Stato,Prossimo) :-
            asserisci([afferrato(Blocco)],Stato,Stato1),
            falsifica([libero(Blocco),sul_tavolo(Blocco),mano_vuota],
                      Stato1,Prossimo).
prossimo_stato(appoggia(Blocco),Stato,Prossimo) :-
            asserisci([libero(Blocco),sul_tavolo(Blocco),mano_vuota],
                      Stato,Stato1),
            falsifica([afferrato(Blocco)],Stato1,Prossimo).
prossimo_stato(impila(Blocco1,Blocco2),Stato,Prossimo) :-
            asserisci([libero(Blocco1),sopra(Blocco1,Blocco2),mano_vuota],
                      Stato,Stato1),
            falsifica([afferrato(Blocco1),libero(Blocco2)],Stato1,Prossimo).
prossimo_stato(prendi_da_pila(Blocco1,Blocco2),Stato,Prossimo) :-
            asserisci([afferrato(Blocco1),libero(Blocco2)],Stato,Stato1),
            falsifica([libero(Blocco1),sopra(Blocco1,Blocco2),mano_vuota],
                      Stato1,Prossimo).

%asserisci(+FList, +List, -List2) % Esercizio

%falsifica(+FList, +List, -List2) % Esercizio
```

Esercizio 14.2. Si fornisca la definizione dei predicati `asserisci` e `falsifica`. Si tenga presente che gli stati sono liste ordinate di fluenti.

Esercizio 14.3. Si modifichi il programma Prolog precedente al fine di ottimizzare la scelta della prossima azione da eseguire. Il criterio da adottare sarà quello di ritenere più promettente la mossa che porta ad uno stato "più vicino" al goal. [SUGGERIMENTO: si considerino le varie strategie di visita descritte nel Capitolo 10.]

5.3. Problema delle *torri di Hanoi*. La tecnica utilizzata per il mondo dei blocchi risulta sufficientemente generale da permettere il trattamento di problemi di complessità non banale. Tuttavia non è sempre necessario adottare un approccio così strutturato. Per problemi semplici è possibile operare in modo più diretto affrontando la specifica istanza del problema, eventualmente sfruttando delle funzionalità offerte da uno specifico ASP-solver. Riportiamo qui un programma che sfrutta l'espressività del linguaggio accettato da lparse. Esso risolve il problema delle torri di Hanoi, con soli 4 dischi, utilizzando il planning.

Al solver andrà indicato il numero di azioni da compiere, con il comando

```
lparse -c n=16 file.lp | smodels
```

si noti che il passo i si compie al tempo i. Ecco gli oggetti del dominio:

```
tempo(1..n).
passo(1..n-1).

perno(perno1).
perno(perno2).
perno(perno3).
disco(1..4).
```

Le seguenti regole definiscono le mosse (ovvero le azioni) ammesse e i fluenti `infilato` e `in_cima`:

```
{ mossa(Disco,From,To,Step) } :- disco(Disco), perno(From;To), passo(Step),
                                 infilato(Disco,From,Step),
                                 in_cima(Disco,From,Step).
infilato(D,P1,S+1) :- disco(D), perno(P1;P2), passo(S), mossa(D,P2,P1,S).
in_cima(D,P1,S+1) :- disco(D), perno(P1;P2), passo(S), mossa(D,P2,P1,S).
```

Le regole di inerzia sono:

```
cambia(D,S) :- mossa(D,P1,P2,S), disco(D), perno(P1;P2), passo(S).
infilato(D,P,S+1) :- infilato(D,P,S), not cambia(D,S),
                     passo(S), disco(D), perno(P).
```

Una condizione per la eseguibilità di una mossa è che non si cerchi di spostare un disco da un perno allo stesso perno:

```
:- mossa(D,P1,P2,S), disco(D), perno(P1;P2), passo(S), P1=P2.
```

Un solo movimento di un disco è possibile in ogni mossa/tempo:

```
:- cambia(D,S), cambia(D1,S), D < D1, disco(D;D1), passo(S).
:- infilato(D,P,T), infilato(D,P1,T), disco(D),
   perno(P;P1), tempo(T), P < P1.
```

Le seguenti regole vincolano l'insieme di mosse ammesse e al contempo correlano i fluenti `infilato` e `in_cima`:

```
{ in_cima(D,P,T):disco(D) } :- perno(P), tempo(T).
:- in_cima(D,P,T), perno(P), disco(D;D1),
   tempo(T), infilato(D1,P,T), D1 < D.
:- not infilato(D,P,T), in_cima(D,P,T), disco(D), perno(P), tempo(T).
:- not in_cima(1,P,T), not in_cima(2,P,T),
   not in_cima(3,P,T), not in_cima(4,P,T),
   infilato(D,P,T), disco(D), perno(P), tempo(T).
```

In conclusione, la descrizione dello stato iniziale e del goal:

```
infilato(4, perno1, 1).          infilato(3, perno1, 1).
infilato(2, perno1, 1).          infilato(1, perno1, 1).

compute 1 { infilato(4, perno3, n), infilato(3, perno3, n),
            infilato(2, perno3, n), infilato(1, perno3, n) }.
```

Esercizio 14.4. Realizzare un programma ASP che, utilizzando il planning, sia in grado di risolvere (qualora sia possibile farlo) in al più n mosse il problema delle torri di Hanoi con m dischi. (Sia n che m saranno due costanti fornite a lparse sulla linea di comando.)

6. Esercizi

Esercizio 14.5. Si scriva un programma di planning per il seguente problema dei cinque mariti gelosi:

> Cinque coppie si trovano su di un'isola non collegata alla terra ferma. Sull'isola c'è una barca che può trasportare al più tre persone alla volta. I mariti sono così gelosi che non sopportano di lasciare la propria consorte né sulla barca né su una delle due rive assieme ad altri uomini se loro non sono presenti. Si trovi una strategia che permette di portare tutti sulla terraferma senza incappare in crisi di gelosia.[1]

Esercizio 14.6. Si scriva un programma di planning per il problema del lupo, della capra e del cavolo: un uomo deve traghettare i tre sulla sponda opposta di un fiume ma ha a disposizione una barca che può portare al più due tra essi (oltre all'uomo). Inoltre la capra non può essere lasciata sola né con il lupo né con il cavolo. Determinare un piano che risolva la situazione.

[1] Chi desidera può risolvere l'analogo problema delle cinque mogli gelose.

CAPITOLO 15

Vincoli e loro risoluzione

Daremo una definizione semantica, relazionale, della nozione di vincolo e dei concetti e problemi relativi. Negli esempi invece saranno forniti vincoli usando una notazione sintattica intuitiva. Nella seconda parte del capitolo saranno introdotti i *vincoli globali* e algoritmi di propagazione per gli stessi basati su risultati di ricerca operativa.

1. Vincoli e Problemi vincolati

Un *dominio* $\mathcal{D}$ è un insieme (che può essere anche vuoto o infinito) di valori. In questo contesto, ad ogni variabile $X_i \in \mathcal{V}$ si associa un dominio $\mathcal{D}_i$.

Definizione 15.1 (Vincolo). Data una lista finita di variabili $X_1, \ldots, X_k$ (in breve $\vec{X}$) con rispettivi domini $\mathcal{D}_1, \ldots, \mathcal{D}_k$, un *vincolo* C su $\vec{X}$ è una relazione su $\mathcal{D}_1 \times \cdots \times \mathcal{D}_k$, ovvero

$$C \subseteq \mathcal{D}_1 \times \cdots \times \mathcal{D}_k.$$

Diremo che una k-upla $\langle d_1, \ldots, d_k \rangle \in \mathcal{D}_1 \times \cdots \times \mathcal{D}_k$ *soddisfa* un vincolo C su $X_1, \ldots, X_k$ se $\langle d_1, \ldots, d_k \rangle \in C$.

Definizione 15.2 (CSP). Un problema di soddisfacibilità di vincoli o *Constraint Satisfaction Problem (CSP)* è costituito da una lista finita di variabili $X_1, \ldots, X_k$ con rispettivi domini $\mathcal{D}_1, \ldots, \mathcal{D}_k$ e da un insieme *finito* di vincoli $\mathcal{C}$. Un CSP si denota in breve come

$$\mathcal{P} = \langle \mathcal{C}; \mathcal{D}_\in \rangle$$

dove $\mathcal{D}_\in$ identifica la formula:

$$\bigwedge_{i=1}^{k} X_i \in \mathcal{D}_i.$$

La congiunzione $\mathcal{D}_\in$ viene detta *domain expression*.

Una k-upla $\langle d_1, \ldots, d_k \rangle \in \mathcal{D}_1 \times \cdots \times \mathcal{D}_k$ è una *soluzione* di un CSP $\mathcal{P} = \langle \mathcal{C}, \mathcal{D}_\in \rangle$ se $\langle d_1, \ldots, d_k \rangle$ soddisfa ogni vincolo $C \in \mathcal{C}$. L'insieme delle soluzioni di $\mathcal{P}$ si indica con $Sol(\mathcal{P})$. Se $Sol(\mathcal{P}) \neq \emptyset$ (ovvero $\mathcal{P}$ ammette soluzioni) allora $\mathcal{P}$ si dice *consistente*.

Due CSP $\mathcal{P}_1$ e $\mathcal{P}_2$ su una stessa lista di variabili $\vec{X}$ sono *equivalenti* se $Sol(\mathcal{P}_1) = Sol(\mathcal{P}_2)$ (ovvero se ammettono esattamente le stesse soluzioni). Saremo anche interessati alla sostanziale equivalenza di CSP che differiscono per qualche variabile. Diremo che due CSP $\mathcal{P}_1$ e $\mathcal{P}_2$, ove $vars(\mathcal{P}_1) = \vec{X} \subseteq vars(\mathcal{P}_2)$, sono *equisoddisfacibili* se $Sol(\mathcal{P}_1) = Sol(\mathcal{P}_2)|_{\vec{X}}$. In altre parole, data una soluzione di $\mathcal{P}_1$, esiste una soluzione di $\mathcal{P}_2$ in cui le variabili comuni hanno lo stesso valore, e data una soluzione di $\mathcal{P}_2$, rimuovendo gli assegnamenti per le variabili non presenti in $\mathcal{P}_1$, si ottiene una soluzione per $\mathcal{P}_1$.

Esempio 15.1. Si consideri il seguente CSP, ove vincoli e espressioni di dominio sono descritti da una sintassi intuitiva:

$$\mathcal{P} = \langle X_1 > X_2 + 1; X_1 \, in \, 0..5, X_2 \, in \, 2..7 \rangle$$

$\mathcal{P}$ è consistente in quanto, ad esempio, la coppia $\langle 4, 2 \rangle$ è soluzione di $\mathcal{P}$. Inoltre $\mathcal{P}$ è equivalente al CSP

$$\mathcal{P}' = \langle X_1 \geq X_2 + 2; X_1 \, in \, 4..5, X_2 \, in \, 2..3 \rangle$$

Più in generale, un CSP $\mathcal{P}$ tale che $vars(\mathcal{P}) = \vec{X}$, e un insieme di CSP (o equivalentemente una disgiunzione di CSP) $\{\mathcal{P}_1, \ldots, \mathcal{P}_k\}$, ove per ogni $i = 1, \ldots, k$ $vars(\mathcal{P}_i) \supseteq \vec{X}$, sono *equisoddisfacibili* se:

$$Sol(\mathcal{P}) = \bigcup_{i=1}^{k} Sol(\mathcal{P}_i)|_{\vec{X}}$$

ovvero:

- Ogni soluzione di $\mathcal{P}$ può essere estesa per ottenere una soluzione di (almeno) uno dei $\mathcal{P}_i$ (ovvero esistono dei valori per le eventuali variabili non presenti in $\mathcal{P}$ che permettono di trovare una soluzione per $\mathcal{P}_i$).
- Per ogni $i = 1, \ldots, k$, la restrizione di ogni soluzione di $\mathcal{P}_i$ alle variabili di $\mathcal{P}$ è soluzione di $\mathcal{P}$.

Esempio 15.2. Si consideri il seguente CSP, ove i domini sono di tipo insiemistico:

$$\mathcal{C} = \langle \{X_1\} \cup X_2 = \{0, 1\}; X_1 \, in \, \mathbb{N}, X_2 \, in \, \wp(\mathbb{N}) \rangle$$

$\mathcal{C}$ è equivalente al seguente insieme di CSP:

$$\begin{array}{ll} \{ & \langle X_1 = 0, X_2 = \{1\} \cup X_3; X_1 \, in \, \mathbb{N}, X_2 \, in \, \wp(\mathbb{N}), X_3 \, in \, \wp(\{0, 1\}) \rangle, \\ & \langle X_1 = 1, X_2 = \{0\} \cup X_3; X_1 \, in \, \mathbb{N}, X_2 \, in \, \wp(\mathbb{N}), X_3 \, in \, \wp(\{0, 1\}) \rangle \ \ \} \end{array}$$

Dato un CSP $\mathcal{P}$ siamo interessati a diversi problemi (simili a quelli visti per la E-unificazione):

- $\mathcal{P}$ è consistente?
- se lo è, qual è una sua soluzione?
- esiste un CSP equivalente/equisoddisfacibile ma più semplice (secondo qualche accezione)?
- esiste una rappresentazione finita di tutte le soluzioni?

Spesso un CSP $\mathcal{P} = \langle \mathcal{C}; \mathcal{D}_\in \rangle$ viene associato ad una funzione $f : Sol(\mathcal{P}) \longrightarrow E$ ove E è ordinato da una relazione solitamente denotata con $<$ (tipicamente $E = \mathbb{R}$ o $E = \mathbb{N}$). In tal caso, dato $k \in E$, siamo interessati a cercare le soluzioni $\vec{d}$ di $\mathcal{P}$ tali che $f(\vec{d}) \geq k$ (o $f(\vec{d}) \leq k$). Oppure, più in generale:

Definizione 15.3 (COP). Un problema di ottimizzazione vincolato o *Constrained Optimization Problem* (COP) è un CSP $\mathcal{P}$ con associata una funzione f. Una *soluzione* per $\langle \mathcal{P}, f \rangle$ è una soluzione $\vec{d}$ di $\mathcal{P}$ che massimizza (o minimizza, a seconda delle richieste—in tal caso si sostituisca $\geq$ con $\leq$ nella formula sotto riportata) la funzione f, ovvero tale che

$$(\forall \vec{e} \in Sol(\mathcal{P}))(f(\vec{d}) \geq f(\vec{e}))$$

Esempio 15.3. Si consideri la seguente istanza del problema di knapsack generalizzato:

- una bottiglia di vino (v) occupa spazio 10 e fornisce un profitto 6,
- una bottiglia di grappa (g) occupa spazio 17 e fornisce un profitto 10,
- una confezione di spaghetti (s) occupa spazio 4 e fornisce un profitto 2.

Un onesto ricercatore deve portare nella sua missione in USA, nel suo zaino che dispone di uno spazio totale di 49, almeno un oggetto per tipo, ma ovviamente vuole massimizzare il guadagno che farà rivendendo il contenuto ai più pagati colleghi americani. Questo problema si può formalizzare mediante un COP:

$$\langle 10v + 17g + 4s \leq 49; v\,in\,\mathbb{N} \setminus \{0\}, g\,in\,\mathbb{N} \setminus \{0\}, s\,in\,\mathbb{N} \setminus \{0\}\rangle$$

ove la funzione $f(v, g, s) = 6v + 10g + 2s$.

Il lettore può calcolare (a mano o codificandolo, ad esempio, in CLP(FD)) la soluzione per tale COP, e versare una lacrima per gli scarsi introiti del baldo ricercatore.

2. Risolutori di vincoli

In questa sezione analizzeremo alcune tecniche sviluppate per risolvere CSP e COP. Iniziamo introducendo il seguente concetto:

DEFINIZIONE 15.4 (Solver). Un risolutore di vincoli (*Constraint Solver*, o semplicemente solver, per brevità) è una procedura che trasforma un CSP $\mathcal{P}$ in uno equisoddisfacibile (anche se spesso vale l'equivalenza). Un solver si dice:

completo: se, dato $\mathcal{P}$, lo trasforma in un CSP o in una disgiunzione finita di CSP ad esso equisoddisfacibile, e tale che da ciascuno dei disgiunti sia immediato trarre ogni sua soluzione; se $\mathcal{P}$ è inconsistente, viene restituito `fail`.

incompleto: se non è completo. Intuitivamente, dato $\mathcal{P}$, lo trasforma in un CSP più semplice ma non ancora abbastanza semplice.

Si osservi che un solver completo è in grado si stabilire la consistenza di un CSP, mentre ciò non è detto nel caso di solver incompleto.

ESEMPIO 15.4. Il classico esempio di solver completo è l'algoritmo di unificazione. In questo caso il dominio di tutte le variabili è l'insieme $T(\Sigma)$.

Come vedremo, un solver (volutamente) incompleto è il solver della libreria `clpfd` di SICStus Prolog.

DEFINIZIONE 15.5 (Proof rules). Un constraint solver è basato sull'applicazione ripetuta di *regole di dimostrazione*

$$\frac{\varphi}{\psi}$$

ove φ e ψ sono dei CSP. In ψ possono occorrere anche variabili non presenti in φ. Una regola mantiene l'equivalenza (è *equivalence preserving*) se $Sol(\varphi) = Sol(\psi)|_{vars(\varphi)}$, ovvero φ e ψ sono equisoddisfacibili.

In queste note non tratteremo regole di dimostrazione non-deterministiche

$$\frac{\varphi}{\psi_1 | \cdots | \psi_k}$$

utilizzate per sviluppare solver completi (usate ad esempio per problemi di tipo insiemistico come quelli dell'esempio 15.2). Una regola siffatta può, se la sintassi lo permette essere simulata da un'unica regola deterministica che introduce un constraint disgiuntivo:

$$\frac{\varphi}{\psi_1 \vee \cdots \vee \psi_k}$$

Le regole si suddividono in 3 famiglie:

2.1. Domain Reduction Rules. L'idea è quella di restringere i domini sfruttando le informazioni presenti nei vincoli. Tale riduzione può avere come conseguenza la semplificazione dei vincoli o la rilevazione di situazioni di inconsistenza del CSP. Le regole hanno la forma:

$$\frac{\varphi = \langle \mathcal{C}; \mathcal{D}_\in \rangle}{\psi = \langle \mathcal{C}'; \mathcal{D}'_\in \rangle}$$

dove:

- $\mathcal{D}_\in = X_1 \in \mathcal{D}_1, \ldots, X_k \in \mathcal{D}_k$,
- $\mathcal{D}'_\in = X_1 \in \mathcal{D}'_1, \ldots, X_k \in \mathcal{D}'_k$,
- per ogni $i = 1, \ldots, k$ vale che $\mathcal{D}'_i \subseteq \mathcal{D}_i$, e infine
- $\mathcal{C}'$ è la restrizione di $\mathcal{C}$ ai nuovi domini delle variabili.

A seguito delle riduzioni dei domini, alcuni vincoli in $\mathcal{C}'$ possono diventare ridondanti (implicati dai domini) e dunque rimossi. Se tutti i vincoli si possono rimuovere, $\mathcal{C}'$ è sostituito da `true` (il vincolo sempre soddisfatto). In tal caso nella domain expression $\mathcal{D}'_\in$ sono rappresentate tutte le soluzioni.

Se invece uno dei $\mathcal{D}_i$ risultasse vuoto ($\mathcal{D}_i = \emptyset$) allora ψ diventerebbe semplicemente `fail` (un'abbreviazione per denotare un CSP inconsistente).

Esempio 15.5. Si considerino le seguenti istanze di regole che preservano l'equivalenza:

(1)

$$\frac{\langle Y < X; X\ in\ 0..10, Y\ in\ 5..15 \rangle}{\langle Y < X; X\ in\ 6..10, Y\ in\ 5..9 \rangle}$$

In questo caso, poiché l'equivalenza è preservata,

$$\begin{aligned} \mathcal{C} = \mathcal{C}' = \ & \{(5,6),(5,7),(5,8),(5,9),(5,10),(6,7),(6,8),(6,9),(6,10), \\ & (7,8),(7,9),(7,10),(8,9),(8,10),(9,10)\} \end{aligned}$$

Tuttavia cambiano i domini di cui tali vincoli sono relazioni: $\mathcal{C} \subseteq \{5, \ldots, 15\} \times \{0, \ldots, 10\}$ mentre: $\mathcal{C}' \subseteq \{5, \ldots, 9\} \times \{6, \ldots, 10\}$.

(2)

$$\frac{\langle Y \neq X; X\ in\ 0..10, Y\ in\ 0..0 \rangle}{\langle \mathtt{true}; X\ in\ 1..10, Y\ in\ 0..0 \rangle}$$

Si noti che quando il dominio consta di un unico valore, tale variabile potrà solo assumere quel valore. In questo caso, inoltre, il vincolo è soddisfatto da ogni assegnamento per le variabili e pertanto sostituibile da `true`.

(3)

$$\frac{\langle Y < X; X\ in\ 5..10, Y\ in\ 10..15 \rangle}{\langle Y < X; X\ in\ \emptyset, Y\ in\ \emptyset \rangle}$$

Abbiamo raggiunto dei domini vuoti. Il CSP non ammette soluzioni. Potremmo riscrivere la regola come:

$$\frac{\langle Y < X; X \; in \; 5..10, Y \; in \; 10..15 \rangle}{\texttt{fail}}$$

2.2. Transformation Rules. In questo caso sono i vincoli ad essere trasformati/semplificati. È possibile che la semplificazione introduca nuove variabili a cui va assegnato un dominio non vuoto. Se viene introdotto il vincolo sempre falso $\perp$, allora si raggiunge una situazione generale di `failure`. Le regole hanno la forma:

$$\frac{\varphi = \langle \mathcal{C}; \mathcal{D}_{\in} \rangle}{\psi = \langle \mathcal{C}'; \mathcal{D}'_{\in} \rangle}$$

dove:

- $\mathcal{D}_{\in} = X_1 \in \mathcal{D}_1, \ldots, X_k \in \mathcal{D}_k$,
- $\mathcal{D}'_{\in} = X_1 \in \mathcal{D}_1, \ldots, X_k \in \mathcal{D}_k$, più eventualmente nuovi domini non vuoti per nuove variabili (i domini delle variabili già presenti non si riducono)

Se in $\mathcal{C}'$ è presente $\perp$, allora ψ diventerebbe semplicemente `fail`.

Esempio 15.6. Si considerino le seguenti istanze di regole che preservano l'equivalenza:

(1)
$$\frac{\langle e_1 \neq e_2; \mathcal{D}_{\in} \rangle}{\langle X = e_1, X \neq e_2; \mathcal{D}_{\in}, X \; in \; \mathbb{Z} \rangle}$$
In questo caso viene introdotta una nuova variabile, il cui dominio iniziale è il più grande possibile (assumiamo di lavorare con numeri interi). L'introduzione di una nuova variabile rende necessario ricorrere al concetto di equisoddisfacibilità tra CSP.

(2) Tutte le regole dell'algoritmo di unificazione sono transformation rules. Vediamo, ad esempio, la regola di applicazione della sostituzione:
$$\frac{\langle \mathcal{C}; \mathcal{D}_{\in}, X \; in \; 0..0 \rangle}{\langle \mathcal{C}[X/0]; \mathcal{D}_{\in}, X \; in \; 0..0 \rangle}$$

(3) ed un caso di fallimento:
$$\frac{\langle f(s_1, \ldots, s_m) = g(t_1, \ldots, t_n), \mathcal{C}; \mathcal{D}_{\in} \rangle}{\langle \perp, \mathcal{C}; \mathcal{D}_{\in} \rangle}$$
In questo caso si può sostituire ψ con `fail`.

2.3. Introduction Rules. In alcuni casi risulta necessario o utile introdurre nuovi vincoli (tipicamente implicati dai vincoli già presenti). Le regole hanno la forma:

$$\frac{\varphi = \langle \mathcal{C}; \mathcal{D}_{\in} \rangle}{\psi = \langle \mathcal{C}, C; \mathcal{D}_{\in} \rangle}$$

dove C è un nuovo constraint. In questo caso non si raggiunge mai il CSP fallimentare.

Esempio 15.7. Consideriamo questa regola con domini sui reali.

$$\frac{\langle X^2 - 2XY + Y^2 < 5; X > 0, Y > 0 \rangle}{\langle X^2 - 2XY + Y^2 < 5, X - Y < 3; X > 0, Y > 0 \rangle}$$

Il nuovo vincolo non aggiunge informazione essendo implicato dal primo CSP. Tuttavia introdurre vincoli semplici, se pur ridondanti, può rendere più efficiente il processo di inferenza di un constraint solver.

3. Constraint Propagation

L'applicazione ripetuta delle regole dei tre tipi visti viene detta fase di propagazione di vincoli. Più in dettaglio, in questo contesto, una *derivazione* è una sequenza di applicazioni delle regole di dimostrazione, previa opportuna rinomina delle variabili presenti nelle regole.

Una derivazione finita è:

- di *fallimento* se l'ultimo CSP è `fail`,
- *stabile* se non è di fallimento e si raggiunge un CSP per cui nessuna regola è applicabile,
- di *successo* se è stabile e l'ultimo CSP è in forma risolta (ovvero è consistente e da esso è immediato individuare una soluzione).

La nozione di forma risolta, qui enunciata in modo intuitivo, viene precisata quando i domini e i tipi di operatori ammessi per definire i vincoli sono fissati. Abbiamo già incontrato una nozione di questo tipo studiando l'algoritmo di unificazione, che di fatto è un constraint solver completo.

Cerchiamo ora di essere meno generali rispetto alle regole di dimostrazione. Verificare proprietà *globali* di un CSP è spesso molto difficile. Per rendersene conto, si osservi che è immediato, avendo a disposizione il simbolo $\neq$, mappare problemi di map coloring—dunque NP-completi—in un CSP. Ciò che si cerca di ottenere tramite la applicazione delle regole è di analizzare proprietà locali del CSP che, se non verificate, garantiscono l'inconsistenza globale. Vediamo le più utilizzate tra queste proprietà locali.

3.1. Node consistency. Un CSP $\mathcal{P}$ è *node consistent* se per ogni variabile X in $\mathcal{P}$, ogni vincolo unario relativo a X coincide con il dominio di X.

Esempio 15.8.

$$\langle X_1 \geq 0, X_2 \geq 0; X_1\ in\ \mathbb{N}, X_2\ in\ \mathbb{N}\rangle$$

è node consistent, mentre:

$$\langle X_1 \geq 5, X_2 \geq 1; X_1\ in\ \mathbb{N}, X_2\ in\ \mathbb{N}\rangle$$

non lo è (è comunque consistente). Per finire:

$$\langle X_1 \neq X_2, X_1 = 0, X_2 = 0; X_1\ in\ 0..0, X_2\ in\ 0..0\rangle$$

è node consistent, ma non è consistente.

Consideriamo la regola, di tipo *domain reduction*, in cui evidenziamo solo la parte del CSP che andiamo a sostituire:

$$(NC) \qquad \frac{\langle C; X\ in\ D_X\rangle}{\langle C; X\ in\ D_X \cap C\rangle}$$

dove C è un vincolo unario sulla variabile X.

E' immediato dimostrare la proprietà: *un CSP è node consistent se e solo se è chiuso per la regola (NC).*

3.2. Arc consistency. Un constraint binario C sulle variabili X ed Y aventi rispettivi domini D_X e D_Y è *arc consistent* se:

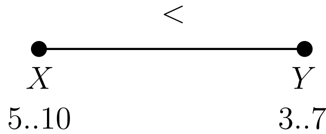
(1) $(\forall a \in D_X)(\exists b \in D_Y)\,((a,b) \in C)$, e
(2) $(\forall b \in D_Y)(\exists a \in D_X)\,((a,b) \in C)$.

Un CSP $\langle \mathcal{C}; \mathcal{D}_\in \rangle$ è arc consistent se lo sono tutti i constraint binari presenti in $\mathcal{C}$.

Le nozioni di arco e nodo vanno fatte risalire al grafo che possiamo costruire mettendo come nodi le variabili di un CSP (etichettati dai loro domini) e come archi i vincoli binari tra queste. Ad esempio, il CSP

$$\langle X < Y; X \; in \; 5..10, Y \; in \; 3..7 \rangle$$

si può rappresentare con il grafo:



Esempio 15.9. Il CSP appena descritto non è arc consistent ma è consistente. Il CSP:

$$\langle X \neq Y, Y \neq Z, X \neq Z; X \; in \; 0..1, Y \; in \; 0..1, Z \; in \; 0..1 \rangle$$

è invece arc consistent, ma non consistente.

Possiamo definire due regole per l'arc consistency (anch'esse di tipo *domain reduction*):

$$(AC1) \qquad \frac{\langle C; X \; in \; D_X, Y \; in \; D_Y \rangle}{\langle C; X \; in \; D'_X, Y \; in \; D_Y \rangle}$$

$$(AC2) \qquad \frac{\langle C; X \; in \; D_X, Y \; in \; D_Y \rangle}{\langle C; X \; in \; D_X, Y \; in \; D'_Y \rangle}$$

dove C è un vincolo binario sulle variabili X ed Y, mentre:

- $D'_X = \{ \, a \in D_X \mid (\exists b \in D_Y)((a,b) \in C) \, \}$
- $D'_Y = \{ \, b \in D_Y \mid (\exists a \in D_X)((a,b) \in C) \, \}$

E' immediato dimostrare la proprietà: un CSP è arc consistent se e solo se è chiuso per le regole (AC1) e (AC2).

Esempio 15.10. L'applicazione di (AC1) al CSP

$$\langle X < Y; X \; in \; 5..10, Y \; in \; 3..7 \rangle$$

lo trasforma nel CSP

$$\langle X < Y; X \; in \; 5..6, Y \; in \; 3..7 \rangle$$

la successiva applicazione di (AC2) ci permette di ottenere:

$$\langle X < Y; X \; in \; 5..6, Y \; in \; 6..7 \rangle$$

Si osservi che invertendo le applicazioni delle due regole si sarebbe giunti allo stesso risultato (questo può essere provato).

3.3. Bounds consistency. Per applicare le regole (AC1) e (AC2) nell'esempio 15.10, è stato sufficiente guardare gli estremi (bounds) dei domini. Ciò è corretto quando i domini sono degli intervalli. Tuttavia, la maggior parte dei constraint solver a nostra disposizione opera in questo modo anche nel caso di domini non intervalli. In tal caso non si garantisce di raggiungere l'arc consistency, ma la più semplice proprietà detta *bounds consistency.* Se i domini sono "grandi" e hanno al loro interno molti "buchi" ciò permette di realizzare una procedura estremamente più efficiente (seppur meno precisa).

Qui diamo la nozione su vincoli binari; la stessa si può dare per vincoli n-ari generici (si veda Sezione 3.6).

Siano min e max due funzioni che restituiscono, rispettivamente, il minimo ed il massimo elemento di un dominio D.

Un constraint binario C sulle variabili X ed Y aventi rispettivi domini D_X e D_Y è *bounds consistent* se:

(1) $(\exists b \in \min(D_Y)..\max(D_Y))\,((\min(D_X), b) \in C)$, e
$(\exists b \in \min(D_Y)..\max(D_Y))\,((\max(D_X), b) \in C)$,
(2) $(\exists a \in \min(D_X)..\max(D_X))\,((a, \min(D_Y)) \in C)$, e
$(\exists a \in \min(D_X)..\max(D_X))\,((a, \max(D_Y)) \in C)$,

Un CSP $\langle \mathcal{C}; \mathcal{D}_\in \rangle$ è bounds consistent se lo sono tutti i constraint binari presenti in $\mathcal{C}$.

Esempio 15.11. Il seguente CSP è consistent, bounds consistent, ma non arc consistent:

$$\langle 2X = Y; X\ in\ \{0, 1, 2\}, Y\ in\ \{0, 4\} \rangle$$

Applicando (AC1) si otterrebbe infatti:

$$\langle 2X = Y; X\ in\ \{0, 2\}, Y\ in\ \{0, 4\} \rangle$$

Le due regole per la bounds consistency sono (anch'esse di tipo *domain reduction*):

$$(BC1) \qquad \frac{\langle C; X\ in\ D_X, Y\ in\ D_Y \rangle}{\langle C; X\ in\ D'_X, Y\ in\ D_Y \rangle}$$

$$(BC2) \qquad \frac{\langle C; X\ in\ D_X, Y\ in\ D_Y \rangle}{\langle C; X\ in\ D_X, Y\ in\ D'_Y \rangle}$$

dove C è un vincolo binario sulle variabili X ed Y, mentre:

- $D'_X = D_X \cap (\min(\{\, a \in D_X \mid (\exists b \in \min(D_Y)..\max(D_Y))((a, b) \in C) \,\})..$
$\max(\{\, a \in D_X \mid (\exists b \in \min(D_Y)..\max(D_Y))((a, b) \in C) \,\}))$
- $D'_Y = D_Y \cap (\min(\{\, b \in D_Y \mid (\exists a \in \min(D_X)..\max(D_X))((a, b) \in C) \,\})..$
$\max(\{\, b \in D_Y \mid (\exists a \in \min(D_X)..\max(D_X))((a, b) \in C) \,\}))$

L'applicazione delle regole (BC1) e (BC2) può, a prima vista, sembrare complessa. In realtà si tratta semplicamente di operare sui 4 bounds dei domini e pertanto, nella gran parte dei vincoli che vengono considerati, si riescono ad applicare in tempo pressocché costante. Al solito, vale la proprieà: *un CSP è bounds consistent se e solo se è chiuso per l'applicazione delle regole (BC1) e (BC2).*

3.4. Directional arc consistency. Questa condizione è un altro indebolimento dell'arc consistency. La nozione è la stessa, ma si assume un ordine sulle variabili. La propagazione è di fatto monodirezionale.

Un constraint binario C sulle variabili X ed Y, con $X \lhd Y$, aventi rispettivi domini D_X e D_Y è *directional arc consistent* rispetto a $\lhd$ se:

(1) $(\forall a \in D_X)(\exists b \in D_Y)\,((a, b) \in C)$, e

Un CSP $\langle \mathcal{C}; \mathcal{D}_\in \rangle$ è directional arc consistent se lo sono tutti i constraint binari presenti in $\mathcal{C}$.

Si userà dunque la regola (AC1) se $X \lhd Y$, la (AC2) in caso contrario. Pertanto le variabili più "grandi" guidano la riduzione dei domini di quelle più "piccole".

Algoritmi basati su directional arc consistency raggiungono il punto fisso molto prima di quelli per l'arc consistency. Oviamente il CSP che viene raggiunto sarà meno semplice rispetto a quello raggiungibile usando arc consistency.

Si può ovviamente applicare la nozione di direzionalità anche alla bounds consistency.

3.5. Hyper arc consistency. Si tratta di una generalizzazione dell'arc consistency a vincoli non binari. Un vincolo n-ario C sulle variabili $X_1, \ldots, X_n$ è *hyper arc consistent* se per ogni $i = 1, \ldots, n$ vale che:

- $(\forall a_i \in D_i)(\exists a_1 \in D_1) \cdots (\exists a_{i-1} \in D_{i-1})(\exists a_{i+1} \in D_{i+1}) \cdots (\exists a_n \in D_n)(\langle a_1, \ldots a_n \rangle \in C)$

Un CSP è *hyper arc consistent* se lo sono tutti i suoi vincoli.

ESEMPIO 15.12. Si consideri il seguente CSP:

$$\langle 2X + 3Y < Z; X \; in \; 1..10, Y \; in \; 1..10, Z \; in \; 1..10 \rangle$$

Notiamo che qualunque coppia di valori per X ed Y garantisce almeno la somma 5. Dunque, essendoci 1 in D_Z il CSP non è hyper arc consistent. Anche il 3 in D_Y è un valore da togliere. Infatti, con tale valore l'espressione a sinistra arriva a 9, a cui va sommato almeno 2 come contributo di X. Ció supera il valore massimo a sinistra (9).

Il CSP equivalente e hyper arc consistent è:

$$\langle X + Y < Z; X \; in \; 1..3, Y \; in \; 1..2, Z \; in \; 6..10 \rangle$$

Per ottenere questa proprietà vanno applicate le regole $(HACi)$ di seguito descritte, ove $i = 1, \ldots, n$:

$$(HACi) \qquad \frac{\langle C; X_1 \; in \; D_1, \ldots, X_{i-1} \; in \; D_{i-1}, X_i \; in \; D_i, X_{i+1} \; in \; D_{i+1}, \ldots, X_n \; in \; D_n \rangle}{\langle C; X_1 \; in \; D_1, \ldots, X_{i-1} \; in \; D_{i-1}, X_i \; in \; D'_i, X_{i+1} \; in \; D_{i+1}, \ldots, X_n \; in \; D_n \rangle}$$

ove C è un vincolo su $a_1, \ldots, a_n$ e

$$D'_i = \{\, a_i \in D_i \mid (\exists a_1 \in D_1) \cdots (\exists a_{i-1} \in D_{i-1})(\exists a_{i+1} \in D_{i+1}) \cdots (\exists a_n \in D_n)(\langle a_1, \ldots a_n \rangle \in C) \,\}$$

Per ogni vincolo n-ario vi saranno dunque n di queste regole proiettive.

3.6. Hyper Bounds consistency. La tecnica di bounds consistency che permette di calcolare, in modo approssimato ma efficiente l'arc consistency, può essere applicata anche a vincoli di grado n-generico. Diremo che un vincolo n-ario C sulle variabili $X_1, \ldots, X_n$ è *hyper bounds consistent* se per ogni $i = 1, \ldots, n$ vale che:

- $(\exists a_1 \in \min(D_1)..\max(D_1)) \cdots (\exists a_{i-1} \in \min(D_{i-1})..\max(D_{i-1}))$
  $(\exists a_{i+1} \in \min(D_{i+1})..\max(D_{i+1})) \cdots (\exists a_n \in \min(D_n)..\max(D_n))$
  $(\langle a_1, \ldots, a_{i-1}, \min(D_i), a_{i+1}, \ldots, a_n \rangle \in C)$, e
- $(\exists a_1 \in \min(D_1)..\max(D_1)) \cdots (\exists a_{i-1} \in \min(D_{i-1})..\max(D_{i-1}))$
  $(\exists a_{i+1} \in \min(D_{i+1})..\max(D_{i+1})) \cdots (\exists a_n \in \min(D_n)..\max(D_n))$
  $(\langle a_1, \ldots, a_{i-1}, \max(D_i), a_{i+1}, \ldots, a_n \rangle \in C)$

Esempio 15.13. Si consideri il CSP

$$\langle X + Y < Z; X \ in\ 1..100, Y \ in\ 10..100, Z \ in\ 1..50 \rangle$$

Non è hyper bounds consistent. Ad esempio si osservi che non esistono $a \in 1..100$ e $b \in 10..100$ tali che $a + b < \min(D_Z) = 1$. Il CSP equivalente ma hyper bounds consistent sarà:

$$\langle X + Y < Z; X \ in\ 1..39, Y \ in\ 10..48, Z \ in\ 12..50 \rangle$$

Come esercizio, si trovi un CSP hyper bounds consistent, ma non hyper arc consistent.

Per raggiungere la hyper bounds consistency si tratta, al solito, di applicare delle regole che necessitano, sperabilmente, solo dell'analisi dei bounds degli intervalli. Dal punto di vista formale, le regole da applicare sono una immediata generalizzazione delle regole (BC1) e (BC2) descritte nella Sezione 3.3.

3.7. Path consistency. Il fatto che un algoritmo di propagazione di vincoli non riesca ad accorgersi della non consistenza del semplice CSP dell'esempio 15.9 può essere una situazione non accettabile. Inoltre, usando arc consistency, servono 3 milioni di passi per accorgersi dell'inconsistenza di:

$$\langle X < Y, Y < Z, Z < X; X \ in\ 1..1000000, Y \ in\ 1..1000000, Z \ in\ 1..1000000 \rangle$$

La nozione di path consistency è stata introdotta per ovviare a questi problemi.

Un CSP $\mathcal{P}$ è *normalizzato* se per ogni coppia non ordinata di variabili X, Y esiste al più un vincolo binario su X, Y in $\mathcal{P}$. Se $\mathcal{P}$ è *normalizzato* indichiamo con $C(X, Y)$ tale vincolo se esiste. Se non c'è, possiamo definire $C(X, Y) = D_X \times D_Y$. Un CSP $\mathcal{P}$ è *standard* se per ogni coppia non ordinata di variabili X, Y esiste esattamente un vincolo binario su X, Y $(C(X, Y))$ in $\mathcal{P}$.

Esempio 15.14.

$$\langle \underbrace{X + Y < 5}_{C(X,Y)}, \underbrace{X + Y \leq Z}_{C(X,Y,Z)}, \underbrace{X < Z}_{C(X,Z)}, \underbrace{2X \neq 3Z}_{C(X,Z)}, \underbrace{X + Y \neq Z}_{C(X,Y,Z)}; \mathcal{D}_{\in} \rangle$$

Non è normalizzato in quanto ci sono due vincoli per la coppia X, Z. Possiamo (si tratta di un concetto semantico, ma di solito anche la sintassi lo permette) rimpiazzare due vincoli C_1 e C_2 con l'unico vincolo $C_1 \wedge C_2$.

COn questa modifica sarebbe normalizzato, ma non standard, in quanto mancherebbe il vincolo per la coppia Y, Z. Possiamo introdurre il vincolo $D_Y \times D_Z$. Dal punto di vista sintattico, possiamo pensare ad un predicato binario `all` che non pone vincoli tra le due variabili passate come argomento.

Si ottiene dunque:

$$\langle \underbrace{X + Y < 5}_{C(X,Y)}, \underbrace{(X < Z \wedge 2X \neq 3Z)}_{C(X,Z)}, \underbrace{\texttt{all}(Y,Z)}_{C(Y,Z)}, \underbrace{X + Y < Z}_{C(X,Y,Z)}; \mathcal{D}_\in \rangle$$

L'esistenza o l'assenza di vincoli ternari non influenza la nozione di standard data in queste note (in altri testi potreste trovare definizioni diverse).

Si può dimostrare che per ogni CSP ne esiste uno standard equivalente. In questa fase di normalizzazione, si può perdere l'arc consistency (e pure accorgersi dell'inconsistenza).

Esempio 15.15.

$$\langle X + Y = 0, X - Y = 0; X\ in\ \{-1, 1\}, Y\ in\ \{-1, 1\} \rangle$$

è arc consistent, ma non standard.

$$\langle (X + Y = 0 \wedge X - Y = 0); X\ in\ \{-1, 1\}, Y\ in\ \{-1, 1\} \rangle$$

è standard ma non arc consistent.

Data una relazione R binaria definiamo la sua trasposta

$$R^T = \{ (b, a) \mid (a, b) \in R \}$$

Data due relazioni binarie R e S, definiamo la loro composizione

$$RS = \{ (a, b) \mid \exists c\, ((a, c) \in R, (c, b) \in S) \}$$

Definizione 15.6 (Path consistency). Un CSP standard è *path consistent* se per ogni tripla di variabili X, Y, Z vale che

$$C(X, Z) \subseteq C(X, Y)C(Y, Z)$$

ovvero, se $(a, b) \in C(X, Z)$ esiste $c \in D_Y$ tale che $(a, c) \in C(X, Y)$ e $(c, b) \in C(Y, Z)$.

Si osservi che, per la generalità delle variabili, nella definizione è implicitamente richiesto anche:

$$C(X, Y) \subseteq C(X, Z)C(Z, Y) \text{ e } C(Y, Z) \subseteq C(Y, X)C(X, Z)$$

Esempio 15.16. Si consideri il solito CSP arc consistent, ma non consistente:

$$\langle X \neq Y, Y \neq Z, X \neq Z; X\ in\ 0..1, Y\ in\ 0..1, Z\ in\ 0..1 \rangle$$

Abbiamo che $C(X, Z) = C(X, Y) = C(Y, Z) = \{(0, 1), (1, 0)\}$. Si osservi che $C(X, Y)C(Y, Z) = \{(0, 0), (1, 1)\}$ e dunque

$$C(X, Z) = \{(0, 1), (1, 0)\} \not\subseteq \{(0, 0), (1, 1)\} = C(X, Y)C(Y, Z)$$

ovvero non è path consistent.

Per ottenere la path consistency a partire da un CSP standard vanno applicate le seguenti regole:

$$(PC1) \qquad \frac{\langle C(X,Y), C(X,Z), C(Y,Z); \mathcal{D}_{\in} \rangle}{\langle C'(X,Y), C(X,Z), C(Y,Z); \mathcal{D}_{\in} \rangle}$$

$$(PC2) \qquad \frac{\langle C(X,Y), C(X,Z), C(Y,Z); \mathcal{D}_{\in} \rangle}{\langle C(X,Y), C'(X,Z), C(Y,Z); \mathcal{D}_{\in} \rangle}$$

$$(PC3) \qquad \frac{\langle C(X,Y), C(X,Z), C(Y,Z); \mathcal{D}_{\in} \rangle}{\langle C'(X,Y), C(X,Z), C'(Y,Z); \mathcal{D}_{\in} \rangle}$$

ove:

- $C'(X,Y) = C(X,Y) \cap (C(X,Z)C(Y,Z)^T)$
- $C'(X,Z) = C(X,Z) \cap (C(X,Y)C(Y,Z))$
- $C'(Y,Z) = C(Y,Z) \cap (C(X,Y)^T C(X,Z))$

Al solito, si può dimostrare che un CSP in forma standard è path consistent se e solo se è chiuso per l'applicazione delle tre regole (PC1), (PC2) e (PC3). Anche in questo caso si può fornire una versione più debole basata su un ordine delle variabili, ovvero la directional path consistency.

3.8. k-consistency. L'ultima nozione di consistenza locale che riportiamo in queste note è la nozione di k-consistency. Tale nozione generalizza le principali nozioni viste finora.

Sia dato un CSP $\mathcal{P}$ su un certo insieme di variabili $\mathcal{V}$. Un *assegnamento* I è una funzione di alcune (o tutte) variabili $X_1, \ldots, X_k$ di $\mathcal{V}$ su elementi dei rispettivi domini. In breve $I = \{(X_1, d_1), \ldots, (X_k, d_k)\}$. Diremo che il dominio di I, $\text{dom}(I) = \{X_1, \ldots, X_k\}$.

Sia dato un vincolo C. Se $vars(C) \subseteq \{X_1, \ldots, X_k\}$ diciamo che I soddisfa C se la restrizione di I alle variabili di C è una soluzione di C.

Relativamente ad un CSP $\mathcal{P}$, un *assegnamento* I con dominio $\{X_1, \ldots, X_k\}$, ovvero $|\text{dom}(I) = k|$, è k-consistente se soddisfa tutti i vincoli di $\mathcal{P}$ definiti su sottoinsiemi di $\{X_1, \ldots, X_k\}$.

Esempio 15.17. Si consideri il CSP:

$$\langle X \neq Y, Y < Z; X \; in \; 0..1, Y \in in 0..1, \in Z \; in \; 0..1 \rangle$$

L'assegnamento $I = \{(X/0), (Y/1)\}$ (ovvero, usando la notazione delle sostituzioni, $X/0, Y/1$) è 2-consistente. Infatti l'unico vincolo da considerarsi è il primo. Si osservi che non esiste nessuna estensione di I che sia soluzione del CLP (che però è consistente: verificare).

L'esempio appena visto sottolinea che la k-consistenza risulta essere un'altra nozione di consistenza locale. Ovviamente, se $dom(I) \supseteq vars(\mathcal{P})$ e I è k-consistente, allora I è una soluzione di $\mathcal{P}$.

Cerchiamo di portare questa nozione ai CSP.

Definizione 15.7 (k-consistency). Un CSP $\mathcal{P}$ è:

- 1-consistente se è node consistent.
- k-consistente ($k > 1$) se per ogni assegnamento I che sia $k-1$ consistente, e per ogni variabile $X \notin \text{dom}(I)$, esiste un valore in D_X tale che l'assegnamento risultante è k-consistente.

Si osservi che se $k > 1$ e non esiste nessun assegnamento $k-1$ consistente, allora $\mathcal{P}$ è banalmente k-consistente.

Ragioniamo un po' sulla nozione:

- Per definizione, si ha che $\mathcal{P}$ è 1-consistente sse è node consistent.
- Supponiamo $\mathcal{P}$ sia 2-consistente. Allora per ogni vincolo binario $C(X,Y)$ e per ogni assegnamento di una sola delle due variabili, poniamo Y, esiste un valore nel dominio di X che rende vero C. Ma ciò equivale a dire che è arc-consistent.
- Supponiamo $\mathcal{P}$ sia 3-consistente. Consideriamo i vincoli $C(X,Z), C(X,Y), C(Y,Z)$. Per mostrare che $\mathcal{P}$ è path-consistent dobbiamo mostrare che $C(X,Z) \subseteq C(X,Y)C(Y,Z)$. Prendiamo un assegnamento I che sia 2-consistente per $\mathcal{P}$ e supponiamo che I contenga $X/a, Z/b$. Essendo 3-consistente, abbiamo che esiste un valore per Y che soddisfi $C(a,Y)$ e $C(Y,b)$.

  Si completi questo punto mostrando che path-consistency implica 3-consistenza.

La k-consistenza si determina/verifica utilizzando operazioni tipiche dell'algebra relazionale. Per approfondimenti, si veda [**Apt03**].

3.9. Procedure per la propagazione. Una volta scelto il livello di consistenza locale da garantire, si tratta di mettere assieme le regole di derivazione in un unico algoritmo.

Solitamente viene prima effettuata una fase per raggiungere la *node consistency*. Si tratta di una procedura di punto fisso che, sostanzialmente, seleziona una alla volta le variabili di $\mathcal{P}$ e per ogni variabile va a cercare i vincoli unari che la coinvolgono. Per ogni variabile e vincolo unario si applica la regola (NC).

Poi c'è la procedura per l'arc consistency. Si può pensare di applicare la banale procedura di punto fisso descritta in via informale: *si selezioni, uno ad uno ogni constraint binario, vi si applichino le due regole (AC1) e (AC2), si continui finché non si fa una passata intera sui vincoli binari senza aggiornare alcun dominio.* Tale procedura viene detta *AC1* (il lettore scuserà l'abuso di notazione).

Tuttavia una sua naturale ottimizzazione permette computazioni mediamente più efficienti. Tale procedura è detta *AC3* [**Mac77**]. Si tratta di una procedura che apparentemente effettua la directional arc consistency. Per garantirne la completezza rispetto alla arc consistency (generale), all'inizio per ogni vincolo viene inserito il suo trasposto (ad esempio, dal punto di vista sintattico, se c'è il vincolo $X < Y$ viene aggiunto $Y > X$). Sia *ordvar* una funzione che restituisce la lista delle variabili presenti in un vincolo nel loro ordine di apparizione da sinistra verso destra ($ordvar(X < Y) = [X,Y]$, $ordvar(Y > X) = [Y,X]$). L'algoritmo è il seguente:

$S_0 = \{\, C, C^T \mid C \text{ vincolo binario in } \mathcal{C} \,\}$;
$S = S_0$;
`while` $S \neq \emptyset$ `do`
    scegli e togli C da S;
    Sia $[X_i, X_j] = ordvar(C)$;
    $D_i = \{\, a \in D_i \mid \exists b \in D_j\, (a,b) \in C \,\}$;
    `if` D_i è modificato dall'istruzione precedente
        $S = S \cup \{\, C' \in S_0 \mid (\exists Y \neq X_j)(ordvar(C) = [Y, X_i]) \,\}$;

Tale algoritmo evita di considerare vincoli le cui variabili non siano state oggetto di modifiche del dominio.

Il lettore è invitato ad approfondire le procedure basate sulle altre famiglie di regole. In linea di principio, si parte dalla procedura di punto fisso più generale e la si ottimizza evitando di riprocessare vincoli le cui variabili sicuramente non hanno subito riduzione di domini.

Esercizio 15.1. Si studino le complessità delle procedure per ottenere node consistency e arc consistency.

4. Alberi di ricerca

Le procedure di propagazione di vincoli (basate su verifica di proprietà locali) vengono utilizzate assieme a delle regole di *splitting* (domain splitting, constraint splitting) per visitare lo spazio di ricerca delle soluzioni, al fine di individuare soluzioni a un CSP o a un COP.

Per iniziare definiamo la nozione di regola di splitting. Assumiamo che i domini delle variabili siano finiti. Le regole sono regole di riscrittura non deterministica in cui i domini delle variabili vengono ridotti, fino a diventare di un solo elemento o addirittura vuoti. Ciò avviene sia restringendo esplicitamente i domini che rendendo i vincoli più stretti.

Le seguenti sono tipiche regole di splitting di dominio:

(1) (domain) labeling:
$$\frac{X \in \{a_1, \dots, a_k\}}{X \in \{a_1\} | \cdots | X \in \{a_k\}}$$
(2) (domain) enumeration:
$$\frac{X \in \mathcal{D}}{X \in \{a\} | X \in \mathcal{D} \setminus \{a\}}$$
ove $a \in \mathcal{D}$
(3) (domain) bisection:
$$\frac{X \in \mathcal{D}}{X \in \min(\mathcal{D})..a | X \in b..\mathcal{D}}$$
ove $a, b \in \mathcal{D}$, e b è l'elemento immediatamente più grande di a in $\mathcal{D}$. Se $\mathcal{D}$ è un intervallo $x..y$ si prenderanno $a = \lfloor (x+y)/2 \rfloor$ e $b = a + 1$.

Tra queste tre, l'ultima si può applicare anche a domini infiniti, quali ad esempio intervalli di reali. In questo caso si procede su intervalli sempre più piccoli. Si tratterà poi di far terminare la procedura dando un valore ε di precisione: intervalli di lunghezza minore a ε non vanno più suddivisi.

Le regole di splitting di un vincolo sono meno generalizzabili, in quanto strettamente dipendenti dalla sintassi. Alcuni esempi sono i seguenti:

(1) implicazione:
$$\frac{(C_1 \rightarrow C_2)}{\neg C_1 | C_2}$$

(2) valore assoluto:

$$\frac{|e| = X}{X = e | X = -e}$$

(3) diseguaglianza:

$$\frac{e_1 \neq e_2}{e_1 < e_2 | e_2 < e_1}$$

Come dicevamo, la ricerca di soluzioni è un amalgama di propagazione e regole di splitting. La seguente definizione cerca di inquadrare il contesto globale. *prop* sta ovviamente per propagation. Daremo la nozione nel caso la regola di splitting usata sia sempre il *labeling*. Non è difficile modificare la definizione per le altre regole.

DEFINIZIONE 15.8 (Prop-labeling-tree). Sia $\mathcal{P} = \langle C; X_1 \in \mathcal{D}_1, \dots, X_n \in \mathcal{D}_n \rangle$ un CSP. Un *prop-labeling-tree* per $\mathcal{P}$ è un albero tale che:

(1) I nodi sono etichettati da sequenze di espressioni di dominio

$$X_1 \in E_1, \dots, X_n \in E_n$$

(2) In particolare, la radice è etichettata da

$$X_1 \in \mathcal{D}_1, \dots, X_n \in \mathcal{D}_n$$

(3) ogni nodo di livello (pari) $2i$ ove $i = 1, \dots, n$ è etichettato con una etichetta della forma:

$$X_1 \in \{d_1\}, \dots, X_i \in \{d_i\}, X_{i+1} \in E_{i+1}, \dots, X_n \in E_n$$

Se $i = n$ allora il nodo è una foglia. Altrimenti ha *esattamente* un discendente diretto etichettato da:

$$X_1 \in \{d_1\}, \dots, X_i \in \{d_i\}, X_{i+1} \in E'_{i+1}, \dots, X_n \in E'_n$$

ove $E'_j \subseteq E_j$ sono state ottenute per propagazione delle informazioni contenute nel nodo $2i$.

(4) ogni nodo di livello (dispari) $2i + 1$ ove $i = 1, \dots, n$ è etichettato con una etichetta della forma:

$$X_1 \in \{d_1\}, \dots, X_i \in \{d_i\}, X_{i+1} \in \{d_{i+1}\}, X_{i+2} \in E'_{i+2}, \dots, X_n \in E'_n$$

per ogni $d_{i+1} \in E'_{i+1}$ (ove E'_{i+1} è il dominio che appariva nel nodo genitore di livello $2i$), tale che l'assegnamento

$$X_1/d_1, \dots, X_i/d_i, X_{i+1}/d_{i+1}$$

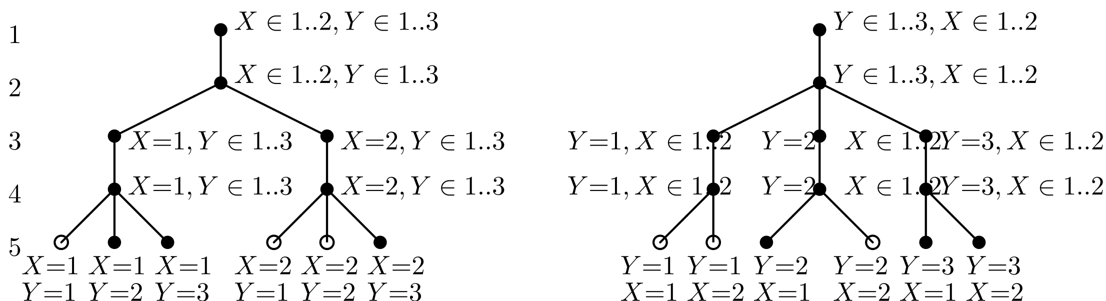
è consistente con (non invalida alcun vincolo di) $\mathcal{C}$. Questo insieme potrebbe essere vuoto.

Ogni foglia di livello $2n$ è di successo. Tutte le altre foglie dell'albero sono di fallimento.

Si osservi che il numero di nodi di successo non dipende dall'ordine. Tuttavia vi sono ordini di variabili che permettono di costruire grafi più piccoli.

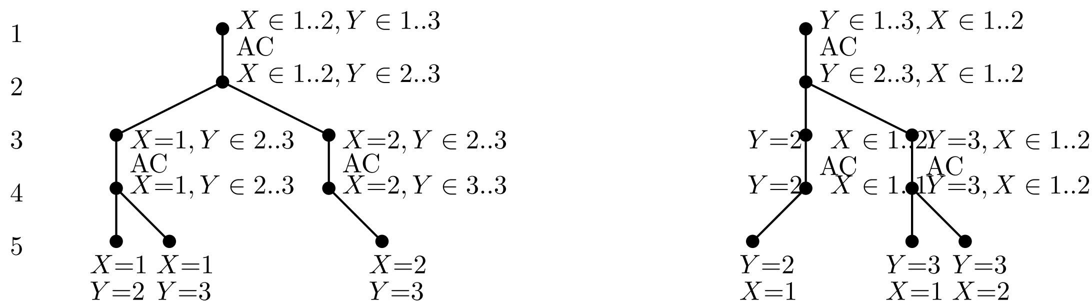
Vediamo come cambiano i due alberi in seguito all'applicazione della propagazione (arc consistency).

Alcune osservazioni sulle dimensioni di un albero siffatto. Conviene, per iniziare, ragionare in assenza di propagazione (ovvero assumiamo che la propagazione non faccia nulla). Si consideri $\mathcal{P} = \langle X < Y; X \in \{1, 2\}, Y \in \{1, 2, 3\} \rangle$. Potremmo iniziare dalla variabile



3 nodi di successo. Globalmente: 12 nodi. 3 nodi di successo. Globalmente: 14 nodi.

FIGURA 15.1. labeling-trees per $\mathcal{P} = \langle X \leq Y; X \in \{1,2\}, Y \in \{1,2,3\}\rangle$. Cerchi vuoti: nodi da non aggiungere (ma comunque calcolati prima di essere scartati)



3 nodi di successo. Globalmente: 9 nodi. 3 nodi di successo. Globalmente: 9 nodi.

FIGURA 15.2. prop-labeling-trees per $\mathcal{P} = \langle X < Y; X \in \{1,2\}, Y \in \{1,2,3\}\rangle$.

X o dalla variabile Y, ottenendo i due alberi della Figura 15.1. Si osservi che l'albero in cui siamo partiti dalla variabile con *dominio più piccolo* necessita di meno nodi. In Figura 15.2 sono illustrati i labeling trees con propagazione (arc consistency). L'esempio è molto piccolo, tuttavia si riesce a scorgere il taglio (che può diventare importante) del numero di nodi inutili dell'albero che una opportuna propagazione può garantire.

4.1. Tecniche di Propagazione (dall'IA). In questa sezione elencheremo alcune tecniche di propagazione da usarsi in un algoritmo di ricerca di soluzioni, così famose da avere un nome proprio.

Per capire le differenze ci concentriamo su un problema ben noto, quello delle N regine. Sia dato un CSP definito come segue:

- la domain expression è: $X_1 \in 1..N, \ldots, X_N \in 1..N$
- i vincoli sono i vincoli di non attacco: for $i, j = 1, \ldots, N$, $i \neq j$:
  - $X_i \neq X_j$
  - $X_i - X_j \neq i - j$
  - $X_i - X_j \neq j - i$.

Fissiamo le idee con $N = 5$. Assegnando (con labeling) $X_1 = 1$, sappiamo che le seguenti caselle sono attaccate (o occupate):

5	*				*
4	*			*	
3	*		*		
2	*	*			
1	∘	*	*	*	*

Pertanto dalla propagazione vorremmo ottenere:

$$X_1 \in 1..1, X_2 \in 3..5, X_3 \in \{2,4,5\}, X_4 \in \{2,3,5\}, X_5 \in 2..4$$

In pratica, in seguito all'assegnamento $X_1 = 1$ i vincoli binari che coinvolgono X_1 diventano dei vincoli unari. Ad esempio:

$$X_1 \neq X_2, X_1 - X_2 \neq -1$$

diventano

$$1 \neq X_2, 1 - X_2 \neq -1$$

ovvero l'unico vincolo unario:

$$X_2 \neq 1 \wedge X_2 \neq 2 \equiv X_2 > 2$$

Ciò che si applica è semplicemente una node consistency.

Più in generale, se i valori per $1, \ldots, i$ sono fissati, ovvero per ogni dominio E_j con $j > i$ vogliamo che

$$E'_j = \{\, e \in E_j \mid (X_1, d_1), \ldots, (X_i, d_i), (X_j, e) \text{ è consistente con } \mathcal{C} \,\}$$

Questa propagazione viene detta *forward checking*, ed è, in pratica, una node consistency.

Una tecnica di propagazione più raffinata è la cosiddetta *partial look ahead*. Un valore per una variabile va lasciato nel dominio se, in seguito all'assunzione di quel valore esiste ancora almeno un valore in ogni variabile *seguente*.

Nell'esempio in oggetto, assumendo l'ordine delle variabili indotto dal loro indice, $X_4 = 3$ non sarebbe un valore possible:

5	*				*
4	*			*	*
3	*		*	◦	*
2	*	*			*
1	◦	*	*	*	*

Dunque i domini diventerebbero:

$$X_1 \in 1..1, X_2 \in 3..5, X_3 \in \{2,4,5\}, X_4 \in \{2,5\}, X_5 \in 2..4$$

Cercando di formalizzare, se i valori per $1, \ldots, i$ sono fissati, per ogni dominio E_j con $j > i$ vogliamo che

$$E'_j = \{\, e \in E_j \mid \begin{array}{l} \text{fissati } X_1 = d_1, \ldots, X_i = d_i, X_j = e, \text{ per ogni } k > j \\ \text{esiste un valore } d_k \text{ che non rende inconsistente } \mathcal{C} \end{array} \,\}$$

Si noti che l'inconsistenza riguarda tutto $\mathcal{C}$. Consideriamo nell'esempio sopra $X_4 = 3$ e guardiamo X_5 il cui dominio è $\{2,3,4\}$ Ognuno dei tre vincoli binari

$$X_4 \neq X_5, X_4 - X_5 \neq 1, X_4 - X_5 \neq -1$$

risulterebbe consistente. Ma la congiunzione

$$X_4 \neq X_5 \wedge X_4 - X_5 \neq 1 \wedge X_4 - X_5 \neq -1$$

invece no. Consideriamo la versione normalizzata del CSP (dunque per ogni coppia di variabili c'è un solo vincolo binario). Considero ogni vincolo C in $\mathcal{C}$:

- Se X_j è l'unica variabile, siamo di fronte ad un vincolo unario.
- Se X_j e X_k sono le due variabili di C e $k < j$ non facciamo nulla.
- Se X_j e X_k sono le due variabili di C e $k > j$ facciamo una arc consistency.

Se vi sono X_j e altre due (o più) variabili, non riusciamo ad accorgersi dell'inconsistenza. Dunque quello che si fa è *node consistency* e la *directional arc consistency.*

Si può andare oltre applicando il *full look ahead.* Un valore per una variabile va lasciato nel dominio se, in seguito all'assunzione di quel valore esiste ancora almeno un valore in ogni *altra* variabile.

5	*	*			*
4	*	*	◦	*	*
3	*	*	*	◦	*
2	*	*			*
1	◦	*	*	*	*

Dunque i domini diventerebbero:

$$X_1 \in 1..1, X_2 \in 3..4, X_3 \in \{2,5\}, X_4 \in \{2,5\}, X_5 \in 2..4$$

Cercando di formalizzare, se i valori per $1, \ldots, i$ sono fissati, per ogni dominio E_j con $j > i$ vogliamo che

$$E'_j = \{\, e \in E_j \mid \begin{array}{l} \text{fissati } X_1 = d_1, \ldots, X_i = d_i, X_j = e, \text{ per ogni } k \neq j \\ \text{esiste un valore } d_k \text{ che non rende inconsistente } \mathcal{C} \end{array} \,\}$$

Ripetendo il ragionamento fatto per il caso precedente, e dunque partendo da un CSP normalizzato, quello che fa il look ahead è la *node consistency* e l'*arc consistency.*

4.2. Ricerca di soluzioni per COP. Nel caso di COP, l'obiettivo è quello di cercare la foglia (o le foglie) di successo dell'albero di ricerca che minimizzano o massimizzano una funzione data. Fissiamoci sul problema di massimizzare (per minimizzare si può ripetere un ragionamento analogo).

Dato un problema di ottimizzazione vincolato $\mathcal{P}$ con associata una funzione f, vogliamo trovare una soluzione $\vec{d}$ di $\mathcal{P}$ tale che

$$(\forall \vec{e} \in Sol(\mathcal{P}))(f(\vec{d}) \geq f(\vec{e}))$$

Un modo banale per risolvere ciò è quello di calcolare tutte le foglie, e selezionare, scandendole quella (quelle) che garantisce il massimo di f. L'algoritmo enumerativo può di fatto essere migliorato sia con metodi *esatti* (quali il branch and bound) che con metodi approssimati. Per alcuni metodi approssimati si suggerisce la lettura del manuale di ECLiPSe.

Vediamo ora come funziona il branch and bound su un esempio.[1] Si consideri la seguente istanza di Knapsack: $\mathcal{P} = \langle 17G+10W+4C < 50; G in 1..5, W in 1..5, C in 1..5\rangle$ ove la funzione da massimizzare è $f(W,G,C) = 10G + 6W + 2C$. Teniamo due variabili ausiliarie, *Bound* (nel diagramma indicato con k) e Max. Nella fase di propagazione, effettuata con hyper-arc consistency, si aggiorna il *Bound*. Usando i valori estremi delle variabili e f, si vede quanto sia il limite massimo che si può raggiungere. Di sicuro nessuna foglia del sottoalbero non può ottenere risultati migliori.

Quando si arriva alla prima soluzione si inizializza il valore di Max (in questo caso 28). Si continua dunque una visita dell'albero, ma ogni nodo (foglia o interno) in cui $Bound < Max$ viene etichettato come fallimentare. Ad ogni nuova soluzione calcolata, se essa è migliore della precedente, si aggiorna Max.

Scandendo l'albero di figura 15.3 in profondità ci si accorge dei tagli possibili. Le cose vanno ancora meglio se ci si accontenta di una soluzione e si visita l'albero sempre in profondità ma andando da destra verso sinitra (ovvero usando una diversa strategia di labeling). In tal caso il limite di 28 è raggiunto subito.

5. Esperimenti ed esercizi

Prendiamo SICStus Prolog, package `clpfd`. Verifichiamo che è verificata/mantenuta la node consistency:

```
:- X in 1..10, X #< 5.
X in 1..4 ?
```

Analizziamo ora l'arc-consistency:

```
:- X in 1..1000000, Y in 1..1000000, X #< Y, Y #< X.
no
:- X in 1..10000000, Y in 1..10000000, X #< Y, Y #< X.
no
```

[1]Si osservi che la tecnica di branch and bound usata in ricerca operativa per risolvere problemi di programmazione lineare intera usando intelligenti chiamate ripetute a problemi rilassati è una tecnica totalmente diversa da questa, sua omonima.

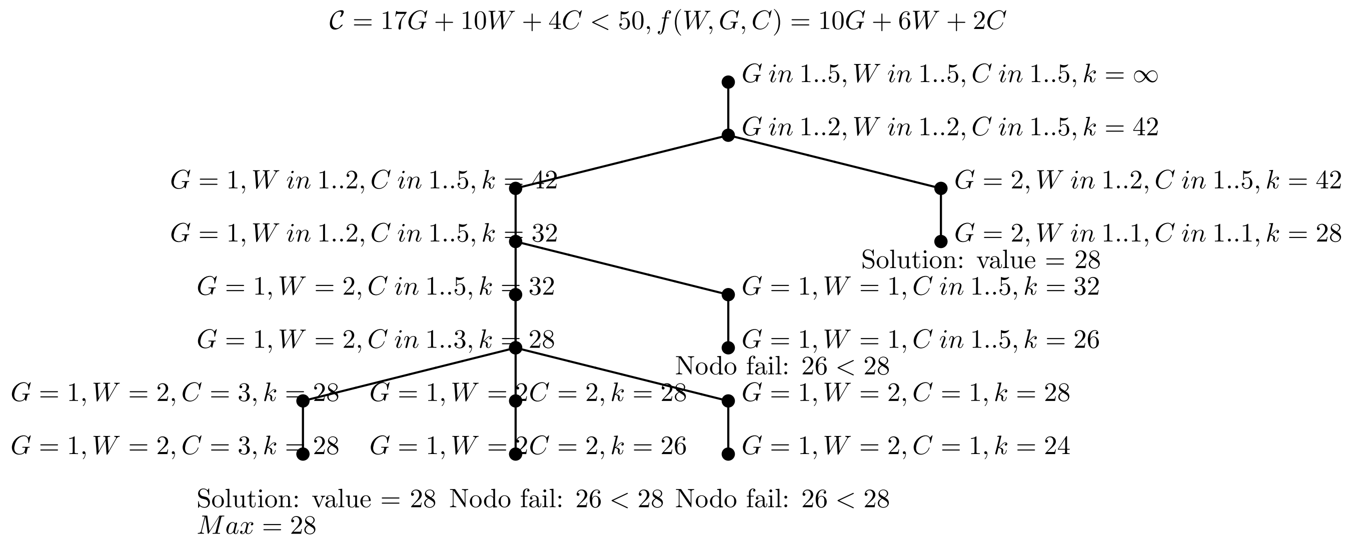


FIGURA 15.3. Esempio di branch and bound

Sembrerebbe verificata. Inoltre è evidente l'aumento di tempo di esecuzione in funzione dell'aumento dei domini. Proviamo comunque a inserire dei "buchi" nei domini in modo da capire se per caso non fosse la bounds consistency ad essere verificata.

```
:- X in {0,1,2}, Y in {0,4}, Y #= 2* X.
X in 0..2,
Y in{0}\/{4} ?
```

Notiamo che, a parte la diversa sintassi per denotare l'insieme $\{0, 4\}$, l'elemento 1 non viene rimosso da D_X. Pertanto sembrerebbe trattarsi di bounds consistency.

Verifichiamo se la path consistency è implementata:

```
:- X in 0..1, Y in 0..1, Z in 0..1, X #\= Y, X #\= Z, Y#\= Z.
X in 0..1,
Y in 0..1,
Z in 0..1 ?
```

Sembrerebbe proprio di no. Vediamo qualcosa circa la hyper arc:

```
:- X in 2..10, Y in 3..20, Z in 0..15, X + Y #= Z.
X in 2..10,
Y in 3..13,
Z in 5..15 ?
```

In questo caso sembrerebbe funzionare. Dunque una forma di propagazione per vincoli n-ari è implementata. Per capire se sia una hyper arc o semplicemente una bounds consistency su predicati n-ari guardiamo il seguente esempio:

```
:- X in {2,10}, Y in {3,4,5,6,7,8,9,10,11,12,23}\/{20},
```

```
    Z in {0,1,3,4,5,6,7,8,9,10,11,12,13,14,15}.
X in{2}\/{10},
Y in(3..12)\/{20}\/{23},
Z in(0..1)\/(3..15) ?
```

Come si poteva sospettare dagli esempi binari precedenti, viene effettuata solo propagazione sugli estremi dei domini.

Per esercizio si ripetano gli esperimenti in ECLiPSe, usando la libreria `ic`. In questo caso, la libreria si richiama con `lib(ic)`, i vincoli `X in a..b` si scrivono `X :: [a..b]`. Il vincolo `X in {a,b,c}` si scrive `X :: [a,b,c]`.

I tempi di esecuzione vengono stampati di default. Sono estremamente utili per vedere il tasso di crescita nel problema $X < Y, Y < X$ che fa sospettare il tipo di propagazione implementato.

Si faccia poi lo stesso in SWI-Prolog. In questo caso la libreria va richiamata come:

```
:- use_module(library('clp/bounds')).
```

La sintassi è coerente con SICStus per le primitive supportate. Non sono ammessi gli insiemi. Dunque per il test di arc consistency si dovrebbe provare:

```
:- X in 0..2,Y in 0..4,Y #\= 2, Y #\= 3, Y #= 2*X.
```

Nel test $X < Y, Y < X$ si vedono alcuni evidenti limiti (quali?) di questa implementazione, che però ha il pregio inestimabile di essere gratuita.

6. Vincoli globali

In Constraint Programming vincoli che riguardano gruppi di variabili (o, iun una accezione più ampia, tutti i vincoli non unari nè binari) sono detti vincoli globali. Solitamente vincoli globali possono essere implementati come congiunzioni di vincoli binari, ma in tal caso la propagazione che si ottiene è spesso molto povera. Pertanto alcuni vincoli globali di uso comune sono sytudiati indipendentemente. In questa sezione studieremo i vincoli di differenza, dopo un breve richiamo ad alcuni risultati di ricerca operativa.

6.1. Grafi bipartiti e matchings. Un *grafo bipartito* è una tripla $G = \langle X, Y, E \rangle$ dove X e Y sono insiemi disgiunti di nodi ed $E \subseteq X \times Y$ è un insieme di archi. Gli archi si intendono non diretti; dunque per noi $(x, y) = (y, x)$.

Dato G bipartito, un *matching* (o accoppiamento) $M \subseteq E$ è un insieme di archi di E tale che nessuna coppia di archi di M condivide un nodo.

Dati G ed M, un nodo si dice *accoppiato* (matched) se è estremo di un arco in M; altrimenti si dice *libero*.

Dati G ed M, un cammino nel grafo G è *alternante* per M se gli archi coinvolti sono alternativamente in M e non in M. E' *aumentante* per M se inizia e termina in nodi liberi. Si osservi che ogni cammino aumentante iniziante in un nodo di X termina in un nodo di Y e viceversa ed è sicuramente aciclico. Si noti che se $M = \emptyset$, ogni insieme contente un solo arco costituisce un cammino aumentante per M.

Nell'esempio in Figura 15.4 sono rappresentati quattro possibili matchings sullo stesso grafo bipartito. $M_1 = \{(b, 2), (e, 3)\}$ è un matching. $P_1 = \{(a, 3), (3, e), (e, 5)\}$ è un cammino aumentante (identificabile dagli archi tratteggiati e dal colore bianco per il nodi iniziale e

a	1	a	1
b	2	b	2
c	3	c	3
d	4	d	4
e	5	e	5
(M_1)		(M_2)	
a	1	a	1
b	2	b	2
c	3	c	3
d	4	d	4
e	5	e	5
(M_3)		(M_4)	

FIGURA 15.4. Accrescimento di matchings tramite cammini aumentanti

finale). Si osservi che rimpiazzando in M l'arco $(e, 3)$ con gli archi $(5, e)$ e $(3, a)$ si ottiene il matching M_2, di cardinalità $|M_1| + 1$. Questo non è un caso:

PROPOSIZIONE 15.1. *Dato un matching M e un cammino aumentante P per M allora $M' = M \oplus P = (M \setminus P) \cup (P \setminus M)$ è un matching tale che $|M'| = |M| + 1$.*

DIM. Sia $M = \{(x_1, y_1), (x_2, y_2), \dots, (x_k, y_k), (x_{k+1}, y_{k+1}), \dots, (x_n, y_n)\}$. Supponiamo che P inizi in $y_0 \in Y$ e termini in $x_0 \in X$ (se così non fosse basta guardarlo al contrario) e che coinvolga i primi k archi di M (se non fosse così, basta riordinare gli archi di M che è un insieme). Dunque P è della forma:

$$\{(y_0, x_1), \underline{(x_1, y_1)}, (y_1, x_2), \underline{(x_2, y_2)}, \dots, (y_{k-1}, x_k), \underline{(x_k, y_k)}, (y_k, x_0)\}$$

con y_0 e x_0 liberi, diversi tra loro e diversi da tutti i nodi che incidono su archi di M. Gli archi sottolineati sono quelli in M. Si consideri

$$\begin{aligned} M' &= (M \setminus P) \cup (P \setminus M) \\ &= \{(y_0, x_1), (y_1, x_2), \dots, (y_{k-1}, x_k), (y_k, x_0), (x_{k+1}, y_{k+1}), \dots, (x_n, y_n)\} \end{aligned}$$

E' un matching in quanto $x_1, \dots, x_n, y_1, \dots, y_n$ sono tutti diversi tra loro in quanto M è matching. x_0 e y_0 sono diversi in quanto liberi. Ha lunghezza $|M| + 1$. $\square$

Siamo interessati a trovare matchings di cardinalità massima. Un matching di massima cardinalità per un grafo bipartito $G = \langle X, Y, E \rangle$ viene detto *matching massimale* (maximum matching). Ovviamente la cardinalità di un matching massimale è minore o uguale a $\min\{|X|, |Y|\}$.

TEOREMA 15.1 (Berge–1957). *Dato un grafo bipartito $G = \langle X, Y, E \rangle$, M è un matching massimale se e solo se non ci sono cammini aumentanti per M.*

DIM. ($\rightarrow$) Sia M massimale e supponiamo per assurdo che ci sia un cammino aumentante per M. Allora (Prop. 15.1) possiamo definire un matching M' di cardinalità maggiore. Assurdo.
($\leftarrow$) Mostriamo che se M non è massimale, allora esiste un cammino aumentante per M. Sia M matching non massimale: allora esiste M' tale che $|M'| > |M|$. Consideriamo il grafo bipartito $B = \langle X, Y, M \oplus M' \rangle$.

(1) Essendo M ed M' matchings, al più un arco di M e uno di M' possono incidere su un nodo di B. Pertanto il grado di ogni nodo di B è al più 2.
(2) Nel grafo B ci possono essere dei cicli. Se ci sono, hanno un numero pari di archi (la metà di M e l'altra metà di M').

Tolti gli archi presenti nei cicli, rimangono dei cammini che coinvolgono alternativamente (in quanto in ogni matching nessuna coppia di archi incide sullo stesso nodo) archi di M e di M'. Poichè $|M'| > |M|$ ci deve essere almeno un cammino con più archi di M' che di M. Poichè il cammino coinvolge alternativamente archi di M e M' e vi sono più archi di M' che di M, deve iniziare e finire con archi di M' che iniziano e terminano rispettivamente in due nodi liberi per M. Dunque è un cammino aumentante per M. □

I due teoremi appena visti ci permettono di giustificare la correttezza di un algoritmo naive per il calcolo di un matching massimale in un grafo bipartito.

Max_Matching_Naive($\langle X, Y, E \rangle$)
1 $M \leftarrow \emptyset$;
2 **while** (esiste un cammino aumentante P per M)
3 **do**
4 $M \leftarrow M \oplus P$;
5 **return** M;

In base al Fatto 15.1 l'algoritmo termina con al più $n = \min\{|X|, |Y|\}$ iterazioni. Sia $m = |E|$.

Studiamo la complessità di ogni singola iterazione (ovvero dato M, trovare se esiste un cammino aumentante). Poichè ogni cammino aumentante ha un nodo libero in X ed uno in Y, per cercare i cammini aumentanti, si può partire da uno solo dei due lati (conviene in quello di cardinalità minima). Supponiamo di partire sempre da X. Si chiami dunque la funzione Find_Augmenting_Path

Find_Augmenting_Path($\langle X, Y, E \rangle, M$)
1 $S \leftarrow X; A \leftarrow E$;
2 $trovato \leftarrow$ false;
3 **while** (S contiene un nodo libero $\wedge \neg trovato$)

```
 4      do
 5          scegli un nodo libero x in S;
 6          ricerca in profondità un cammino aumentante per M in ⟨S, Y, A⟩;
 7          sia E(x) l'insieme dei nodi visitati a partire da x;
 8          if (è stato trovato un cammino)
 9            then
10                trovato ← true
11            else  S ← S \ {x}; A ← A \ E(x);
12  return trovato
```

Si osservi come la rimozione di $E(x)$ non faccia perdere completezza. La ricerca dei cammini aumentanti parte sempre da un nodo libero. Se un arco viene visitato per la prima volta in tale ricerca e non conduce a nessun cammino, non lo potrà fare nemmeno in un secondo momento, partendo da un altro nodo libero. Tale procedura costa evidentemente $O(|E|)$. Globalmente pertanto, il costo dell'algoritmo naive è nm.

Si osservi come selezionando il cammino aumentante $\{(c, 3), (3, a), (a, 1)\}$ nel grafo (M_2) nella figura 15.4 si giunge al matching (M_3) dal quale selezionando l'unico cammino aumentante $\{(d, 5), (5, e), (e, 4)\}$ si giunge al matching massimale (M_4). Ovviamente un matching massimale non è necessariamente unico (ad esempio basta rimpiazzare $\{(b, 2), (c, 3)\}$ con $\{(b, 3), (c, 2)\}$ in M_4).

L'algoritmo proposto può essere velocizzato aumentando un matching con più di un cammino aumentante ad ogni iteratione. Sviluppando bene quest'idea si giunge all'algoritmo di Hopcroft e Karp che ha complessità $O(m\sqrt{n}) = O(n^2\sqrt{n})$ [**HK73**]. A tale complessità si può giungere anche mediante algoritmi ottimi per il flusso massimo in un grafo (si veda, ad esempio, [**PS98**]).

6.2. Vincoli di differenza. Siano $X_1, \ldots, X_k$ variabili con domini rispettivi $\mathcal{D}_1, \ldots, \mathcal{D}_k$. Il vincolo k-ario $\texttt{all_diff}(X_1, \ldots, X_k)$ è definito semanticamente come:

$$\begin{array}{l}\texttt{all_diff}(X_1, \ldots, X_k) = (\mathcal{D}_1 \times \cdots \times \mathcal{D}_k)\setminus \\ \qquad \{(a_1, \ldots, a_k) \in \mathcal{D}_1 \times \cdots \times \mathcal{D}_k \,:\, \exists i \exists j\, 1 \leq i < j \leq k\, (a_i = a_j)\}\end{array}$$

Date $X_1, \ldots, X_k$ variabili con domini rispettivi $\mathcal{D}_1, \ldots, \mathcal{D}_k$ un vincolo di differenza k-ario $\texttt{all_diff}(X_1, \ldots, X_k)$ è *hyper arc consistent* se per ogni $i \in \{1, \ldots, k\}$ e per ogni $a_i \in \mathcal{D}_i$ vale che esistono $a_1 \in \mathcal{D}_1, \ldots, a_{i-1} \in \mathcal{D}_{i-1}, a_{i+1} \in \mathcal{D}_{i+1}, \ldots, a_k \in \mathcal{D}_k$ tali da soddisfare il vincolo $\texttt{all_diff}(X_1, \ldots, X_k)$ (ovvero $a_1, \ldots, a_k$ sono tutti diversi tra loro).

Un CSP è *diff-arc consistent* se ogni vincolo di differenza in esso è *hyper arc consistent.*

Si considerino i CSP $\langle \texttt{all_diff}(X_1, \ldots, X_k); \mathcal{D}_\in \rangle$ e $\langle X_1 \neq X_2, X_1 \neq X_3, \ldots, X_1 \neq X_k, X_2 \neq X_3, \ldots, X_{k-1} \neq X_k; \mathcal{D}_\in \rangle$. La proprietà di hyper-arc-consistency di $\texttt{all_diff}(X_1, \ldots, X_k)$ implica l'arc consistency binaria nel secondo CSP. Il viceversa invece non vale (esercizio). I due CSP tuttavia sono equivalenti (hanno esattamente le stesse soluzioni).

A livello delle proprietà di consistenza locale, $\langle \texttt{all_diff}(X_1, \ldots, X_k); \mathcal{D}_\in \rangle$ coincide invece con il CSP $\langle \bigwedge_{i=1}^{k-1} \bigwedge_{j=i+1}^{k} X_i \neq X_j; \mathcal{D}_\in \rangle$, ove si consideri congiunzione di vincoli binari un unico vincolo k-ario.

Sia $d_i = |\mathcal{D}_i|$ per $i \in \{1, \ldots, k\}$. Sia $d = \max_{i=1}^{k}\{d_i\}$. Un algoritmo per la propagazione delle hyper-arc-consistency di `all_diff`$(X_1, \ldots, X_k)$ basata sulla definizione avrebbe un costo dell'ordine di $d_1 d_2 \ldots d_k = O(d^k)$.

Dato un vincolo di differenza C sulle variabili $X_1, \ldots, X_k$, con domini rispettivi $\mathcal{D}_1, \ldots, \mathcal{D}_k$, definiamo il grafo bipartito $GV(C) = \langle X_C, Y_C, E_C \rangle$ nel seguente modo:

- $X_C = \{X_1, \ldots, X_k\}$
- $Y_C = \bigcup_{i=1}^{k} \mathcal{D}_i$
- $E_C = \{(X_i, a) : a \in \mathcal{D}_i\}$

Ad esempio, in Fig. 6.2 a sinistra è illustrato il grafo per:

$$\langle \ \texttt{all_diff}(X_1, \ldots, X_7); \quad X_1 \in 1..2, X_2 \in 2..3, X_3 \in \{1,3\}, X_4 \in \{2,4\}, X_5 \in 3..6, X_6 \in 6..7, X_7 \in \{8\} \rangle$$

Teorema 15.2 (Regin [**Rég94**]). *Un CSP $\mathcal{P} = (\mathcal{C}; \mathcal{D}_{\in})$ è diff-arc consistent se e solo se per ogni vincolo di differenza C in $\mathcal{C}$ ogni arco in $GV(C)$ appartiene ad un matching di cardinalità pari al numero di variabili di C.*

Dim. Sia dato C in $\mathcal{C}$ vincolo di differenza. Siano $X_1, \ldots, X_k$ le sue variabili.
($\rightarrow$) Scegliamo un arco (X_i, a_i) in $GV(C)$. Poichè $\mathcal{P}$ è diff-arc consistent, C è hyper-arc consistent. Dunque esistono $a_1, \ldots, a_{i-1}, a_{i+1}, \ldots, a_k$ tali che $X_1 = a_1, \ldots, X_k = a_k$ è soluzione di C. Questa soluzione individua un matching della cardinalità cercata.
($\leftarrow$) Sia $a_i \in D_{X_i}$, dunque l'arco (X_i, a_i) appartiene ad un matching di cardinalità k. Da quel matching troviamo i valori per le altre variabili per verificare la proprietà di hyper-arc consistency. □

Dal teorema sopra emerge che un vincolo di differenza C viene rappresentato in modo compatto dal suo grafo $GV(C)$. Nota che:

- con la rappresentazione con vincoli binari mi servivano $k(k-1)/2$ archi e di spazio $d_1 + \cdots + d_k$ per i domini.
- con la rappresentazione `all_diff`$(X_1, \ldots, X_k)$ serve $O(k)$ (per scrivere il termine qui a sinistra) e di spazio $d_1 + \cdots + d_k$ per i domini.
- con la rappresentazione a grafo si usano $O(k)$ (per i nodi variabile), $|\mathcal{D}_1 \cup \cdots \cup \mathcal{D}_k|$ per gli oggeti del dominio, più $e = d_1 + \cdots + d_k \leq kd$ per gli archi.

Sappiamo che possiamo trovare un matching massimale in tempo $O(\sqrt{k}e) = O(k^{3/2}d)$. Però noi vogliamo tenere solo gli archi che appartengono a TUTTI i matching massimali. Per evitare di farlo enumerativamente, usiamo il seguente risultato:

Teorema 15.3 (Berge–1970). *Sia $G = \langle X, Y, E \rangle$ un grafo bipartito. Un arco appartiene ad alcuni ma non a tutti i matching massimali se e solo se, per un arbitrario matching massimale M, l'arco appartiene a:*

- *un cammino alternante PARI che inizia in un vertice libero, oppure*
- *un ciclo alternante PARI.*

Dim. ($\leftarrow$) Sia M un matching massimale.

(1) Sia P un cammino alternante pari con un estremo libero (e dunque l'altro no). $M' = M \oplus P$ è un altro matching con la stessa cardinalità. Metà archi di P stanno in uno, metà nell'altro.

(2) Sia P un ciclo alternante pari. Si può ripetere la considerazione suddetta.

($\rightarrow$) Sia (x, y) un arco presente in tutti i matching massimali. Allora non può stare nè in un cammino alternante pari nè in un ciclo alternante pari. In entrambi i casi infatti sapremmo trovare un matching della stessa cardinalità e che non contiene l'arco suddetto, contraddicendo all'ipotesi.

Sia (x, y) un arco presente in nessun matching massimale. Si consideri un matching massimale M.

Almeno uno tra x e y deve essere accoppiato in M (altrimenti potremmo aggiungere (x, y) a M trovando un matching di cardinalità superiore).

Se x è libero e y no, allora esiste un arco (z, y) in M. Ma allora rimpiazzando (z, y) con (x, y) otteniamo un matching massimale che contiene (x, y): assurdo.

Pertanto x non può essere libero. Similmente per y.

Poichè (x, y) non è presente in nessun matching massimale ma sia x che y sono accoppiati in tutti i matching massimali l'arco (x, y) non può essere presente nè in un cammino alternante pari nè in un ciclo alternante pari. In entrambi i casi infatti sapremmo trovare un matching della stessa cardinalità e che contiene l'arco suddetto, contraddicendo all'ipotesi. □

Si consideri l'esempio in Figura 6.2 a sinistra. Si tratta di un matching massimale M (che è stato pertanto trovato in tempo $O(k^{3/2}d)$).

Il nodo 7 è libero (per evidenziarlo, è l'unico nodo non colorato). Tutti gli archi raggiungibili da cammini alternanti pari a partire da lui sono i 4 archi riportati nella figura 6.2 in centro. Partendo dal nodo 7, tutto i cammini che andando da dx verso sx usano un arco libero e da sx verso dx usano un arco del matching. Risulta pertanto naturale cercare tali cammini su un grafo diretto con gli archi di matching rivolti a sinistra e gli archi a destra. I 4 archi indicati nella figura centrale fanno pertanto parte di alcuni ma non tutti i matching massimali.

A questo punto, finiti i nodi liberi, si individuano i cicli, sempre nel grafo diretto. Questo si può fare cercando le componenti fortemente connesse. Si giunge ad identificare gli archi nella terza figura.

Alcuni archi sono rimasti fuori da queste due visite. Due di questi, $(X_4, 4)$ e $(X_7, 8)$, essendo presenti in questo matching e non soddisfacendo i requisiti del teorema, devono essere presenti in tutti i matching.

Invece gli archi: $(X_4, 2), (X_5, 3), (X_5, 4)$ non stanno in nessun matching massimale e pertanto possono essere eliminati.

La prima fase si può implementare in tempo $O(e) = O(kd)$, la seconda in tempo $O(e+k+d)$. Pertanto la diff-arc consistency può essere vericata in tempo $O(k^{3/2}d)$ (stessa complessità dell'algoritmo di Hopcroft e Karp). Tale fase di propagazione viene anche detta *filtering*.

X_7	8	X_7	8	X_7	8
	7		7		7
X_6	6	X_6	6	X_6	6
X_5	5	X_5	5	X_5	5
X_4	4	X_4	4	X_4	4
X_3	3	X_3	3	X_3	3
X_2	2	X_2	2	X_2	2
X_1	1	X_1	1	X_1	1

FIGURA 15.5. Applicazione del filtering algorithm

CAPITOLO 16

Programmazione logica con vincoli

In questo capitolo studieremo una forma di programmazione dichiarativa affine al Prolog, la *programmazione logica con vincoli* (*constraint logic programming, CLP*). L'idea portante di questo genere di programmazione è il conciliare la dichiaratività di Prolog con metodi di computazione orientati a specifici domini del discorso. Come vedremo un programma CLP è molto simile ad un programma Prolog. Il meccanismo inferenziale, goal-driven, è infatti quello della SLD-risoluzione: da Prolog si ereditano (con opportuni adeguamenti) i concetti di clausola, goal, derivazione di successo o di fallimento, risposta, ecc..

L'arricchimento rispetto a Prolog consiste nel scegliere, per una parte dei simboli del linguaggio, una particolare interpretazione su un prefissato dominio del discorso. Ovviamente, a seconda del dominio prefissato la semantica che risulterà assegnata ad ogni programma (ovvero, in parole povere, l'insieme delle risposte calcolate ottenibili rispetto a quel programma) sarà diversa. Parleremo quindi, ad esempio, di programmazione logica con vincoli sul dominio dei numeri interi, o sul dominio dei numeri reali, e così via. I letterali che posseggono una interpretazione prefissata sono detti *vincoli* (o *constraint*). Avendo prefissato una interpretazione, o meglio un modello, per i constraint, questi ultimi possono essere valutati in tale modello.

La componente "in più" rispetto a Prolog consiste quindi nel non trattare i constraint tramite la SLD-risoluzione, ma nel verificarne la soddisfacibilità facendo ricorso ad un *risolutore di vincoli* (constraint solver). Tale risolutore viene impiegato sia allo scopo di abilitare ogni singolo passo di derivazione, sia per semplificare le eventuali congiunzioni di constraint che si generano durante il processo di inferenza.

Un potenziale vantaggio dal punto di vista computazionale, nasce dal fatto che solitamente i risolutori di vincoli possono essere implementati in modo molto efficiente, utilizzando le tecniche e gli algoritmi più diversi. Ne risulta una integrazione, per lo più trasparente al programmatore, tra il paradigma di programmazione logico e altri paradigmi e metodologie di programmazione.

1. Sintassi e semantica operazionale

Riprendiamo le nozioni sui linguaggi del primo ordine introdotti nel Capitolo 2 adeguandoli al contesto della programmazione logica con vincoli. In particolare, consideriamo un linguaggio del primo ordine basato su un alfabeto $\langle \Pi, \mathcal{F}, \mathcal{V} \rangle$. Inoltre supponiamo che l'insieme Π dei simboli di predicato sia partizionato in due: $\Pi = \Pi_C \cup \Pi_P$ con $\Pi_C \cap \Pi_P = \emptyset$. L'insieme Π_P identifica i simboli predicativi definiti nel programma mentre Π_C è un insieme di simboli di predicato di constraint (che non possono essere definiti dalle regole del programma, cioé non possono occorrere nelle teste delle regole). Una ulteriore assunzione è che Π_C contenga il predicato di uguaglianza "=".

La nozione di *constraint* riveste un ruolo peculiare nella programmazione logica con vincoli. Abbiamo già utilizzato la parola "constraint" in altre occasioni nei capitoli precedenti. La prossima definizione stabilisce cosa denoterà questo termine nel resto di questo capitolo.

DEFINIZIONE 16.1. Un *constraint primitivo* è un letterale sull'alfabeto $\langle \Pi_C, \mathcal{F}, \mathcal{V} \rangle$.

Assumiamo che true e false denotino due constraint primitivi.[1] Un *constraint* è una congiunzione di constraint primitivi. Se $p \in \Pi_P$, e $t_1, \ldots, t_n$ sono termini, allora l'atomo $p(t_1, \ldots, t_n)$ è detto *atomo di programma*.

ESEMPIO 16.1. I seguenti sono constraint primitivi (assumendo $\Pi_C \supseteq \{=, <, \geq, \in, \subseteq\}$)

$$X = Z \quad X \neq Z \quad X \neq Y \quad 0 < X \quad 0 \geq X \quad X \in Y \quad A \subseteq B$$

Riformuliamo ora le definizioni di programmi, regola, goal, ecc., nel contesto del CLP.

DEFINIZIONE 16.2. Un *goal CLP* è una scrittura della forma $\leftarrow \bar{B}$, dove $\bar{B}$ è una congiunzione di atomi di programma e di constraint primitivi.[2]

ESEMPIO 16.2. La formula $\leftarrow X \neq Y, p(X_1, X_2), X_1 \leq X, Y_2 > Y$ è un goal CLP mentre $\leftarrow p(X, Y), \neg q(Y, Z)$ non lo è.

DEFINIZIONE 16.3. Un *fatto CLP* è un atomo $p(t_1, \ldots, t_n)$, dove $p \in \Pi_P$ e ogni t_i è un termine. Una *regola CLP* è una clausola della forma

$$p(t_1, \ldots, t_n) \leftarrow \bar{B}$$

dove $\leftarrow \bar{B}$ è un goal CLP, $p \in \Pi_P$ e ogni t_i è un termine.

Si osservi che una regola CLP non è necessariamente una clausola di Horn. Ad esempio la regola

$$p(X, Y) \leftarrow X \neq Y, X < Z, q(X, Z)$$

è equivalente alla disgiunzione

$$p(X, Y) \vee (X = Y) \vee \neg(X < Z) \vee \neg q(X, Z)$$

in cui occorrono due letterali positivi.

Un concetto fondamentale nella programmazione logica con vincoli è quello di *stato*. Le prossime definizioni fanno riferimento a due procedure, *solv* e *simpl*, che studieremo in seguito.

DEFINIZIONE 16.4. Uno *stato* è una coppia $\langle G \,|\, C \rangle$ dove G è un goal CLP e C è un constraint (anche detto *constraint store*).

Uno stato è detto di *successo* se ha la forma $\langle \leftarrow \Box | C \rangle$ e vale $solv(C) \neq \mathsf{false}$.

Uno stato è di *fallimento* se ha la forma $\langle G | C \rangle$ e vale $solv(C) = \mathsf{false}$, oppure se G è una congiunzione di atomi di programma e non vi è nessuna regola la cui testa abbia simboli di predicato di programma occorrenti nel goal.

---

[1] Potremmo anche definirli come $a = a$ e $a \neq a$, rispettivamente.

[2] Anche in questo caso denoteremo la congiunzione utilizzando il simbolo ",".

DEFINIZIONE 16.5. Sia P un programma e G_1 un goal. Un *passo di derivazione CLP*, denotato con

$$\langle G_1 \,|\, C_1 \rangle \Rightarrow \langle G_2 \,|\, C_2 \rangle$$

è definito nel seguente modo: sia $G_1 = \leftarrow L_1, \ldots, L_m$ con $m \geq 1$. Assumiamo che il letterale selezionato sia L_1, allora:

(1) se L_1 è un constraint primitivo, allora si pone $C_2 = L_1 \wedge C_1$. Inoltre se $solv(C_2) = \mathsf{false}$, allora si pone $G_2 = \leftarrow \Box$, altrimenti si pone $G_2 = \leftarrow L_2, \ldots, L_n$.
(2) se invece $L_1 = p(t_1, \ldots, t_n)$ è un atomo di programma e $p(s_1, \ldots, s_n) \leftarrow \bar{B}$ è una rinomina di una regola di P, allora si pone $G_2 = \leftarrow t_1 = s_1, \ldots, t_n = s_n, \bar{B}, L_2, \ldots, L_n$ e $C_2 = C_1$.

Una *derivazione per uno stato* S_0 è una sequenza massimale di passi di derivazione che hanno S_0 come primo stato:

$$S_0 \Rightarrow S_1 \Rightarrow \cdots$$

Una *derivazione per un goal CLP G* è una derivazione per lo stato $\langle G \,|\, \mathsf{true} \rangle$.

Una *derivazione* (di lunghezza finita) $S_0 \Rightarrow \cdots \Rightarrow S_n$ è detta di *successo* se S_n è uno stato di successo. In tal caso la *risposta calcolata* è definita essere $simpl(C_n, vars(S_0))$.

Una *derivazione* $S_0 \Rightarrow \cdots \Rightarrow S_n$ è invece è di *fallimento* se S_n è uno stato di fallimento.

Come menzionato la definizione (e conseguentemente la relativa procedura) di derivazione si basa su due funzioni *solv* e *simpl*. Per esse non viene data una definizione rigida: la loro specifica dipende dal dominio in cui vengono valutati i constraint. Al fine di comprendere il loro scopo e significato analizziamo degli esempi specifici.

Si consideri il dominio dei numeri reali $\mathbb{R}$, abbiamo la seguente situazione:

$$\begin{array}{c} solv(X < 5, 4 < X, X \neq 0) \neq \mathsf{false} \\ \Downarrow simpl \\ 4 < X, X < 5 \end{array}$$

Mentre nel dominio dei numeri naturali $\mathbb{N}$:

$$solv(X < 5, 4 < X, X \neq 0) = \mathsf{false}.$$

Si osserva quindi che la valutazione dello stesso vincolo può dare risultati diversi a seconda del modello scelto. Domini frequentemente utilizzati sono CLP($\mathbb{N}$), CLP($\mathbb{R}$), CLP(FD) (CLP su domini finiti), CLP($\mathcal{SET}$) (CLP con vincoli di tipo insiemistico), ecc.

Il prossimo è invece un esempio più concreto dell'uso di *simpl*. Consideriamo la seguente derivazione rispetto a CLP($\mathbb{N}$):

$$\begin{array}{c} \langle \leftarrow p(X) | \mathsf{true} \rangle \\ \Downarrow \\ \vdots \\ \Downarrow \\ \langle \leftarrow \Box \,|\, X = f(X_1), X_2 = f(X_3), X_2 \neq X_3, Y > W, X_1 = f(X_4), X_4 < 3, X_4 \neq 4 \rangle \\ \Downarrow simpl \\ X = f(f(X_4)), X_4 < 3 \end{array}$$

Lo scopo di *simpl* è quindi quello di semplificare un constraint. Poiché in Π c'è sempre l'uguaglianza, la funzione *simpl* deve avere la capacità di effettuare l'unificazione sintattica (in certi casi è prevista anche quella semantica—E-unificazione).

Esempio 16.3. Sia P il programma:

$$\begin{array}{l}\mathsf{num}(0).\\ \mathsf{num}(s(X)) \leftarrow \mathsf{num}(X).\end{array}$$

Si consideri la seguente derivazione per lo stato $S_0 = \langle \leftarrow \mathsf{num}(s(s(0)))|\mathsf{true}\rangle$:

$$\begin{array}{cc}
\langle \leftarrow \mathsf{num}(s(s(0)))|\mathsf{true}\rangle & \\
| & \mathsf{num}(s(X_1)) \leftarrow \mathsf{num}(X_1) \\
\langle \leftarrow s(X_1) = s(s(0)), \mathsf{num}(X_1)|\mathsf{true}\rangle & \\
| & \\
\langle \leftarrow \mathsf{num}(X_1)|s(X_1) = s(s(0))\rangle & solv\big(s(X_1) = s(s(0))\big) \neq \mathsf{false} \\
| & \mathsf{num}(s(X_2)) \leftarrow \mathsf{num}(X_2) \\
\langle \leftarrow X_1 = s(X_2), \mathsf{num}(X_2)|s(X_1) = s(s(0))\rangle & \\
| & \\
\langle \leftarrow \mathsf{num}(X_2)|\underbrace{X_1 = s(X_2), s(X_1) = s(s(0))}_{c_1}\rangle & solv(c_1) \neq \mathsf{false} \\
| & \\
\langle \leftarrow X_2 = 0|X_1 = s(X_1), s(X_1) = s(s(0))\rangle & \\
| & \\
\langle \leftarrow \Box|\underbrace{X_2 = 0, X_1 = s(X_2), s(X_1) = s(s(0))}_{c_2}\rangle & solv(c_2) \neq \mathsf{false}
\end{array}$$

Si osservi come si sia simulata (ma con granularità più fine) una SLD derivazione. In questo caso l'unificazione è svolta da *solv*.

La funzione *simpl* dovrà poi fornire l'output, ovvero la risposta. *simpl* agisce sul vincolo

$$X_2 = 0, X_1 = s(X_2), s(X_1) = s(s(0))$$

che viene semplificato in:

$$\begin{array}{c}X_2 = 0\\ X_1 = s(0)\end{array}$$

inoltre, restringendosi alle variabili del goal nell stato S_0, dato che $vars(G_0) = \emptyset$, la risposta fornita sarà vuota.

Esempio 16.4. Si consideri il programma in CLP($\mathbb{R}$):

$$\begin{array}{c}
p(X,Y) \leftarrow X = Y + 2, Y \geq 0, q(X,Y).\\
q(X,Y) \leftarrow X \leq 1, r(X,Y).\\
q(X,Y) \leftarrow X \leq 3, r(X,Y).\\
r(2,0).\\
r(X,1).
\end{array}$$

Allora la seguente sarà una derivazione per lo stato $S_0 = \langle \leftarrow p(X,Y)|\text{true}\rangle$:

$$
\begin{array}{cr}
\langle \leftarrow p(X,Y) | \text{true} \rangle & \\
| & p(X_1,Y_1) \leftarrow X_1 = Y_1 + 2, Y_1 \geq 0, q(X_1,Y_1) \\
\langle \leftarrow X = X_1, Y = Y_1, X_1 = Y_1 + 2, Y_1 \geq 0, q(X_1,Y_1) | \text{true} \rangle & \\
| & \text{in 4 passi} \\
\langle q(X_1,Y_1) | \underbrace{Y_1 \geq 0, X_1 = Y_1 + 2, Y = Y_1, X = X_1}_{(c_1)} \rangle & solv(c_1) \neq \mathsf{false} \\
| & q(X_2,Y_2) \leftarrow X_2 \leq 1, r(X_2,Y_2) \\
\langle \leftarrow X_1 = X_2, Y_1 = Y_2, X_2 \leq 1, r(X_2,Y_2) | (c_1) \rangle & \\
| & \\
\langle \leftarrow r(X_2,Y_2) | \underbrace{X_2 \leq 1, X_1 = X_2, Y_1 = Y_2, (c_1)}_{(c_2)} \rangle & solv(c_2) = \mathsf{false}
\end{array}
$$

A questo punto, similmente a quanto accade in Prolog, viene effettuato il backtracking fino all'ultima possibilità di scelta, cioè fino a $\langle q(X_1,Y_1) | Y_1 \geq 0, X_1 = Y_1 + 2, Y = Y_1, X = X_1 \rangle$. Da questo stato si sviluppa la seguente derivazione:

$$
\begin{array}{cr}
\langle q(X_1,Y_1) | \underbrace{Y_1 \geq 0, X_1 = Y_1 + 2, Y = Y_1, X = X_1}_{(c_1)} \rangle & solv(c_1) \neq \mathsf{false} \\
| & q(X_2,Y_2) \leftarrow X_2 \leq 3, r(X_2,Y_2) \\
\langle \leftarrow X_2 = X_1, Y_2 = Y_1, X_2 \leq 3, r(X_2,Y_2) | (c_1) \rangle & \\
| & \text{in 3 passi} \\
\langle \leftarrow r(X_2,Y_2) | \underbrace{X_2 = X_1, Y_2 = Y_1, X_2 \leq 3, (c_1)}_{c_3} \rangle & solv(c_3) \neq \mathsf{false} \\
| & r(X_3, 1) \\
\langle \leftarrow X_2 = X_3, Y_2 = 1 | (c_3) \rangle & \\
| & \text{in 2 passi} \\
\langle \leftarrow \Box | X = Y + 2, Y > 0, X \leq 3, Y = 1 \rangle & \\
\Downarrow & \\
X = 3, 1 > 0, 3 \leq 3, Y = 1 &
\end{array}
$$

Quest'ultimo stato, tramite *simpl*, fornisce la risposta $X = 3, Y = 1$

2. CLP(FD) in SICStus

In questa sezione e in gran parte di questo e del successivo capitolo descriveremo il constraint solver su domini finiti offerto da SICStus Prolog. Altri CLP-solvers, quali GNU-Prolog, offrono un trattamento simile che solitamente si differenzia solamente per la differente sintassi adottata. La Sezione 5 è dedicata a illustrare le principali differenze tra i solver di SICStus Prolog e GNU-Prolog, relativamente ai vincoli su domini finiti.

Innanzitutto, per utilizzare il constraint solver sui domini finiti di SICStus Prolog è necessario caricare una apposita libreria. Ciò viene fatto con la direttiva:

```
:- use_module(library(clpfd)).
```

In seguito sarà possibile utilizzare i costrutti che descriveremo nelle prossime pagine. Analizziamo quindi le principali funzionalità relative a questo constraint solver.

2.1. Assegnare/restringere domini a variabili. Per una singola variabile, il constraint:

```
A in 1..20
```

impone che `A` possa assumere solo valori tra 1 e 20.
Per una lista di variabili `L = [`A_1`,...,`A_n`]` si utilizza invece un constraint della forma:

```
domain(L,-5,12)
```

Una alternativa a SICStus e GNU-Prolog è l'interprete ECLiPSe, disponibile presso il sito `http://eclipse.crosscoreop.com/eclipse`. Lo menzioniamo in questo punto perchè questo interprete offre una comoda notazione per denotare questo tipo di constraint. In ECLiPSe è possibile scrivere un constraint della forma

```
L ::  1..20
```

per vincolare il dominio di tutte le variabili nella lista `L` ai valori tra 1 e 20. Per utilizzare una simile notazione anche in SICStus è sufficiente definire l'operatore `::` ed aggiungere al programma una clausola ausiliaria:

```
:- op(100,xfx, ::).
L ::  A..B :- domain(L,A,B).
```

Nel caso di voglia assegnare come dominio di una variabile vincolata un insieme di valori non consecutivi `X`, allora SICStus mette a disposizione il vincolo `?X in_set +FDSet`, dove `FDSet` deve essere un "FD-insieme", ovvero un insieme di valori denotato adottando la rappresentazione propria di SICStus. Non forniamo qui una trattazione particolareggiata delle caratteristiche di questa struttura dati, il lettore interessato può fare riferimento al manuale di SICStus Prolog. Per gli scopi di questo capitolo ci basti sapere che possiamo utilizzare gli FD-insiemi in modo semplice nel seguente modo: per svincolare i programmi dalla rappresentazione degli FD-insiemi utiliziamo il predicato built-in `list_to_fdset(+List, -FDSet)` per convertire una lista di valori numerici in un FD-insieme. Ecco un esempio che vincola il dominio di `X` ad essere l'insieme {`10,2,32,5,7,1`}:

```
?- list_to_fdset([10,2,32,5,7,1], FDS),
   X in_set FDS.
```

2.2. Operatori matematici. In CLP(FD) si possono utilizzare gli usuali operatori aritmetici e simboli relazionali per imporre dei vincoli sui possibili valori che delle variabili vincolate potranno assumere. Per fare ciò è necessario anteporre il simbolo `#` agli usuali simboli predicativi di confronto. Così facendo otteniamo i corrispondenti vincoli. Alcuni simboli predicativi di constraint ammessi sono:

```
#=   #<   #>   #=<   #>=   #\=
```

I simboli funzionali ammessi (che operano come usuali operatori atirmetici) sono invece:

```
+   -   *   /   mod   min   max   abs
```

Esempio 16.5. Consideriamo il goal

```
?- domain([A,B,C],1,4), A #< B, B #< C.
```

ad esso viene risposto:

```
A in 1..2,
B in 2..3,
C in 3..4
```

Si noti come la forma di risposta non sia molto espressiva (i vincoli non vengono esplicitati). Ciò che la risposta asserisce è semplicemente che esistono delle soluzioni e conseguentemente vengono mostrate le restrizioni dei domini delle variabili coinvolte. Questa è una scelta implementativa di SICSstus. Si osservi che anche la soluzione (sbagliata) `A=2, B=2, C=3` soddisfa i vincoli forniti ai domini. È possibile comunque forzare l'esplicitazione dei vincoli eseguendo la seguente asserzione (essa modifica uno dei parametri di funzionamento di SICStus):

```
:- clpfd:assert(clpfd:full_answer).
```

Ri-sottomettendo lo stesso goal, ora la risposta che si ottiene è completa:

```
clpfd:'t>=u+c'(B,A,1),
clpfd:'t>=u+c'(C,B,1),
A in 1..2,
B in 2..3,
C in 3..4
```

Si osservi che se invece avessimo sottoposto il goal

```
domain([A,B,C],1,3), A #< B, B #< C.
```

avremmo (in entrambi i casi) ottenuto come risposta

```
A=1, B=2, C=3
```

2.3. Comandi per istanziare variabili. I comandi illustrati nella sezione precedente permettono di assegnare un insieme di valori ammissibili per una o più variabili della clausola. Ora descriveremo alcuni comandi CLP(FD) utili a effettuare la istanziazione di variabili con valori ammissibili (cioé che soddisfano i vincoli imposti dai comandi appena visti).

Per istanziare una singola variabile `V` con uno dei possibili valori compresi nel suo dominio si utilizza:

```
indomain(V)
```

Se questo atomo viene valutato più volte per effetto del backtracking, ad ogni valutazione si avrà una diversa istanziazione della variabile in corrispondenza a diversi valori appartenenti al suo dominio.

Particolari valori del dominio associato alla variabile `Var` si possono ottenere tramite:

- `fd_max(Var,Max)`: la variabile `Max` viene istanziata al massimo valore del dominio ammesso (in quel momento) per `Var`.
- `fd_min(Var,Max)`: la variabile `Min` viene istanziata al minimo valore del dominio ammesso (in quel momento) per `Var`.

Per istanziare tutte le variabili occorrenti una lista di variabili si utilizza l'atomo:

```
labeling(Opzioni, Lista)
```

Si assume che a tutte le variabili occorrenti in `Lista` siano stati assegnati dei domini finiti. In questo modo, tramite il backtracking, il predicato `labeling` provvede a generare tutte le possibili istanziazioni che rispettano tali domini. Il modo in cui ciò avviene è determinato

dalle opzioni indicate tramite il parametro `Opzioni`. Esso può essere il termine `[]`, oppure una lista di parole chiave che determinano l'algoritmo con cui le varie alternative vengono generate. Le opzioni quindi controllano:

- l'ordine in cui le variabili della lista vengono selezionate per essere istanziate (variable choice heuristic);
- il modo in cui i valori del dominio vengono scelti per essere "assegnati" alle variabili (value choice heuristic);
- se debbano essere generate tutte le possibili istanziazioni, se debba esserne generata una sola, e se una soluzione debba essere generata solo se è ottimale (chiariremo questo punto in seguito).

Le opzioni sono suddivise in quattro gruppi. È possibile selezionare al più una opzione per ogni gruppo.
Si tenga presente che in ogni istante, per ogni data variabile della lista, c'è un insieme finito di valori tra cui scegliere per effettuare l'istanziazione; per semplicità definiamo il *lower bound* come il valore minimo tra essi, e similmente, l'*upper bound* come il valore massimo. I quattro gruppi sono:

I gruppo: le opzioni di questo gruppo determinano il criterio con cui viene scelta la prossima variabile da istanziare.

leftmost: viene scelta sempre la variabile più a sinistra nella lista (ovviamente, tra quelle non ancora istanziate). Questa è la opzione di default se non si seleziona nessuna delle parole chiave di questo gruppo.

min: viene scelta la variabile che ha il lower bound più piccolo. A parità di lower bound si sceglie quella più a sinistra.

max: viene scelta la variabile che ha l'upper bound più grande. A parità di upper bound si sceglie quella più a sinistra.

ff: si utilizza il proncipio detto *first-fail*: si sceglie la variabile che ha il dominio più piccolo. A parità di cardinalità del dominio si sceglie quella più a sinistra.

ffc: questa è la euristica più restrittiva: si sceglie una variabile che abbia il dominio più piccolo; in caso di molteplici possibilità si discrimina utilizzando due criteri, nell'ordine:

(1) selezionando la variabile che è vincolata dal maggior numero di constraint non ancora valutati;
(2) selezionando la variabile più a sinistra.

variable(Sel): dove `Sel` deve essere un simbolo di predicato. In questo caso il predicato `Sel` viene utilizzato per selezionare la prossima variabile. In particolare, se `Vars` è la lista delle variabili non ancora selezionare SICStus effettuerà una valutazione dell'atomo `Sel(Vars,Selected,Rest)`.
Si assume che `Sel` sia stato dichiarato e che abbia successo in modo deterministico, unificando le variabili `Selected` e `Rest` con la variabile selezionata e il resto della lista delle variabili, rispettivamente.
`Sel` può anche essere un termine composto, come ad esempio `selettore(Param)`. In questo caso SICStus valuterà `selettore(Param,Vars,Selected,Rest)`.

II gruppo: una volta seleziona una variabile, diciamo `X`, seguendo la strategia determinata dalle opzioni del I gruppo, le opzioni di questo gruppo determinano come

selezionare il prossimo valore da utilizzare per istanziare `X`:

step: effettua una scelta binaria tra le possibilità `X#=B` and `X#\=B`, dove `B` è il lower o l'upper bound per `X`. Questo è il comportamento di default.

enum: effettua una scelta multipla selezionando tra tutti i valori possibili per `X`.

bisect: effettua una scelta binaria tra `X#=<M` e `X#>M`, dove `M` è il valore mediano del dominio di `X`. Questa strategia è nota come *domain splitting*.

value(Enum): in questo caso `Enum` deve individuare un predicato che si assume in grado di restringere il dominio di `X` possibilmente (ma non necessariamente) ad un singolo valore. SICStus effettuerà la valutazione di `Enum(X,Rest,BB0,BB)` dove `Rest` è la lista delle variabili ancora da etichettare privata di `X` e `BB0` e `BB` sono istanziate a due parametri il cui uso è descritto di seguito.
Il predicato individuato da `Enum` deve avere successo in modo non-deterministico, restringendo l'insieme dei valori possibili per `X`. A seguito di backtracking dovrà quindi fornire differenti restrizioni del dominio di valori per `X`. Affinché sia realizzata correttamente una strategia *branch-and-bound*, si assume che `Enum` effettuia sua la, prima volta che ha successo, la valutazione del predicato ausiliario `first_bound(BB0,BB)`; inoltre si assume che le successive valutazioni di `Enum` comporti la valutazione del predicato ausiliario `later_bound(BB0,BB)`.
`Enum` può essere un termine composto, come ad esempio `enumera(Param)`. In questo caso SICStus valuterà `enumera(Param,X,Rest,BB0,BB)`.

III gruppo: queste opzioni controllano l'ordine in cui, rispetto alla variabile `X`, sono effettuate le scelte alternative. Non hanno effetto se nel II gruppo si è scelta l'opzione `value(Enum)`.

up: il dominio è esplorato in ordine crescente. Questo è il default.

down: il dominio è esplorato in ordine decrescente.

IV gruppo: le opzioni di questo gruppo determinano se tutte le soluzioni debbano essere enumerate tramite il backtracking o se debba essere prodotta una singola soluzione (se esiste) che minimizzi (o massimizzi) il valore di `X`.

all: tutte le soluzioni vengono enumerate. Questo è il default.

minimize(X), maximize(X): si impiega un algoritmo di branch-and-bound per determinare un assegnamento che minimizzi (o massimizzi) `X`. (Il processo di labelling deve far sì che per ogni alternativa possibile venga assegnato un valore alla variabile `X`).

È possibile contare il numero di assunzioni (le scelte) effettuate durante la computazione. Ciò tramite l'opzione **assumptions(K)**. Quando una soluzione viene individuata, `K` viene istanziata al numero delle scelte effettuate.

Ad esempio, per enumerare le soluzioni usando un ordinamento statico delle variabili, si usa il goal:

```
?- constraints(Variabili), labeling([], Variabili).
```

(ove, ovviamente, in predicato `constraints` deve essere definito dal programmatore tramite opportune clausole CLP) In un uso come il precedente, il non indicare alcuna opzione equivale a scegliere il comportamento di default, ovvero: `[leftmost,step,up,all]`. Vediamo un altro esempio di uso delle opzioni: al fine di minimizzare una funzione di costo utilizzando un algoritmo branch-and-bound, un ordine dinamico delle variabili, il principio first-fail, e

sfruttando il domain splitting in modo da esplorare prima la parte superiore del dominio, si utilizza:

```
?- constraints(Variabili, Cost),
   labeling([ff,bisect,down,minimize(Cost)], Variabili).
```

2.4. Alcuni vincoli globali. Passiamo ora in rassegna alcuni dei vincoli globali forniti da SICStus ed orientati alla codifica di problemi combinatorici.

- Tramite un predicato della forma

  `element(?X,+List,?Y)`

  dove `X` e `Y` sono numeri interi o variabili vincolate (ovvero con un dominio finito di valori), e `List` è una lista di numeri interi o variabili vincolate. L'atomo è soddisfatto se l'`X`-esimo elemento di `List` è uguale a `Y`. In pratica, i domini di `X` e `Y` sono ristretti in modo che per ogni elemento del dominio di `X` esiste un opportuno elemento nel dominio di `Y`, e viceversa.

  Il predicato `element` opera in modo da mantenere la consistenza rispetto al dominio per `X` e rispetto agli intervalli per `List` e `Y`. Esso inoltre ha un comportamento deterministico. Ecco degli esempi:

  `?- element(3, [1,1,2,2,3,3,1,1], Y)`

  restituisce `Y = 2`.

  `?- element(X, [1,1,2,2,3,3,1,1], 1)`

  restituisce `X in (1..2)∨(7..8)`.

  `?- element(X, [1,1,2,2,3,3,1,1], Y)`

  restituisce `X in 1..8, Y in 1..3`, lasciando impliciti gli eventuali constraint coinvolti.
- Il predicato

  `all_different(+Variabili)`

  si impiega con `Variabili` istanziata ad una lista di variabili vincolate. Ogni variabile della lista viene vincolata ad assumere un valore (del suo dominio) distinto da tutti i valori associati alle altre variabili. In termini dichiarativi, questo corrisponde ad imporre un vincolo di diversità tra ogni coppia di variabili della lista. Vi è anche una forma con un secondo parametro `all_different(+Variabili,Opzioni)` tramite il quale è possibile influenzare il comportamento del predicato (in merito, si veda il manuale SICStus).
- Tra le possibilità offerte dal predicato

  `serialized(+Inizi,+Durate,+Opzioni)`

  menzioniamo solo il fatto che scegliendo opportunamente le opzioni possiamo modellare un problema di serializzazione di un insieme di task (le cui durate e istanti di inizio sono indicati rispettivamente in `Durate` e `Inizi`, che sono liste di interi o di variabili vincolate). Ad esempio, i seguenti constraint modellano tre task, tutti della durata pari a 5 unità di tempo. Il task 1 deve precedere il task 2 mentre il task 3 deve essere completato prima del task 2 oppure iniziare almeno 10 unità di tempo dopo l'inizio del task 2.

```
?- domain([S1,S2,S3], 0, 20),
   serialized([S1,S2,S3], [5,5,5],
              [precedences([d(2,1,sup),d(2,3,10)])]).
```

La risposta sarà:

```
S1 in 0..15
S2 in 5..20
S3 in 0..20
```

Si veda il manuale SICStus per una descrizione dettagliata del ricco insieme di opzioni valide per questo constraint.

- Constraint correlati al precedente sono:

```
cumulative(+ListaInizi,+ListaDurate,+ListaRisorse,?Limite)
cumulative(+ListaInizi,+ListaDurate,+ListaRisorse,?Limite,+Opzioni)
```

Si può pensare che il loro effetto sia quello di schedulare n task (ognuno con tempo di inizio durata, e ammontare di risorse necessarie, descritti nei primi tre parametri) in modo che le risorse globali impiegate non superino `Limite`. Più precisamente, se S_j, D_j e R_j indicano rispettivamente inizio, durata e risorse relative al task j-esimo, allora posto

$$\begin{aligned} a &= min(S_1, ..., S_n), \\ b &= max(S_1 + D_1, ..., S_n + D_n) \\ R_{ij} &= \begin{cases} R_j & \text{se } S_j \leqslant i < S_j + D_j \\ 0 & \text{altrimenti} \end{cases} \end{aligned}$$

Il constraint è soddisfatto se $R_{i1} + ... + R_{in} \leqslant$ `Limite`, per ogni $a \leqslant i < b$. L'eventuale parametro `Opzioni` influenza il comportamento del predicato.

3. Constraint reificati

Supponiamo di avere un certo numero di constraint, ad esempio cinque, e che si desideri che almeno tre di essi siano verificati nella soluzione.

Un modo per ottenere tale scopo è illustrato dal seguente schema di goal:

```
?- constraint1 #<=> B1,
   constraint2 #<=> B2,
   constraint3 #<=> B3,
   constraint4 #<=> B4,
   constraint5 #<=> B5,
   B1+B2+B3+B4+B5 #>= 3
```

I vari B_i sono visti come constraint booleani e quindi con valori possibili 0 o 1. Ogni B_i assumerà valore 1 quando il corrispondente constraint ***constraint***$_i$ è verificato. Questo modo di operare viene detto *reificazione dei constraint*.

Ecco degli esempi di goal che utilizzano la reificazione:

```
?- domain([A,B,C],1,2), (A#<B) #<=> B1, (B#<C) #<=> B2, B1+B2 #= 2.

no
```

```
?- domain([A,B,C],1,2), (A#<B) #<=> B1, (B#<C) #<=> B2, B1+B2 #= 1.

A in 1..2,
B in 1..2,
C in 1..2,
B1 in 0..1,
B2 in 0..1
yes
```

Il seguente è un modo per definire, tramite la reificazione, un predicato `occorrenze(X,L,N)` che risulta verificato quando un elemento `X` occorre esattamente `N` volte in una lista `L`.

```
occorrenze(_, [], 0).
occorrenze(X, [Y|L], N) :- (X#=Y) #<=> B,
                           N #= M+B,
                           occorrenze(X, L, M).
```

4. CLP($\mathbb{R}$) in SICStus

Per utilizzare il pacchetto dei risolutori di vincoli sui numeri reali è necessario consultare la opportuna libreria tramite la direttiva:[3]

```
:- use_module(library(clpr)).
```

Per comunicare all'interprete SICStus che un vincolo riguarda variabili reali, questo va racchiuso tra parentesi graffe. Ad esempio sottoponendo il goal:

```
?- { V = I * R, V = 2, R = 5 }.
```

otterremmo la risposta:

```
I = 0.4, R = 5.0, V = 2.0
```

Alcuni simboli predicativi di constraint ammessi sono:

```
=   <   >   =<   >=   =\=
```

Mentre alcuni tra i simboli di funzione interpretati ammessi sono:

```
+    -    *    /    abs
sin   cos   tan
pow  (oppure ^)   min   max
```

Alcuni esempi di goal:

```
?- {X = pow(2,3)}.
```

che restituisce `X = 8.0`.

```
?- {X = min(2*3, pow(1,7))}.
```

che invece restituisce `X = 1.0`.

Si noti è possibile definire il logaritmo come predicato:

```
log(B,X,Y) :- { pow(B,Y) = X}.
```

[3]Similmente si opera con i razionali CLP($\mathbb{Q}$), in tal caso si deve consultare la libreria `library(clpq)`.

Una interessante funzionalità offerta da SICStus è la `maximize(Expr)`.[4] Vediamo come opera con un esempio. Al goal:

```
?- { 2*X+Y =< 16, X+2*Y =< 11, X+3*Y =< 15, Z = 30*X+50*Y }, maximize(Z).
```

Viene risposto:

```
X = 7.0, Y = 2.0, Z = 310.0
```

ovvero viene massimizzata la funzione associata a `Z` relativamente ai vincoli di dominio imposti sui valori delle variabili in gioco.

5. CLP(*FD*) in GNU-Prolog

Il solver CLP(*FD*) di GNU-Prolog presenta alcune differenze (in parte solamente sintattiche) rispetto al solver disponibile in SICStus. Ne illustriamo qui le più rilevanti, rimandando il lettore interessato alla documentazione fornita con l'interprete. Innanzitutto non è necessario caricare alcuna libreria, il solver è subito disponibile all'avvio dell'interprete. Non tutte le funzionalità offerte da SICStus sono presenti in GNU-Prolog.

La prima osservazione da fare riguarda la rappresentazione delle variabili vincolate e dei loro domini. Vi sono due possibili rappresentazioni interne utilizzate da GNU-Prolog per i domini delle variabili. La più semplice viene utilizzata quando il dominio è un intervallo di numeri interi. Questa è la rappresentazione di default: inizialmente ad ogni variabile vincolata viene automaticamente assegnato un dominio di default costituito dai numeri interi `0..fd_max_integer`, dove `fd_max_integer` è un valore reperibile tramite il predicato `fd_max_integer/1`. Tale valore è il più grande numero intero che possa essere assegnato ad una variabile.

Tale dominio potrà venire poi ristretto per effetto della esecuzione del codice del goal che l'utente invoca, qualora questo imponga dei vincoli più restrittivi. Come conseguenza dell'imposizione di tali vincoli, alcuni valori del dominio potranno venir esclusi. Se il dominio resta un intervallo di interi (quindi un insieme di numeri contigui) allora la rappresentazione non viene mutata. Se invece nel dominio si formano "dei buchi", ovvero il dominio diventa unione di più intervalli non contigui, allora la rappresentazione interna viene mutata in una cosidetta "rappresentazione sparsa". Facciamo presente questo dettaglio implementativo per far notare che la rappresentazione sparsa permette di rappresentare domini con valori non superiori a `vector_max` (reperibile tramite `fd_vector_max/1`). Nella configurazione di default il valore di `vector_max` è molto inferiore a `fd_max_integer` (vale solitamente 127). Qualora il dominio di una variabile vincolata rappresentato con la rappresentazione sparsa contenga valori superiori a `vector_max` viene generato un messaggio di warning:

```
Warning: Vector too small - maybe lost solutions (FD Var:N).
```

Esso avverte che alcuni dei valori ammissibili del dominio potrebbero venir ignorati e conseguentemente alcune delle soluzioni potrebbero essere perse. Per ovviare a questo fenomeno, l'utente può modificare il valore di `vector_max`, tramite la direttiva

```
:- fd_set_vector_max(+NewVal).
```

[4]Per una descrizione esaustiva delle funzionalità, così come dei constraint e delle funzioni predefinite, si invita a consultare il manuale SICStus.

Questa operazione va fatta con attenzione: un valore di `NewVal` troppo basso potrebbe portare alla perdita di soluzioni, un valore troppo alto peggiorerebbe l'efficienza del risolutore. Tale direttiva va necessariamente invocata all'inizio del programma, prima di utilizzare qualsiasi constraint, quindi il programmatore deve prevedere un opportuno limite superiore ai valori massimi di tutte le variabili vincolate del programma.

La scelta di rendere la rappresentazione interna dei domini in qualche modo visibile e influenzabile dal programmatore è stata fatta per migliorare l'efficienza del solver ma ovviamente introduce anche se in modo controllato e focalizzato, un aspetto poco dichiarativo nello stile di programmazione CLP.

Alcune differenze tra le sintassi di GNU-Prolog e SICStus Prolog:

Domini: Il vincolo `fd_domain(+X,+I1,+I2)` viene utilizzato per assegnare un dominio di valori. Il termine `X` può essere una variabile o una lista di variabili.

Constraint: GNU-Prolog fornisce il constraint `fd_all_different(ListaVars)` con l'analogo significato dell'`all_different(ListaVars)` di SICStus.

Il constraint `fd_element(I, ListaInteri, Var)` vincola la variabile `Var` ad assumere lo stesso valore dell'`I`-esimo elemento della lista `ListaInteri`.

Il constraint `fd_element_var(I, Lista, Var)` è simile al precedente ma `Lista` può essere anche una lista di variabili vincolate.

Il predicato `fd_labeling(ListaVar, ListaOpzioni)` effettua il labeling. Le opzioni principali permettono di scegliere diverse strategie per la selezione delle variabili, tra esse: `variable_method(standard)` (corrisponde a `leftmost` di SICStus), `variable_method(ff)` (`first_fail`), ecc. Altre opzioni permettono di controllare la selezione dei valori. Si veda la documentazione per una loro trattazione esaustiva.

Ottimizzazione: Il goal `fd_minimize(Goal, Var)` invoca ripetutamente il goal `Goal` (solitamente viene utilizzato congiuntamente ad un predicato di labeling in `Goal`) con l'obiettivo di minimizzare il valore di `X`. Il goal `fd_maximize(Goal, Var)` è pure disponibile.

6. Esercizi

Esercizio 16.1. Si descrivano tutte le possibili risposte (se ne esistono) che CLP(FD) fornisce al goal

```
:- domain([X],1,6), domain([Y],3,10), domain([Z],5,12), X=Y, Y=Z.
```

Esercizio 16.2. Scrivere un programma CLP(FD) che definisca un predicato

```
somma(+Numero,-Tripla).
```

che se invocato con un numero naturale `Numero` come primo argomento produca in `Tripla` una lista di tre numeri naturali la cui somma sia `Numero`. Si implementi il predicato utilizzando il labeling e facendo in modo che una soluzione venga generata ad ogni richiesta di soluzione (digitando ";").

Esercizio 16.3. Si consideri il lancio di quattro dadi e siano v_1, v_2, v_3, e v_4 i quattro punteggi. Scrivere un programma CLP(FD) in cui si definisca un predicato `gioca(+S,+P,-N)`. Tale predicato, dati due numeri `S` e `P` deve avere sempre successo istanziando la variabile `N`. La variabile `N` deve venir istanziata al numero di possibili esiti del lancio dei quattro dadi tali che la somma dei 4 valori sia `S` e il prodotto dei 4 valori sia `P`.

Esercizio 16.4. Un intervistatore bussa alla porta di una casa dove è atteso da una signora. La signora gli apre e lui chiede: “Quanti figli ha?” “Ho tre figlie.” gli risponde la donna. “Di che età?” “Il prodotto delle età è 36 e la somma è uguale al numero civico di questa casa.” “Buon giorno e grazie.” L’intervistatore se ne va, ma dopo un po’ ritorna e le dice: “I dati che mi ha fornito non sono sufficienti.” La signora ci pensa un po’ e replica: “È vero, che sbadata! La figlia maggiore ha gli occhi azzurri.” Con questo dato l’intervistatore può conoscere l’età delle tre figlie. Quanti anni hanno? Scrivere un programma CLP(FD) che risolva il problema.

Esercizio 16.5. Scrivere un programma CLP(FD) che dati due numeri naturali positivi n_1 e n_2 (con $n_1 < n_2$) determini per ogni coppia di numeri x e y (con $n_1 < x < y < n_2$) se x e y siano o meno primi tra loro.

CAPITOLO 17

CLP(FD): la metodologia *constrain and generate*

La metodologia di programmazione dichiarativa *constrain and generate* rappresenta un approccio alla soluzione di problemi (prevalentemente combinatorici) alternativo al generate and test analizzato nei capitolo precedenti. L'idea base consiste nel vincolare inizialmente le variabili (assegnando dei domini ed imponendo dei constraint) e ritardare il più possibile la loro istanziazione, facendo seguire una fase di generazione delle alternative istanziazioni ammissibili. In questo capitolo considereremo alcuni esempi e problemi già affrontati con la programmazione Prolog e con ASP, al fine di cogliere la differente tecnica di programmazione che si può adottare in CLP.

In questo capitolo sfrutteremo spesso delle funzionalità offerte dal risolutore di vincoli CLP di SICStus Prolog. Molte di tali funzionalità sono disponibili anche in altre implementazioni di CLP(FD), quali GNU-Prolog o ECLiPSe, tuttavia potrebbero presentare una differente sintassi.

1. Il problema delle *N regine*

Nel Capitolo 9 (Sezione 2.1) e nel Capitolo 13 (Sezione 2) abbiamo studiato delle soluzioni al problema delle N regine. Illustriamo ora una possibile soluzione basata su CLP(FD):

```
queens(N, Queens) :- length(Queens, N),
                     domain(Queens,1,N),
                     constrain(Queens),
                     labeling([], Queens).

constrain(Queens) :- all_different(Queens),
                     diagonal(Queens).

diagonal([]).
diagonal([Q|Queens]) :- sicure(Q, 1, Queens),
                        diagonal(Queens).

sicure(_,_,[]).
sicure(X,D,[Q|Queens]) :- nonattacca(X,Q,D),
                          D1 is D+1,
                          sicure(X,D1,Queens).

nonattacca(X,Y,D) :- X + D #\= Y,
                     Y + D #\= X.
```

Questo programma illustra bene l'impiego dell'approccio constraint and generate. Infatti, consideriamo il corpo della prima clausola: intuitivamente, il primo letterale genera una lista `Queens` di `N` variabili. Il secondo letterale determina i domini delle variabili in `Queens`. Il terzo letterale impone dei vincoli sui possibili valori di ogni variabile (tramite la valutazione di `all_different` e `diagonal`). Infine la fase "generate" viene compiuta dal labeling.

2. Il problema del *knapsack*

Si consideri la seguente istanza del problema del knapsack. Lo zaino ha dimensione 9. Gli oggetti sono: il whisky che occupa 4 e vale 15, il profumo che occupa 3 e vale 10, le sigarette che occupano 2 e valgono 7. Si ricerca un'allocazione dello zaino che assicuri un profitto di almeno 30. Questa particolare istanza può essere risolta dal goal CLP:

```
?- domain([W,P,S],0,9),
   4*W + 3*P + 2*S #=< 9,
   15*W + 10*P + 7*S #>= 30.
```

A cui viene infatti risposto:

```
W in 0..2, P in 0..3, S in 0..4
```

Il che indica l'esistenza di una soluzione che rispetti i vincoli. Se tuttavia volessimo esplicitarne una, dovremmo sfruttare il labeling e sottoporre il goal:

```
?- domain([W,P,S],0,9),
   4*W + 3*P + 2*S #=< 9,
   15*W + 10*P + 7*S #>= 30,
   labeling([], [W,P,S]).
```

In questo caso (digitando ";" dopo ogni soluzione prodotta) otterremmo le seguenti soluzioni ammissibili:

```
P = 1, S = 3, W = 0 ?  ;
P = 3, S = 0, W = 0 ?  ;
P = 1, S = 1, W = 1 ?  ;
P = 0, S = 0, W = 2 ?  ;
no
```

Una utile opportunità offerta da CLP(*FD*) consiste nella possibilità di determinare una soluzione soddisfacente i vincoli e tale da massimizzare il valore di una espressione. Il goal seguente sfrutta il labeling a tale scopo:

```
?- L #= 15*W + 10*P + 7*S,
   maximize((domain([W,P,S],0,9), 4*W+3*P+2*S #=< 9, labeling([],[W,P,S])),
            L).
```

Esso genera la soluzione

```
L=32, P=1, S=1, W=1.
```

Il constraint impiegato nel goal precedente ha la forma generica

```
maximize(+Goal,?X)
```

La sua valutazione impiega un algoritmo branch-and-bound per ricercare un assegnamento che massimizzi il valore della variabile `X`. Il parametro `Goal` deve essere un legittimo goal che vincoli `X` (e, come visto, può coinvolgere `labeling`). Tale goal viene ripetutamente invocato dall'interprete in corrispondenza di vincoli sempre più restrittivi per i valori di `X` fino a che un ottimo viene individuato. (Esiste ovviamente anche il constraint duale `minimize(Goal,X)`).

Esercizio 17.1. Si scriva un programma CLP(FD) per risolvere il problema del massimo taglio in un grafo (non pesato).

Diamo ora una soluzione al problema decisionale del knapsack generalizzato già affrontato alla fine della Sezione 9 del Capitolo 13.

In questo caso rappresentiamo i pesi e i valori dei n tipi diversi di oggetti tramite due liste di n interi: Per esempio, le due liste del fatto

```
oggetti([2,4, 8,16,32,64,128,256,512,1024],
        [2,5,11,23,47,95,191,383,767,1535]).
```

rappresentano l'esistenza di 10 tipo di oggetti diversi.

Ecco un possibile programma CLP(FD) che risolve il problema:

```
zaino(CapacitaMax,ValoreMin) :-
                           oggetti(Pesi,Costi),
                           length(Pesi,N),
                           length(Vars,N),
                           domain(Vars,0,CapacitaMax),
                           scalar_product(Pesi,Vars,#=<,CapacitaMax),
                           scalar_product(Costi,Vars,#>=,ValoreMin),
                           labeling([ff],Vars).
```

Alcune osservazioni: abbiamo utilizzato la lista di variabili `Vars` per vincolare il numero di copie di ogni oggetto che vengono inserite nello zaino. Inoltre è stato utilizzato il predicato built-in (di SICStus) `scalar_product/4` per imporre il vincolo che la somma di tutti i pesi degli oggetti nello zaino non ecceda la capacità dello stesso. Similmente, sempre tramite `scalar_product/4`, viene imposto il vincolo che la somma di tutti i valori non sia inferiore al valore minimo richiesto.

In generale, il vincolo `scalar_product(+Vect1, +Vect2, +RelOp, ?Val)` impone che il prodotto scalare tra i due vettori `Vect1` e `Vect2` sia in relazione `RelOp` con il valore `Val`. (Si noti che in SICStus Prolog è disponibile anche un predicato `knapsack/3` che corrisponde a utilizzare il vincolo di uguaglianza come terzo argomento di `scalar_product`.)

3. Il problema del *map coloring*

Il problema del map coloring può essere risolto in CLP come segue:

```
clpcoloring(Mapname,N,Archi) :- map(Mapname,Nodi,Archi),
                                domain(Nodi,1,N),
                                constrain(Archi),
                                labeling([],Nodi).

constrain([[A,B]|R]) :- A #\= B,
                        constrain(R).
constrain([]).
```

Si osservi che la descrizione del grafo viene ottenuta tramite il predicato

```
map(Mapname,Nodi,Archi).
```

Si assume quindi che vi siano nel programma delle clausole che definiscono tale predicato e che per ogni possibile "nome di grafo" `Mapname`, il goal `map(Mapname,Nodi,Archi)` abbia successo istanziando le variabili `Nodi` ed `Archi` rispettivamente alle liste dei nodi e degli archi di un grafo. Ad esempio potrebbe esserci un fatto del tipo

```
map(miografo, [A1,A2,A3,A4], [[A1,A2],[A1,A3],[A4,A2],[A4,A3]]).
```

oppure un predicato più complesso come nell'Esercizio 17.2. Si noti anche che nel programma precedente si assume che i nodi del grafo siano variabili non istanziate e conseguentemente gli archi siano rappresentati come coppie di variabili. Questa risulta una comoda rappresentazione (invece di usare atomi Prolog, come verrebbe più naturale, in prima battuta), tenendo presente che la colorazione viene calcolata applicando la tecnica constraint-and-generate. Nella fase constraint infatti non avrebbe senso imporre dei vincoli su variabili già istanziate.

Esercizio 17.2. Si generi automaticamente una cartina di dimensioni ragguardevoli (si veda qui sotto un suggerimento) e si confrontino computazionalmente le soluzioni e l'efficienza dei programmi Prolog, CLP, e ASP per il coloring.

```
map(march9,Nodi,Archi) :- Livelli = 10000,
                          N is 3*Livelli,
                          length(Nodi,N),
                          aggiungi_archi(Livelli,Nodi,Archi).

aggiungi_archi(1,[A,B,C],[[A,B],[B,C]]).
aggiungi_archi(N,[A,B,C,A1,B1,C1|R],
               [[A,A1],[A,B],
               [B,B1],[B,A1],[B,C],
               [C,C1],[C,B1]|E]) :- N>1,
                                    M is N-1,
                                    aggiungi_archi(M,[A1,B1,C1|R],E).
```

4. Il *marriage problem*

Si consideri la istanza del marriage problem, già affrontata nel Capitolo 13:

```
likes(andrea,diana).
likes(andrea,federica).
likes(bruno,diana).
likes(bruno,elena).
likes(bruno,federica).
likes(carlo,elena).
likes(carlo,federica).
```

La seguente è una possibile formulazione in CLP(FD) che non comporta l'impiego dei precedenti fatti. Si osservi come le costanti `diana`, `elena` e `federica` siano rappresentate dagli interi `1`, `2` e `3`.

```
?- Sposi = [Andrea,Bruno,Carlo],
   domain(Sposi,1,3),
   all_different(Sposi),
   Andrea #\= 2,
   Carlo #\= 1.
```

La risposta fornita è

```
Sposi = [Andrea,Bruno,Carlo],
Andrea in{1}\/{3},
Bruno in 1..3,
Carlo in 2..3
```

A questo punto basta aggiungere l'ulteriore letterale `labeling([],Sposi)` per ottenere la generazione esplicita di tutte le risposte:

```
Carlo = 3, Bruno = 2, Sposi = [1,2,3], Andrea = 1 ?  ;
Carlo = 2, Bruno = 3, Sposi = [1,3,2], Andrea = 1 ?  ;
Carlo = 2, Bruno = 1, Sposi = [3,1,2], Andrea = 3 ?  ;
no
```

5. SEND + MORE = MONEY

Un noto problema di *constraint satisfaction (CSP)* è il cosiddetto "Send More Money puzzle". Il problema richiede di sostituire ad ogni lettera una cifra (a lettere uguali devono corrispondere cifre uguali) in modo che sia corretto scrivere la somma SEND + MORE = MONEY.

Il programma presentato di seguito sfrutta ancora una volta il tipico approccio adottato nella programmazione logica con vincoli:

(1) dichiarazione dei domini delle variabili;
(2) imposizione di vincoli;
(3) ricerca di una soluzione ammissibile via backtracking (oppure ricerca della soluzione ottima tramite branch-and-bound.

A volte è utile inserire un ulteriore passo tra il secondo e il terzo: l'imposizione di uno o più vincoli ausiliari allo scopo di rompere la simmetria dello spazio delle soluzioni. Si pensi ad esempio al problema del map coloring: permutando i colori di una soluzione si ottengono altre soluzioni isomorfe alla prima. Introdurre un vincolo che elimini parte delle simmetrie

(ad esempio fissando a priori il colore di uno dei nodi) permette solitamente di ottenere maggior efficienza.

Nella soluzione del Send More Money puzzle utilizzeremo appunto un *symmetry breaking constraint*. I domini sono determinati dal predicato `domain/3` e dall'imporre che `S` e `M` siano maggiori di 0. I vincoli sono imposti tramite l'equazione codificata dal predicato `sum/8` unitamente al constraint `all_different`. Infine il backtracking viene attivato tramite il predicato `labeling`.

Si noti che differenti strategie possono essere impostate agendo sul parametro `Type`.

```
smmpuzzle([S,E,N,D,M,O,R,Y], Type) :- domain([S,E,N,D,M,O,R,Y],0,9),
                                      S#>0, M#>0,
                                      all_different([S,E,N,D,M,O,R,Y]),
                                      sum(S,E,N,D,M,O,R,Y),
                                      labeling(Type,[S,E,N,D,M,O,R,Y]).

sum(S,E,N,D,M,O,R,Y) :- 1000*S + 100*E + 10*N + D +
                        1000*M + 100*O + 10*R + E #=
                        10000*M + 1000*O + 100*N + 10*E + Y.
```

Nel seguente goal si opta per la strategia di default per il labeling (ovvero si seleziona la variabile più a sinistra e si procede per valori crescenti):

```
?- smmpuzzle([S,E,N,D,M,O,R,Y], []).
```

Ecco la soluzione:

```
D = 7, E = 5, M = 1, N = 6, O = 0, R = 8, S = 9, Y = 2
```

Esercizio 17.3. Si risolva il Send More Money puzzle utilizzando diverse scelte delle opzioni per il labeling e si confrontino i risultati e l'efficienza ottenuti.

6. Uso del predicato `cumulative`

Come abbiamo accennato, `cumulative` di SICStus Prolog è il costrutto di base per risolvere problemi di scheduling. Vediamo di seguito un semplice esempio di questo genere di problemi.

Supponiamo vi siano 4 differenti lavori che richiedono la stessa risorsa. Il primo lavoro dura 3 unità di tempo e usa 2 unità di risorsa. Il secondo dura 2 e usa 3 unità, il terzo dura 1 e usa 2. Il quarto dura 2 e usa 3 unità. Si vuole determinare una allocazione dei lavori. Ricerchiamo una soluzione tramite il goal:

```
?- domain([S1,S2,S3,S4],1,20),
   cumulative([S1,S2,S3,S4],[3,2,1,2],[2,3,2,3],5),
   labeling([],[S1,S2,S3,S4]).
```

Otteniamo la risposta

```
S1=1, S2=1, S3=3, S4=4
```

Un secondo esempio di impiego del `cumulative` può essere fornito dal seguente problema di allocazione per una variante della battaglia navale in cui le navi hanno forma rettangolare.

Iniziamo col descrivere le dimensioni delle navi:

- Portaerei: larghezza 2, lunghezza 4.
- Corazzata: larghezza 1, lunghezza 4.
- Incrociatore: larghezza 1, lunghezza 3.
- Cacciamine: larghezza 1, lunghezza 1.

Si desidera disporre una flotta che consta di una portaerei, due corazzate, tre incrociatori e due cacciamine in modo tale che questa possa attraversare, restando in formazione, un canale largo 3.

Ecco un goal CLP che calcola la soluzione:

```
?- domain([P1,C1,C2,I1,I2,I3,D1,D2],1,20),
   cumulative([P1,C1,C2,I1,I2,I3,D1,D2],
              [4,4,4,3,3,3,1,1],
              [2,1,1,1,1,1,1,1],3),
   labeling([], [P1,C1,C2,I1,I2,I3,D1,D2]).
```

L'interprete genera la risposta:

```
C1=1, C2=5, D1=8, D2=9, I1=5, I2=5, I3=8, P1=1
```

Tuttavia tale schieramento può essere ulteriormente compattato aggiungendo questi ulteriori vincoli al goal sopra riportato:

```
?- P1+4 #< A, C1+4 #< A, C2+4 #< A,
   I1+3 #< A, I2+3 #< A, I3+3 #< A,
   D1+1 #< A, D2+1 #< A, A = 11
```

Così facendo si ottiene la nuova soluzione:

```
A=11, C1=5, C2=5, D1=9, D2=9, I1=1, I2=4, I3=7, P1=1
```

7. Il problema della *allocazione di compiti*

Riprendiamo l'istanza del problema della allocazione di compiti già illustrata nel Capitolo 13:

	p_1	p_2	p_3	p_4
w_1	7	1	3	4
w_2	8	2	5	1
w_3	4	3	7	2
w_4	3	1	6	3

Cerchiamo di massimizzare il profitto totale utilizzando un programma CLP. Una prima soluzione sfrutta una rappresentazione esplicita della matrice all'interno del goal:

```
?- Matrix = [B11, B12, B13, B14,
             B21, B22, B23, B24,
             B31, B32, B33, B34,
             B41, B42, B43, B44],
   domain(Matrix,0,1),
   B11 + B12 + B13 + B14 #= 1,
   B21 + B22 + B23 + B24 #= 1,
   B31 + B32 + B33 + B34 #= 1,
   B41 + B42 + B43 + B44 #= 1,
   B11 + B21 + B31 + B41 #= 1,
   B12 + B22 + B32 + B42 #= 1,
   B13 + B23 + B33 + B43 #= 1,
   B14 + B24 + B34 + B44 #= 1,
   Profitto #= 7*B11 + 1*B12 + 3*B13 + 4*B14 +
               8*B21 + 2*B22 + 5*B23 + 1*B24 +
               4*B31 + 3*B32 + 7*B33 + 2*B34 +
               3*B41 + 1*B42 + 6*B43 + 3*B44,
   Profitto #>= 19,
   labeling([], Matrix).
```

Esercizio 17.4. Per evitare l'impiego di un goal complesso come il precedente, si scriva un programma CLP in cui viene definito un insieme di predicati ausiliari atti a gestire il calcolo della somma dei valori di una riga, della somma dei valori di una colonna, ecc..

La seguente è una soluzione alternativa che impiega `element` e permette di scrivere un goal meno complesso.

```
?- domain([W1,W2,W3,W4],1,4),
   all_different([W1,W2,W3,W4]),
   element(W1, [7,1,3,4], WP1),
   element(W2, [8,2,5,1], WP2),
   element(W3, [4,3,7,2], WP3),
   element(W4, [3,1,6,3], WP4),
   Profitto #= WP1 + WP2 + WP3 + WP4,
   Profitto #>= 19,
   labeling([], [W1,W2,W3,W4]).
```

Esercizio 17.5. Si utilizzi la l'opzione `assumptions(K)` del labeling per verificare quale dei due approcci giunga con meno tentativi alla soluzione. Cercare di giustificare l'eventuale diverso comportamento delle due soluzioni.

8. Il problema del *circuito hamiltoniano*

In questa sezione riprendiamo il problema del circuito hamiltoniano già affrontato nel Capitolo 13. Scriveremo un programma CLP(*FD*) in grado di determinare se un grafo sia o meno hamiltoniano. La rappresentazione che adotteremo prevede di denotare un grafo con un atomo Prolog del tipo:

```
grafo(Nodi, Archi)
```

dove `Nodi` è la lista ordinata dei primi N interi che rappresentano N nodi del grafo, mentre `Archi` è una lista di coppie di nodi, come ad esempio in:

```
grafo([1,2,3],[[1,2],[1,3],[2,3]]).
```

Si noti che è importante scegliere di rappresentare gli N nodi con i primi N interi per poter utilizzare correttamente il constraint `circuit/1`, predefinito di SICStus. Il constraint `circuit(Lista)` presuppone che `Lista` sia una lista di N interi o di variabili vincolate. Esso impone il vincolo che l'elemento i-esimo di tale lista identifichi il nodo successore al nodo i in un circuito hamiltoniano. In altre parole, `circuit([X1,...,Xn])`, con `X1,...,Xn` variabili vincolate, impone che gli archi ⟨1,`X1`⟩, ⟨2,`X2`⟩, ..., ⟨n,`Xn`⟩ formino un circuito hamiltoniano.

Ecco il programma CLP(FD):

```
hamiltoniano(Cammino) :- grafo(Nodi, Archi),
                         length(Nodi, N),
                         length(Cammino, N),
                         domain(Cammino, 1, N),
                         make_domains(Cammino, 1, Archi, N),
                         circuit(Cammino),
                         labeling([ff], Cammino).

make_domains([], _, _, _).
make_domains([X|Y], Nodo, Archi, N) :- findall(Z, member([Nodo,Z], Archi),
                                                Successori),
                                       reduce_domains(N, Successori, X),
                                       Nodo1 is Nodo+1,
                                       make_domains(Y, Nodo1, Archi, N).

reduce_domains(0, _, _) :- !.
reduce_domains(N, Successori, Var) :- N>0,
                                      member(N, Successori), !,
                                      N1 is N-1,
                                      reduce_domains(N1, Successori, Var).
reduce_domains(N, Successori, Var) :- Var #\= N,
                                      N1 is N-1,
                                      reduce_domains(N1, Successori, Var).
```

Si osservi come tramite il predicato `make_domains` venga vincolata ogni variabile X_i ad assumere valori che identifichino uno dei successori del nodo i nel grafo.

9. Il problema dei *numeri di Schur*

Per risolvere in CLP(FD) il problema (decisionale) dei numeri di Schur affrontato nel Capitolo 13 introduciamo una lista di variabili vincolate `[B1,...,BN]` con dominio $1,\dots,P$. In questo modo il valore della i-esima variabile `Bi` indicherà in quale delle P partizioni andrà inserito il numero i. Ecco il codice:

```
schur(N,P) :- length(Lista,N),
              domain(Lista,1,P),
              vincola(Lista,N),
              labeling([ff],Lista).

vincola(Lista, N) :- Lista=[1,2|_],
                     ricorsivamente(Lista,1,1,N).
```

Il predicato `vincola/2` viene definito tramite il predicato `ricorsivamente/4` di seguito riportato. Quest'ultimo impone la condizione che per ogni tripla di interi `I`, `J`, `K`, tali che `I+J=K`, qualora il blocco `BI` di `I` coincida con il blocco `BJ` di `J` allora `K` sia assegnato ad blocco `BK` diverso da `BI`:

```
ricorsivamente(_,I,_,N) :- I>N, !.
ricorsivamente(Lista,I,J,N) :- I+J>N, !,
                               I1 is I+1,
                               ricorsivamente(Lista,I1,1,N).
ricorsivamente(Lista,I,J,N) :- I>J, !,
                               J1 is J+1,
                               ricorsivamente(Lista,I,J1,N).
ricorsivamente(Lista,I,J,N) :- K is I+J, J1 is J+1,
                               nth(I,Lista,BI),
                               nth(J,Lista,BJ),
                               nth(K,Lista,BK),
                               (BI #= BJ) #=> (BK #\= BI),
                               ricorsivamente(Lista,I,J1,N).
```

Tramite `vincola/2` inoltre si è imposta la condizione che `B1 = 1` e `B2 = 2`. Questo riduce alcune delle simmetrie del problema aumentando l'efficienza. Si veda in merito anche l'Esercizio 13.10.

10. Esercizi

Esercizio 17.6. Si testi il programma delle N regine riportato nella Sezione 1, cercando di determinare il massimo valore di N per cui il risolutore CLP(FD) riesce a risolvere il problema in tempi ragionevoli. Si effettuino poi le stesse prove utilizzando diverse opzioni (ad esempio `ff` (first-fail), `leftmost`, ...) per il labeling.

Esercizio 17.7. Si confronti l'efficienza della soluzione al problema delle N regine presentata in Sezione 1 con le corrispondenti soluzioni illustrate nei Capitoli 9 e 13. Per esempio si utilizzino le stesse istanze usate nell'esercizio Esercizio 17.6 (ed eventualmente altre) al fine di comprendere quale sia la soluzione più efficiente. Qualora si rilevino notevoli differenze tra i comportamenti delle tre soluzioni, si cerchi di ipotizzarne le ragioni.

Esercizio 17.8. Si confronti criticamente la soluzione del problema dello zaino presentata in questo capitolo con quella presentata nella Sezione 9 del Capitolo 13.

Esercizio 17.9. Provare a sperimentare diverse strategie di labeling nel programma CLP(FD) che risolve il problema dei numeri di Schur. Ad esempio si utilizzi la strategia `leftmost`, o `ffc`, invece di `ff`. Vi sono differenze dal punto di vista dell'efficienza con cui il programma risolve il problema?

CAPITOLO 18

Concurrent constraint programming

1. Concurrent Constraint (Logic) Programming

I modelli concorrenti a memoria condivisa tradizionali sono basati su un supporto di memoria in cui delle variabili $X_1, \ldots, X_n$ vengono memorizzate. In ogni istante, ogni variabile X_i ha un fissato valore in $\mathcal{D}_i$. I vari processi possono leggere il valore di una variabile ($\texttt{read}(X_i)$) o aggiornare tale valore ($\texttt{write}(X_i)$) [**Dij76**].

Saraswat [**Sar00, SRP01**] invece propone un modello in cui lo spazio condiviso, detto *store* contiene un vincolo σ su $\mathcal{D}_1 \times \cdots \times \mathcal{D}_n$. Le variabili pertanto *possono* avere un valore preciso, ma più in generale, hanno loro associato un insieme di valori ammissibili. Ciò permette di rappresentare e condividere conoscenza incompleta. Tale modello ha dato vita al *Concurrent Constraint Programming (CCP)*. Le azioni elementari in questo caso sono di due tipi:

$\texttt{tell}(c)$**:** Si verifica la consistenza del vincolo $\sigma \wedge c$.
- Se è consistente, allora $\sigma := \sigma \wedge c$.
- Se è inconsistente, allora l'azione fallisce.

$\texttt{ask}(c)$**:** In questo caso, si verifica se c, oppure $\neg c$ sono conseguenze logiche di σ.
- Se $\sigma \Rightarrow c$, allora l'azione ha successo.
- Se $\sigma \Rightarrow \neg c$, allora l'azione ha fallimento.
- Altrimenti il processo che ha richiesto l'azione rimane *bloccato* in attesa che uno dei due casi sopra avvenga per effetto della modifica dello store da parte di altri processi.

ESEMPIO 18.1. Si consideri la seguente sequenza di applicazioni di `ask` e `tell`:

(1) $\texttt{tell}(X > 0)$ implica che $\sigma = X > 0$
(2) $\texttt{ask}(X < 0)$ fallisce, in quanto $X > 0 \Rightarrow \neg(X < 0)$.
(3) $\texttt{ask}(X > -1)$ ha successo, in quanto $X > 0 \Rightarrow X > -1$.
(4) $\texttt{ask}(X > 1)$ rimane bloccato.
(5) $\texttt{tell}(X \neq 1)$. Il nuovo vincolo $X \neq 1$ è consistente con lo store σ. Quest'ultimo viene aggiornato a: $\sigma = X > 0 \wedge X \neq 1 = X > 1$. In questo momento il processo relativo all'azione 4 riparte.

Si osservi che ogni sequenza di `ask` e `tell` può essere eseguita in qualsiasi ordine senza cambiare l'effetto globale (anche se il tempo in cui i vari processi sono bloccati può cambiare). Questa proprietà viene detta di stabilità ed è garantita dalla consistenza globale dello store, e dalla sospensione degli `ask` di vincoli per cui non vi sia ancora informazione sufficiente.

Nell'esempio appena visto abbiamo usato variabili che possiamo immaginare sui numeri interi (o reali) e simboli di predicato ($>$, $\neq$) a cui abbiamo dato l'usuale interpretazione. Quando si descrive un linguaggio CCP, bisogna al solito essere più precisi in questo punto.

Dato un linguaggio del primo ordine $\langle \Pi, \Sigma, \mathcal{V} \rangle$ dobbiamo fissare una interpretazione $\mathcal{A} = \langle A, I \rangle$ dove A è un insieme non vuoto detto dominio, che è anche il dominio D_i di ogni variabile X_i in gioco (in realtà questa apparente restrizione può essere risolta in un modello *multi-sorted* nel quale ad ogni variabile può essere attribuito un dominio diverso) e I è una funzione di interpretazione di simboli di Σ e Π su A.

A questo punto possiamo fornire una semantica più precisa delle operazioni di `ask` e `tell`: sia dato uno store σ:

- se $\mathcal{A} \models \vec{\exists}(\sigma \wedge c)$ allora `tell`(c) ha successo e $\sigma := \sigma \wedge c$.
- Altrimenti `tell`(c) fallisce e σ rimane inalterato.
- Se $\mathcal{A} \models \vec{\forall}(\sigma \rightarrow c)$ allora `ask`(c) ha successo.
- Se $\mathcal{A} \models \vec{\forall}(\sigma \rightarrow \neg c)$ allora `ask`(c) fallisce.
- Altrimenti il processo che ha richiesto l'azione `ask`(c) rimane *bloccato*.

Ove al solito con $\vec{\exists}$ e $\vec{\forall}$ si intende la chiusura esistenziale ed universale, rispettivamente.

La metodologia suddetta viene ben inglobata nei linguaggi logici con vincoli. Sfortunatamente non vi è disponibile una implementazione universalmente diffusa. La cosa che più si avvicina è il linguaggio di cui parleremo nella prossima sezione.

2. Linda

Linda [**CG89**] è un linguaggio per il coordinamento di processi su uno spazio condiviso, detto spazio delle tuple. Vi è un processo *server* che funge da gestore di tale spazio condiviso. Usando SICStus Prolog, su una shell di comando di Prolog, si consulta un file del tipo seguente:

```
:-use_module(library('linda/server')).

principale(H,P) :-
   write('nome Host: '),write(H),nl,
   write('nome Port: '),write(P),nl.

:-linda([(H:P)-principale(H,P)]).
```

Sul video appariranno i dati del PC su cui il server gira, tipicamente un nome per `Host` (poniamo `Wineandroses`) e un numero per `Port` (poniamo 2078).

Su questo spazio delle tuple possono agire diversi processi `client` in modo concorrente usando le seguenti primitive principali:

out(+Tuple): Mette la tupla (un termine) nello spazio delle tuple.

in(?Tuple): Rimuove la tupla dallo spazio delle tuple se ce n'è almeno una che unifichi con il termine passato come parametro. Se non ce n'è nessuna, allora l'esecuzione si blocca finché qualche processo non ne inserisce una.

in_noblock(?Tuple): Rimuove la tupla dallo spazio delle tuple se ce n'è almeno una che unifichi con il termine passato come parametro. Se non ce n'è nessuna, allora l'esecuzione fallisce (non si blocca).

rd(?Tuple): Ha successo se c'è almeno una tupla dallo spazio delle tuple che unifica con il termine passato come parametro. Se non ce n'è nessuna, allora l'esecuzione si blocca finché qualche processo non ne inserisce una.

rd_noblock(?Tuple): Ha successo se c'è almeno una tupla dallo spazio delle tuple che unifica con il termine passato come parametro. Se non ce n'è nessuna, allora l'esecuzione fallisce (non si blocca).

Ad esempio il seguente processo `producer`, dopo essere stato consultato, in una diversa shell di Prolog, sullo stesso o su un diverso computer, in seguito all'esecuzione del goal `:- produci(10).` aggiungerà le tuple: `p(1), p(2),...,p(10)`.

```
:-use_module(library('linda/client')).
produci(0) :-
  write('fatto'),nl.
produci(X) :-
   X > 0,
   out(p(X)),
   write('messo: '),write(X),nl,
   Y is X - 1,
   produci(Y).
:- linda_client('Wineandroses':2023).
```

Il seguente processo, che può essere duplicato in varie shell, operando pertanto in modo parallelo, in seguito all'esecuzione del goal `:- molti.` sceglie a caso due tuple tra quelle nello spazio, le preleva e mette la tupla con il prodotto dei valori scelti. Se vi è una sola tupla, si limita a stamparla e a rimetterla nello spazio. Quale sarà l'ultima tupla?

```
:-use_module(library('linda/client')).

molti :-
   in(p(X)),
   (in_noblock(p(Y)),
   Z is X * Y,
   out(p(Z)),
   write('messo: '),write(Z),nl,
   molti;
   write('risultato: '),write(X),nl,
   out(p(X))).

:- linda_client('Wineandroses':2078).
```

Il package mette a disposizione altre primitive di lettura/scrittura multipla e delle direttive per fissare dei tempi massimi (timeout) di attesa da un input bloccante. Si invita il lettore ad approfondire tali tematiche sul manuale di SICStus Prolog.

Il package Linda può essere usato assieme alle librerie di vincoli (quali ad esempio il `clpfd`). In tal modo è possibile usare una metodologia di programmazione logica concorrente, con vincoli locali ai singoli processi, ma che possono modificarsi in seguito ad informazioni (puntuali) introdotte da altri processi. Il maggior limite di questa libreria sono gli alti tempi di comunicazione tra processi che lo rendono uno strumento di prototipazione di software concorrente su memoria condivisa più che un vero strumento per la programmazione concorrente. Tuttavia può essere visto come un comodo strumento per la parallelizzazione

su diversi PC di algoritmi di ricerca di soluzioni (sia con la tecnica generate and test che constraint and generate), dove le comunicazioni sono infinitesime rispetto ai tempi di ricerca.

APPENDICE A

Ordini, reticoli e punti fissi

In questa appendice riportiamo brevemente alcuni concetti ausiliari relativi a ordini, reticoli e punti fissi.

Definizione A.1. Una relazione $\prec$ su un insieme A è un *ordine parziale* su A se è riflessiva, antisimmetrica e transitiva.

Definizione A.2. Una relazione d'ordine $\prec \subseteq A \times A$ è un *buon ordine* quando non esistono infiniti elementi $a_1, a_2, \dots \in A$ tali che $\dots \prec a_4 \prec a_3 \prec a_2 \prec a_1$ (ovvero non esiste alcuna catena discendente infinita).

Se nella precedente definizione invece di considerare una relazione d'ordine ci si riferisse ad una qualsiasi relazione binaria allora si parlerebbe di relazione *ben fondata* (in luogo di buon ordine).

Esempio A.1. Se $\prec$ è l'usuale relazione di minore sugli interi, allora $\langle \mathbb{N}, \prec \rangle$ è un buon ordine. Invece $\langle \mathbb{Z}, \prec \rangle$ non è un buon ordine.

La seguente definizione stabilisce come sia possibile combinare due buoni ordini tramite il prodotto cartesiano. Questo permette di combinare buoni ordini su domini elementari per ottenere buoni ordini su domini più complessi.

Definizione A.3. Se $\langle A, \prec_A \rangle$ e $\langle B, \prec_B \rangle$ sono ordini, allora l'ordine lessicografico ottenuto dai due è l'ordine $\langle A \times B, \prec \rangle$ definito come:

$$\langle X, Y \rangle \prec \langle X', Y' \rangle \quad \text{se e solo se} \quad \begin{array}{l} X \prec_A X' \vee \\ X =_A X' \wedge Y \prec_B Y' \end{array}$$

L'idea si può facilmente estendere a terne, quaterne e cosí via. Intuitivamente, l'ordine lessicografico altro non è che l'ordinamento del vocabolario: si confronta prima il primo carattere, poi il secondo, il terzo, e cosí via.

Sussiste la seguente proprietà:

Proposizione A.1. *Se $\langle A, \prec_A \rangle$ e $\langle B, \prec_B \rangle$ sono buoni ordini, allora l'ordine lessicografico ottenuto dai due è un buon ordine.*

Dim. Esercizio. □

Si può intuitivamente dimostrare che l'ordine lessicografico è un buon ordine immaginando lo spazio generato dagli insiemi A e B, rappresentati su assi cartesiani. Preso un punto a caso nello spazio di coordinate $\langle X, Y \rangle$ con $X \in A$ e $Y \in B$ si nota che tutti i punti minori di questo sono tutti quelli con ascissa minore di X (che sono infiniti) e quelli con ordinata minore di Y (che invece sono finiti). Si potrebbe pensare che il fatto che esistano infiniti punti minori di $\langle X, Y \rangle$ sia in contraddizione con la definizione di buon ordine. In realtà si

vede facilmente che, se $\langle A, \prec_A \rangle$ e $\langle B, \prec_B \rangle$ sono buoni ordini, allora non si può costruire una catena discendente infinita per $A \times B$. Questa infatti indurrebbe una catena discendente infinita per uno tra A o B, contraddicendo l'ipotesi.

Definizione A.4. Sia $\leq$ un ordine parziale su un insieme A e sia $X \subseteq A$. Allora

- Un elemento $a \in A$ è *upper bound* di X se per ogni $x \in X$ vale $x \leq a$.
- Un elemento $a \in A$ è *lower bound* di X se per ogni $x \in X$ vale $a \leq x$.
- Un elemento $a \in A$ è il *least upper bound* di X (e si scrive $lub(X)$) se a è upper bound per X e per ogni altro upper bound a' di X vale che $a \leq a'$.
- Un elemento $a \in A$ è il *greatest lower bound* di X (e si scrive $glb(X)$) se a è lower bound per X e per ogni altro lower bound a' di X vale che $a' \leq a$.

Esercizio A.1. Dimostrare che se esiste un lub di X allora esso è unico. Dimostrare che se esiste un glb di X allora esso è unico.

Definizione A.5. Sia $\langle A, \leq \rangle$ un ordine parziale. Allora $\langle A, \leq \rangle$ è un *reticolo completo* se $lub(X)$ e $glb(X)$ esistono in A per ogni sottoinsieme X di A. In particolare i due elementi $lub(A)$ e $glb(A)$ si dicono rispettivamente *top* ($\top$) e *bottom* ($\bot$) di A.

Esempio A.2. Per ogni insieme I il suo insieme potenza (l'insieme delle parti) 2^I, ordinato tramite la relazione di inclusione, è un reticolo completo. Dato un sottoinsieme $X \subseteq 2^I$, si ha che $lub(X) = \bigcup X$ e $glb(X) = \bigcap X$. I due elementi $lub(2^I)$ e $glb(2^I)$ sono rispettivamente I e $\emptyset$.

Definizione A.6. Sia $\langle A, \leq \rangle$ un reticolo completo. Sia inoltre $F : A \rightarrow A$ una funzione. Allora diremo che F è *monotona* se vale che $x \leq y \ \rightarrow \ F(x) \leq F(y)$.

Definizione A.7. Sia $\langle A, \leq \rangle$ un reticolo completo. Sia inoltre $X \subseteq A$. Allora X è *diretto* se ogni sottoinsieme finito di X ha un upper bound in X.

Definizione A.8. Sia $\langle A, \leq \rangle$ un reticolo completo. Sia inoltre $F : A \rightarrow A$ una funzione. Allora diremo che F è *continua* se vale che $F(lub(X)) \leq lub(F(X))$ per ogni sottoinsieme X di A.

Definizione A.9. Sia $\langle A, \leq \rangle$ un reticolo completo. Sia inoltre $F : A \rightarrow A$ una funzione. Allora diremo che $a \in A$ è *minimo punto fisso (lfp)* di F se è punto fisso di F (ovvero $F(a) = a$) e per ogni altro punto fisso a' vale che $a \leq a'$.

Teorema A.1 (Tarski). *Sia $\langle A, \leq \rangle$ un reticolo completo. Sia inoltre $F : A \rightarrow A$ una funzione monotona. Allora F ha un minimo punto fisso. Inoltre*

$$lfb(F) = glb\{x \, : \, F(x) = x\} = glb\{x \, : \, F(x) \leq x\}.$$

Definizione A.10. Sia $\langle A, \leq \rangle$ un insieme parzialmente ordinato. Sia $F : A \rightarrow A$ un operatore. Allora definiamo

$$\begin{cases} F \uparrow 0(I) & = & I \\ F \uparrow (n+1)(I) & = & F(F \uparrow n(I)) \\ F \uparrow \omega(I) & = & \bigcup_{i \geq 0} F \uparrow i(I) \end{cases}$$

Se $I = \bot$ allora $F \uparrow \alpha(I)$ si denota semplicemente con $F \uparrow \alpha$.

Dato un operatore F, si può anche definire la sua iterazione *all'ingiù* nel modo seguente:

DEFINIZIONE A.11. Sia $\langle A, \leq \rangle$ un insieme parzialmente ordinato. Sia $F : A \longrightarrow A$ un operatore. Definiamo:

$$\begin{cases} F \downarrow 0(I) & = & I \\ F \downarrow (n+1)(I) & = & F(F \downarrow n(I)) \\ F \downarrow \omega(I) & = & \bigwedge_{i \geq 0} F \downarrow i(I) \end{cases}$$

Se $I = \top$, allora $F \downarrow \alpha(I)$ si denota semplicemente con $F \downarrow \alpha$.

Vale il seguente risultato.

TEOREMA A.2 (Tarski). *Sia F operatore continuo su un reticolo completo. Allora $F \uparrow \omega$ esiste ed è minimo punto fisso di F.*

APPENDICE B

Spigolature sull'uso di Prolog e degli ASP-solver

1. Prolog e CLP

1.1. Reperimento di un interprete Prolog. Diverse sono le implementazioni di Prolog disponibili. Alcune commerciali altre no. Tra le alternative, menzioniamo solamente alcuni Prolog che includano anche le funzionalità del CLP(FD):

SICStus Prolog: Commerciale. Disponibile per diverse piattaforme. Informazioni e documentazione sono reperibili al sito `http://www.sics.se/sicstus`.

GNU-Prolog: Open source. Sviluppato da Daniel Diaz e disponibile sia per linux che per windows. Informazioni, documentazione e distribuzioni sono reperibili al sito `http://pauillac.inria.fr/~diaz/gnu-prolog`.

ECLiPSe: Gratuito per scopi non commerciali. Sviluppato all'Imperial College di Londra. Disponibile per diverse piattaforme. Informazioni, documentazione e distribuzioni sono reperibili al sito `http://eclipse.crosscoreop.com/eclipse`.

B-Prolog: Open source. Disponibile per diverse piattaforme. Informazioni, documentazione e distribuzioni sono reperibili al sito `http://www.probp.com`.

Si noti che mentre la sintassi base e le funzionalità del Prolog standard sono in generale supportate da ognuna di queste implementazioni, possono esserci differenze (solitamente sintattiche, ma anche semantiche) relativamente sia alle varie estensioni "non standard" del linguaggio, sia alle funzionalità relative al CLP. Ad esempio non tutti i sistemi sopra elencati forniscono gli stessi domini di vincoli e constraint solver. Vi sono anche notevoli differenze relativamente ad altre funzionalità quali l'integrazione con altri linguaggi di programmazione, disponibilità di ambienti grafici di sviluppo e debugging, ecc.

1.2. Invocazione di un goal con simbolo predicativo non definito. Alcuni interpreti Prolog, (questo, ad esempio, è il caso sia di SICStus Prolog che di GNU-prolog) nella configurazione *default*, generano un errore quando si cerca di dimostrare un atomo costruito con un simbolo predicativo non definito da alcuna clausola del programma (cioé mai presente come testa di ancuna regola del programma).

Solitamente è utile modificare questo comportamento dell'interprete. Ciò è possibile invocando opportuni comandi o direttive. Ad esempio:

- In SICStus Prolog è necessario impartire il comando:
  ```
  prolog_flag(unknown, error, fail).
  ```
  direttamente come goal o alternativamente inserirlo come *direttiva* (ovvero anticipato dall'operatore `:-`) all'inizio del file contenente il programma stesso.
- In GNU-prolog il goal da utilizzare è invece:
  ```
  set_prolog_flag(unknown, fail).
  ```

1.3. Predicati `dynamic`. Tramite la direttiva

```
:- dynamic(DescrizioneDiPredicato)
```

dove `DescrizioneDiPredicato` è della forma `simboloPred/numerointero`, è possibile dichiarare che il predicato descritto da `DescrizioneDiPredicato` è "dinamico" ovvero le clausole che lo definiscono possono essere modificate, rimosse o aggiunte durante l'esecuzione.

Solitamente, negli interpreti che prevedono sia la possibilità di compilare che di consultare definizioni di clausole, i predicati dinamici vengono comunque interpretati. Si consulti la documentazione dello specifico Prolog utilizzato per determinare se questo sia il caso.

Una altra particolarità che spesso si riscontra, ad esempio in SICStus Prolog (ma non in GNU-Prolog), è che solamente i predicati dinamici possono essere oggetto della "meta-variable facility" (cfr. Sezione 12 del Capitolo 7).

1.4. Accesso alle "librerie". Molte implementazioni di Prolog offrono un insieme nutrito di librerie di predicati predefiniti e raccolti in modo omogeneo. Spesso prima di scrivere delle definizioni per dei predicati può essere utile consultare la documentazione associata al Prolog che si sta utilizzando per verificare se il predicato desiderato sia già presente in una delle librerie fornite con l'interprete.

In SICStus Prolog il comando per importare una librerie è

```
:- use_module(library(clpfd)).
```

Ciò causa il caricamento la libreria relativa a CLP(FD). L'analoga direttiva di ECLiPSe è:

```
:- lib(ic).
```

Nel caso di SWI-Prolog le librerie si caricano similmente a come avviene in SICStus Prolog, anche se il nome delle librerie potrebbe non essere lo stesso. Ad esempio, la direttiva:

```
:- use_module(library('clp/bounds')).
```

carica il solver CLP(FD).

1.5. Statistiche di esecuzione di un goal. Molte implementazioni di Prolog offrono delle funzionalità utilizzabili per ottenere informazioni sul sistema e sull'esecuzione dei goal. Ad esempio in SICStus Prolog tramite il predicato `statistics/2` possono essere ricavate informazioni sull'uso della memoria, tempi di esecuzione, garbage collection, ecc. (Questa funzionalità è stata ad esempio utilizzata nella Sezione 2.1.1 del Capitolo 9.)

Lo stesso predicato è anche disponibile in GNU-Prolog e in ECLiPSe. Un altro predicato usualmente disponibile è `profile` (SWI-Prolog). Per maggiori informazioni su questo genere di predicati si rimanda alla documentazione associata allo specifico Prolog utilizzato.

1.6. Tracing. Molti Prolog offrono la possibilità di seguire passo passo l'esecuzione di un goal. Solitamente ciò avviene attivando la modalità di esecuzione detta *tracing*, invocando preventivamente il goal

```
:- trace.
```

In questa modalità di esecuzione l'interprete mostra i passi base dell'esecuzione di un goal seguendo la costruzione dell'SLD-albero, permette di ispezionare le istanziazioni delle variabili, impostare degli *spy-point*, ecc. Solitamente la modalità si disattiva con la direttiva

```
:- notrace.
```

2. ASP-solver

2.1. Reperimento di un ASP-solver. Sono diversi gli ASP-solver disponibili. Quasi tutti sono open source e sviluppati in ambiente accademico.

Tra le possibilità menzioniamo alcuni ASP-solvers che meglio si adattano alla trattazione sviluppata in questo testo:

Lparse+Smodels: Sviluppato presso la Helsinki University of Technology. Distribuzioni per linux e windowsXP. Sia il front-end `lparse` che il vero e proprio solver `smodels` sono disponibili presso `http://www.tcs.hut.fi/Software/smodels`.

Lparse+Cmodels: Sviluppato da Yuliya Lierler e da altri ricercatori [**LM04, Lie05**]. La distribuzione disponibile è per linux e include un certo numero di SAT-solver che vengono impiegati "dietro le quinte" dall'ASP-solver. Utilizza il front-end `lparse`. È reperibile al sito `http://www.cs.utexas.edu/users/tag/cmodels`.

DLV: Il solver DLV è sviluppato presso la TU Wien ed è disponibile unitamente alla relativa documentazione, al sito `http://www.dbai.tuwien.ac.at/proj/dlv`. Rappresenta una implementazione della programmazione logica disgiuntiva, offre quindi un linguaggio più espressivo di quello trattato da `smodels`/`cmodels`. Si veda la Sezione 7.1 del Capitolo 12.

Si noti che sia `smodels` che `cmodels` utilizzano `lparse` come pre-processore per effettuare la fase di grounding (si veda la Sezione 4 del Capitolo 12), e quindi adottano una sintassi comune. Altri ASP-solver adottano sintassi differenti.

Tra gli altri ASP-solver, sviluppati a livello più o meno sperimentale, resi disponibili da diversi istituti di ricerca, ne citiamo alcuni unitamente ai rispettivi riferimenti web:

ASSAT: `http://assat.cs.ust.hk`
CCalc: `http://www.cs.utexas.edu/users/tag/cc`
DeReS: `http://www.cs.engr.uky.edu/ai/deres.html`
noMoRe: `http://www.cs.uni-potsdam.de/~linke/nomore`

APPENDICE C

Soluzioni degli esercizi

1. Esercizi dal Capitolo 2

Esercizio 2.7:

(1) $\theta = [V/g(Z), W/f(Y), X/f(Y)]$;
(2) non esiste;
(3) $\theta = [X_1/f(Y), X_2/W, X_3/g(Z)]$;
(4) $\theta = [W/g(Y), X/h(a, g(Y)), Z/a]$;
(5) $\theta = [X/h(g(a)), Y/g(a), Z/a]$;
(6) non esiste se ci si limita a considerare termini finiti;
(7) $\theta = [X/g(b, b), Y/b]$;
(8) $\theta = [Y/g(X, a), Z/g(X, a)]$;
(9) $\theta = [W/f(a), X/g(f(a), Y), Z/g(g(f(a), Y), Y)]$;
(10) non esiste;
(11) $\theta = [Z/f(a, Y)]$;
(12) non esiste;
(13) $\theta = [W/f(a, Y), Z/f(a, Y)]$;
(14) $\theta = [X/b, Y/b, Z/a]$;
(15) non esiste;
(16) $\theta = [X/b]$;
(17) $\theta = [Z/f(a, X)]$;
(18) $\theta = [X/h(Y, b), Z/f(a, h(Y, b))]$.

Esercizio 2.8: La sostituzione ottenuta è: $\rho = \big[X/f(b), Z/Y\big]$.

Esercizio 2.9: La sostituzione ottenuta è:

$$\rho = \big[A/f(B), B/b, C/b, D/B\big].$$

Quindi si ha $t\rho = h(f(f(B)), g(b), B)$.

2. Esercizi dal Capitolo 4

Esercizio 4.3: Le soluzioni dell'Esercizio 2.7 riportate a pagina 291 sono m.g.u.

Esercizio 4.5: Soluzione: $L_1 = p(a, X)$, $L_2 = p(Y, b)$, $L_3 = p(Z, Z)$. Infatti: $\sigma_{1,2} = [X/b, Y/a]$, $\sigma_{2,3} = [Y/b, Z/b]$, e $\sigma_{1,3} = [X/a, Z/a]$.

3. Esercizi dal Capitolo 6

Esercizio 6.5: Si tratta di un insieme di clausole definite, quindi il modello minimo si può determinare applicando l'operatore di conseguenza immediata fino al raggiungimento del minimo punto fisso. Il modello così determinato è:

$$M_P = \{q(b),\ \ q(f(b)),\ \ q(f(f(b))),\ \ q(f(f(f(b)))),\ \ \ldots\}.$$

Il modello massimo è la base di Herbrand

$$\begin{array}{rcl} \mathcal{B}_P & = & \{\ q(b),\ \ q(f(b)),\ \ q(f(f(b))),\ \ \ldots \\ & & \ \ q(a),\ \ q(f(a)),\ \ q(f(f(a))),\ \ \ldots \\ & & \ \ p(b),\ \ p(f(b)),\ \ p(f(f(b))),\ \ \ldots \\ & & \ \ p(a),\ \ p(f(a)),\ \ p(f(f(a))),\ \ \ldots\}. \end{array}$$

Un terzo modello è, ad esempio:

$$\begin{array}{l} \{\ q(b),\ \ q(f(b)),\ \ q(f(f(b))),\ \ \ldots \\ \ \ p(f(f(f(a)))),\ \ p(f(f(f(f(a))))),\ \ p(f(f(f(f(f(a)))))),\ \ \ldots\}. \end{array}$$

Esercizio 6.6: Il programma seguente risponde al primo quesito dell'esercizio:

$$\begin{array}{rcl} p(f(X)) & \leftarrow & p(X). \\ q(a). & & \\ q(b). & & \\ q(f(f(b))). & & \end{array}$$

La risposta è giustificata dato che il modello minimo del precedente programma è

$$\{q(a),\ \ q(b),\ \ q(f(f(b)))\}.$$

Il programma seguente risponde al secondo quesito:

$$\begin{array}{rcl} p(f(X)) & \leftarrow & p(X). \\ q(a). & & \\ p(f(f(f(f(a))))). & & \end{array}$$

La risposta è giustificata perchè il modello minimo del precedente programma è

$$\{q(a),\ \ p(f(f(f(f(a))))),\ \ p(f(f(f(f(f(a)))))),\ \ p(f(f(f(f(f(f(a))))))),\ \ \ldots\},$$

quindi ogni possibile modello deve essere infinito.

4. Esercizi dal Capitolo 7

Esercizio 7.1: Una possibile soluzione:

```
permutazione([],[]).
permutazione([X|Xs],Ys1) :- permutazione(Xs,Ys),
                            select(X,Ys1,Ys).

select(X,[X|Xs],Xs).
select(X,[Y|Ys],[Y|Zs]) :- select(X,Ys,Zs).
```

Esercizio 7.10: Una possibile soluzione:

```
cammino(X,Y,Path) :- cammino_aux(X,Y,[X],RevPath),
                     reverse(RevPath,Path).

cammino_aux(X,Y,Visti,[Y|Visti]) :- arco(X,Y),
                                    nonmember(Y,Visti).
cammino_aux(X,Y,Visti,RevP) :- arco(X,Z),
                               nonmember(Z,Visti),
                               cammino_aux(Z,Y,[Z|Visti],RevP).
```

Esercizio 7.11: Una possibile soluzione:

```
prof(X,0) :- var(X),!.
prof(X,1) :- atomic(X),!.
prof(T,N) :- T =.. [_|Args],
             proflist(Args, 0, N1),
             N is N1+1.

proflist([],N,N).
proflist([A|R],I,O) :- prof(A,Na),
                       max(I,Na,M),
                       proflist(R, M, O).
```

Esercizio 7.12: Una possibile soluzione:

```
ciclo(X) :- path(X, X).

path(X,Y) :- arco(X,Y).
path(X,Y) :- arco(X,Z), path(Z,Y).
```

Esercizio 7.13: Una possibile soluzione:

```
penultimo([X,Y],X).
penultimo([X|R],P) :- penultimo(R,P).
```

Esercizio 7.14: Una semplice soluzione che sfrutta il fatto che la lista è di sole costanti è la seguente:

```
palindroma(L) :- reverse(L,L).
```

Una soluzione per il secondo quesito dell'esercizio richiede uno sforzo leggermente maggiore. Eccola:

```
palindromaVar(L) :- reverse(L,R), listSimile(L,R).

listSimile([],[]).
listSimile([A|As],[B|Bs]) :- simile(A,B), listSimile(As,Bs).

simile(X,Y) :- var(X), var(Y).
simile(X,Y) :- atomic(X), atomic(Y), X == Y.
simile(X,Y) :- compound(X), compound(Y),
               X =.. [F|As], Y =.. [F|Bs],
               listSimile(As,Bs).
```

Esercizio 7.15: Una possibile soluzione:

```
espandi([],[]).
espandi([[1,T]|R],[T|Res]) :- espandi(R,Res).
espandi([[N,T]|R],[T|Res]) :- N>1, N1 is N-1,
                              espandi([[N1,T]|R],Res).
```

Esercizio 7.16: Una possibile soluzione:

```
mymember(A,[B|_]) :- A==B,!.
mymember(A,[_|B]) :- mymember(A,B).
```

Esercizio 7.17: Una possibile soluzione è la seguente:

```
ounion([],L,OL) :- norep(L,OL).
ounion([A|R],[],OL) :- norep([A|R],OL).
ounion([A|R],[A|S],OL) :- ounion(R,[A|S],OL).
ounion([A|R],[B|S],OL) :- A < B, ounion(R,[A,B|S],OL).
ounion([A|R],[B,B|S],OL) :- B < A, ounion([A|R],[B|S],OL).
ounion([A|R],[B,C|S],[B|OL]) :- B < A, B < C, ounion([A|R],[C|S],OL).

norep([],[]).
norep([A],[A]).
norep([A,A|R],OL) :- norep([A|R],OL).
norep([A,B|R],[A|OL]) :- A \== B, norep([B|R],OL).
```

Una soluzione alternativa è:

```
ounion(L1,L2,OL) :- ounionAux(L1,L2,L), norepbis(L,OL).

ounionAux([],L,L).
ounionAux([A|R],[],[A|R]).
ounionAux([A|R],[A|S],[A|OL]) :- ounionAux(R,S,OL).
ounionAux([A|R],[B|S],[A|OL]) :- A < B, ounionAux(R,[B|S],OL).
ounionAux([A|R],[B|S],[B|OL]) :- B < A, ounionAux([A|R],S,OL).

norepbis([],[]).
norepbis([A],[A]).
norepbis([A,B|R],OL) :- A == B, norepbis([B|R],OL).
norepbis([A,B|R],[A|OL]) :- A \== B, norepbis([B|R],OL).
```

Esercizio 7.18: Una possibile soluzione è la seguente:

```
ointer([],_L,[]).
ointer([_A|_R],[],[]).
ointer([A,A|R],S,OL) :- !,ointer([A|R],S,OL).
ointer(R,[A,A|S],OL) :- !,ointer(R,[A|S],OL).
ointer([A|R],[B|S],OL) :- A < B, ointer(R,[B|S],OL).
ointer([A|R],[B|S],OL) :- B < A, ointer([A|R],S,OL).
ointer([A|R],[A|S],[A|OL]) :- ointer(R,S,OL).
```

Una soluzione alternativa è:

```
ointer(R,S,OL) :- ointerAux(R,S,L), norepbis(L,OL).

ointerAux([],_L,[]).
ointerAux([_A|_R],[],[]).
ointerAux([A|R],[B|S],OL) :- A < B, ointerAux(R,[B|S],OL).
ointerAux([A|R],[B|S],OL) :- B < A, ointerAux([A|R],S,OL).
ointerAux([A|R],[A|S],[A|OL]) :- ointerAux(R,S,OL).
```

Esercizio 7.19: Una possibile soluzione è la seguente:

```
osimdif([],L,OL) :- norep(L,OL).
osimdif([A|R],[],OL) :- norep([A|R],OL).
osimdif([A,A|R],S,OL) :- !,osimdif([A|R],S,OL).
osimdif(R,[A,A|S],OL) :- !,osimdif(R,[A|S],OL).
osimdif([A|R],[B|S],[A|OL]) :- A < B, osimdif(R,[B|S],OL).
osimdif([A|R],[B|S],[B|OL]) :- B < A, osimdif([A|R],S,OL).
osimdif([A|R],[A|S],OL) :- osimdif(R,S,OL).
```

Esercizio 7.20: Una possibile soluzione è la seguente:

```
forma(Term, a) :- var(Term).
forma(Term, a) :- atomic(Term).
forma(Term, F) :- compound(Term),
                  Term =.. [_|Args],
                  formaArgs(Args,Args1),
                  F =.. [f|Args1].

formaArgs([],[]).
formaArgs([T|Ts],[F|Fs]) :- forma(T,F), formaArgs(Ts,Fs).
```

Esercizio 7.21: Una possibile soluzione è la seguente:

```
alberello(N):- integer(N), N>1, !, B is N-1, punta(B,1), tronco(N).
alberello(_).

punta(0,N):- N2 is 2*N-1, nwrite('O',N2), nl.
punta(B,N):- B>0, nwrite(' ',B), N2 is 2*N-1,
             nwrite('O',N2), nl, B1 is B-1, N1 is N+1,
             punta(B1,N1).

tronco(N) :- B is N-2, nwrite(' ',B), write('I I'), nl.

nwrite(_Char,0).
nwrite(Char,N) :- write(Char), N1 is N-1, nwrite(Char,N1).
```

Esercizio 7.22: Una possibile soluzione è la seguente:

```
diamante(N):- integer(N), N mod 2 =:= 1, !,
              R is N//2, alto(R,1), basso(1,R).
diamante(_).

alto(0,N) :- N2 is 2*N-1, nwrite('O',N2), nl.
alto(B,N):- B>0, nwrite(' ',B), N2 is 2*N-1,
            nwrite('O',N2), nl, B1 is B-1, N1 is N+1,
            alto(B1,N1).

basso(_N,0).
basso(B,N):- N>0, nwrite(' ',B), N2 is 2*N-1,
             nwrite('O',N2), nl, B1 is B+1, N1 is N-1,
             basso(B1,N1).

nwrite(_Char,0).
nwrite(Char,N) :- write(Char), N1 is N - 1, nwrite(Char,N1).
```

Esercizio 7.23: Vi sono diverse soluzioni possibili. Una consiste nel rappresentare un albero come

`tree(Nodo,ListaSottoalberi).`

In questa rappresentazione, una foglia viene rappresentata come un albero senza sottoalberi: `tree(Nodo,[])`. Un esempio di albero con 7 nodi:

`tree(a,[tree(b,[tree(e,[]),tree(f,[])]),tree(c,[]),tree(d,[tree(g,[])])]).`

Ecco un predicato che implementa la visita in post-ordine, assumendo questa rappresentazione, scrivendo tramite `write` le etichette dei nodi:

```
visitaFigli([]).
visitaFigli([F|Figli]) :- visita(F),
                          visitaFigli(Figli).

visita(tree(N,Figli)) :- visitaFigli(Figli),
                         write(N), write(' ').
```

(Si osservi che un albero è un particolare tipo di grafo, quindi le rappresentazioni studiate per i grafi costituiscono soluzioni adeguate.)

Esercizio 7.24: Una semplice rappresentazione di un polinomio di grado n si può ottenere considerando la lista dei coefficienti a_i, per $i = n, n-1, \ldots, 1, 0$. Quindi, ad esempio, il polinomio di grado 5:

$$p(x) \quad \equiv \quad 4x^5 + 3x^3 - 6x^2 + 3x + 7$$

può essere rappresentato dal termine `poli([4,0,3,-6,3,7])`.

Adottando questa rappresentazione, il seguente è un predicato che risolve la seconda parte dell'esercizio:

```
ha_radici_intere(Poli,I,_J) :- valuta(Poli,I,0).
ha_radici_intere(Poli,I,J) :- I<J, I1 is I+1,
                              ha_radici_intere(Poli,I1,J).

valuta(poli([A|As]),ValX,Risultato) :- valutaAux([A|As],ValX,0,Risultato).

valutaAux([A0],ValX,Parz,Risultato) :- Risultato is Parz*ValX+A0.
valutaAux([Ai|As],ValX,Parz,Risultato) :- Temp is Parz*ValX+Ai,
                                          valutaAux(As,ValX,Temp,Risultato).
```

5. Esercizi dal Capitolo 8

Esercizio 8.3: È sufficiente considerare, dato n la singola clausola

$$\texttt{p :- a}_1\texttt{, a}_2\texttt{,}\ldots\texttt{,a}_k$$

Il modello minimo di questo programma è $\emptyset$, quindi ogni sottoinsieme della base di Herbrand è un modello. Dato che la cardinalità della base di Herbrand è $k+1$, esistono $2^{(k+1)}$ modelli distinti. L'esercizio è risolto se si sceglie k tale che $n \leq 2^{(k+1)}$.

Esercizio 8.4: Possiamo ottenere un programma che abbia più di un modello minimale utilizzando la negazione. Dato n, un programma costituito dalla singola clausola (non definita):

$$\texttt{p :- not a}_1\texttt{, not a}_2\texttt{,...,not a}_{n-1}$$

rappresenta la soluzione dell'esercizio. Infatti tale programma ha i seguenti n modelli minimali: {p}, $\{\texttt{a}_1\}$, $\{\texttt{a}_2\}$, $\ldots$, $\{\texttt{a}_{n-1}\}$.

6. Esercizi dal Capitolo 9

Esercizio 9.12: Una semplice soluzione è:

```
mcd(X,0,X) :- X > 0.
mcd(X,Y,G) :- Y > 0, Z is X mod Y, mcd(Y,Z,G).
```

Esercizio 9.13: Utilizzando il predicato `mcd/3` definito nell'Esercizio 9.12, possiamo dare la seguente semplice soluzione:

```
primi_tra_loro(X,Y) :- mcd(X,Y,1).
```

Esercizio 9.14: Una possibile soluzione è:

```
setaccio(Lista1, Lista2) :- setaccio_aux(Lista1, Lista2, 1).

setaccio_aux([], [], _).
setaccio_aux([N|Resto1], [N|Resto2], Pos) :-
                              Pos== N, P1 is Pos + 1,
                              setaccio_aux(Resto1, Resto2, P1).
setaccio_aux([N|Resto1], Lista2, Pos) :-
                              Pos \== N, P1 is Pos + 1,
                              setaccio_aux(Resto1, Lista2, P1).
```

Esercizio 9.15: Prendendo ad esempio la soluzione dell'Esercizio 9.14, possiamo facilmente scrivere il seguente programma:

```
rev_setaccio(Lista1, Lista2) :- length(Lista1,N),
                                rev_setaccio_aux(N, Lista1, Lista2).

rev_setaccio_aux([], [], _).
rev_setaccio_aux([N|Resto1], [N|Resto2], Pos) :-
                              Pos == N, P1 is Pos - 1,
                              rev_setaccio_aux(Resto1, Resto2, P1).
rev_setaccio_aux([N|Resto1], Lista2, Pos) :-
                              Pos \== N, P1 is Pos - 1,
                              rev_setaccio_aux(Resto1, Lista2, P1).
```

Esercizio 9.16: Ecco una soluzione che utilizza il predicato `reverse/2` per invertire una lista:

```
specchio(X,M) :- compound(X),
                 X =..  [F|Args],
                 specchio_list(Args,ArgsM),
                 reverse(ArgsM,RevArgsM),
                 M =..[F|RevArgsM].
specchio(X,X) :- atomic(X).
specchio(X,X) :- var(X).

specchio_list([],[]).
specchio_list([A|As],[R|Rs]) :- specchio(A,R),
                                specchio_list(As,Rs).
```

Esercizio 9.17: Utilizzando il predicato `member/2` si può risolvere il problema come segue:

```
unico(L,E) :- member(E,L), conta(E,L,1).

conta(E,[],0).
conta(E,[E|R],N) :- !, conta(E,R,N1), N is N1+1.
conta(E,[X|R],N) :- X\==E, conta(E,R,N).
```

Esercizio 9.18: Ecco una possibile soluzione:

```
fresh(X,X) :- atomic(X),!.
fresh(X,_) :- var(X),!.
fresh(T,NewT) :- T =..  [F|Args],
                 freshlist(Args, NewArgs),
                 NewT =..  [F|NewArgs].

freshlist([],[]).
freshlist([T|R],[NewT|NewR]) :- fresh(T,NewT),
                                freshlist(R,NewR).
```

Esercizio 9.20: La soluzione desiderata è:

```
select_n([Elemento|Resto],1,Elemento,Resto).
select_n([E1|Resto1],N,Elemento,[E1|Resto2]) :-
                              N > 1, N1 is N - 1,
                              select_n(Resto1,N1,Elemento,Resto2).
```

In cui si potrebbe eventualmente aggiungere un controllo che il terzo argomento sia istanziato ad un numero positivo.

7. Esercizi dal Capitolo 11

Esercizio 11.3: Riconosce tutte le stringhe `(1+1+1+1+...+1)` per un qualsiasi numero di 1 (maggiore di 0) tranne le due stringhe `(1+1)` e `(1+1+1)`.

Esercizio 11.5: Una possibile soluzione:

```
numero --> segno, nums | nums.
segno --> ['+'] | ['-'].
nums --> nonzero, cifre | ['0'].
nonzero --> ['1']|['2']|['3']|['4']|['5']|['6']|['7']|['8']|['9'].
cifra --> ['0'] | nonzero.
cifre --> [] | cifra, cifre.
```

Esercizio 11.6: Una possibile soluzione:

```
numero(N) --> segno, nums(N) | nums(N).
segno --> ['+'] | ['-'].
nums(1) --> ['0'].
nums(N) --> nonzero, cifre(N1), {N is N1 + 1}.
nonzero --> ['1']|['2']|['3']|['4']|['5']|['6']|['7']|['8']|['9'].
cifra --> ['0'] | nonzero.
cifre(0) --> [].
cifre(N) --> cifra, cifre(N1), {N is N1 + 1}.
```

8. Esercizi dal Capitolo 12

Esercizio 12.5: Soluzione esempio: il programma

```
a1 :- not b1.
b1 :- not a1.
 ⋮       ⋮
ak :- not bk.
bk :- not ak.
```

ha 2^k modelli stabili, si scelga k opportuno.

Esercizio 12.6: Soluzione: solo {b, c}.

Esercizio 12.7: Soluzione: NO, l'unico modello stabile è {q}

Esercizio 12.8: Soluzione: solo {a, d}.

Esercizio 12.9: Soluzione: solo {-a}.

Esercizio 12.10: Il seguente programma risolve l'esercizio:

```
persona(a; b).
tipo(onesto; bugiardo).
1 { persona_ha_tipo(P, T) : tipo(T) } 1 :- persona(P).
dice_il_vero :- persona_ha_tipo(a, bugiardo),
                persona_ha_tipo(b, bugiardo).
:- persona_ha_tipo(a, onesto), not dice_il_vero.
:- persona_ha_tipo(a, bugiardo), dice_il_vero.
```

Esercizio 12.11: Il seguente programma risolve l'esercizio:

```
persona(a; b; c).
tipo(onesto; bugiardo).
1 { persona_ha_tipo(P, T) : tipo(T) } 1 :- persona(P).
dice_il_vero(a) :- persona_ha_tipo(b, onesto),
                   persona_ha_tipo(c, onesto).
dice_il_vero(b) :- persona_ha_tipo(a, bugiardo),
                   persona_ha_tipo(c, onesto).
:- persona_ha_tipo(P, onesto), not dice_il_vero(P), persona(P).
:- persona_ha_tipo(P, bugiardo), dice_il_vero(P), persona(P).
```

Esercizio 12.12: Ecco una possibile soluzione dell'enigma:

```
tipo(bugiardo;sincero).
sesso(maschio;femmina).
pianeta(marte;venere).
alieno(ork;bog).

1{ha_sesso(X,S) : sesso(S) }1 :- alieno(X).
1{ha_pianeta(X,P) : pianeta(P) }1 :- alieno(X).
1{ha_tipo(X,T) : tipo(T) }1 :- alieno(X).

ha_tipo(X,sincero) :- alieno(X), ha_sesso(X,maschio),
                      ha_pianeta(X,marte).
ha_tipo(X,bugiardo) :- alieno(X), ha_sesso(X,femmina),
                      ha_pianeta(X,marte).
ha_tipo(X,bugiardo) :- alieno(X), ha_sesso(X,maschio),
                      ha_pianeta(X,venere).
ha_tipo(X,sincero) :- alieno(X), ha_sesso(X,femmina),
                      ha_pianeta(X,venere).

ha_pianeta(bog,venere) :- ha_tipo(ork,sincero).
ha_tipo(ork,sincero) :- ha_pianeta(bog,venere).
ha_sesso(bog,maschio) :- ha_tipo(ork,sincero).
ha_tipo(ork,sincero) :- ha_sesso(bog,maschio).

ha_pianeta(ork,marte) :- ha_tipo(bog,sincero).
ha_tipo(bog,sincero) :- ha_pianeta(ork,marte).
ha_sesso(ork,femmina) :- ha_tipo(bog,sincero).
ha_tipo(bog,sincero) :- ha_sesso(ork,femmina).
```

9. Esercizi dal Capitolo 13

Esercizio 13.12: Una semplice soluzione è rappresentata dal seguente programma:

```
und_arco(X,Y):- nodo(X;Y), arco(X,Y).
und_arco(X,Y):- nodo(X;Y), arco(Y,X).
raggiungibile(X,Y) :- nodo(X;Y), und_arco(X,Y).
raggiungibile(X,Y) :- nodo(X;Y;Z), und_arco(X,Z),
                      raggiungibile(Z,Y).
:- nodo(X;Y), X!=Y, not raggiungibile(X,Y).
```

Questo programma non ha modelli stabili se il grafo non è connesso.

Esercizio 13.14: Il seguente programma assume che il grafo sia descritto da fatti della forma `nodo(1)`, `nodo(2)`, `nodo(3)`, ecc. e `arco(2,3)`, `arco(3,1)`, ecc.

```
0{mark(X)}1 :- nodo(X).

inCut(X,Y) :- arco(X,Y), mark(X), not mark(Y).
inCut(X,Y) :- arco(X,Y), mark(Y), not mark(X).

:- inCut(X,Y), mark(X;Y), nodo(X;Y).
:- inCut(X,Y), not mark(X), not mark(Y), nodo(X;Y).

maximize {inCut(X,Y):arco(X,Y), inCut(X,Y):arco(Y,X)}.
```

Esercizio 13.17: È sufficiente sostituire alla ultima regole del programma la seguente regola:

```
en { energy_pair(I1,I2) :  prot(I1,h):  prot(I2,h) }.
```

10. Esercizi dal Capitolo 16

Nei seguenti frammenti di codice CLP si assume di adottare la sintassi accettata da SICStus Prolog. Analoghi programmi per GNU-Prolog possono essere ottenuti riferendosi alla Sezione 5 del Capitolo 16.

Esercizio 16.1: La risposta generata da SICStus Prolog è:

```
?- domain([X],1,6), domain([Y],3,10), domain([Z],5,12), X=Y, Y=Z.
Y = X,
Z = X,
X in 5..6 ?  ;
no
```

Esercizio 16.2: Ecco una semplice soluzione al quesito posto:

```
somma(Numero,[A,B,C]) :-
        integer(Numero),
        domain([A,B,C],0,Numero),
        A+B+C #= Numero,
        labeling([],[A,B,C]).
```

Esercizio 16.3: Ecco una possibile soluzione:

```
tiro_vincolato(S,P,Dadi) :-
            Dadi = [D1,D2,D3,D4],
            domain(Dadi, 1, 6),
            D1+D2+D3+D4 #= S,
            D1*D2*D3*D4 #= P,
            labeling([],Dadi).

gioca(S,P,N) :- setof(Dadi,tiro_vincolato(S,P,Dadi),L),
                length(L,N),!.
gioca(_S,_P,0).
```

Si noti che la seconda clausola del predicato `gioca` garantisce che qualora non vi fossero soluzioni al goal `tiro_vincolato(S,P,Dadi)` (in tal caso `setof` fallisce), `N` venga istanziato a `0` e `gioca` abbia sempre successo. Si noti che il CUT nella prima clausola di `gioca` è rosso. Se si rilassa il requisito che il predicato `gioca` debba avere successo anche quando `tiro_vincolato(S,P,Dadi)` non ha soluzioni, allora sia il CUT che la seconda clausola di `gioca` possono essere eliminati.

Esercizio 16.4: Una possibile soluzione dell'enigma può essere calcolata dal seguente programma CLP(FD).

```
dueRisposte(EtaA,EtaB,EtaC,Civico) :-
            domain([EtaA,EtaB,EtaC],0,36),
            EtaA*EtaB*EtaC #= 36,
            EtaA #=< EtaB,
            EtaB #=< EtaC,
            labeling([],[EtaA,EtaB,EtaC]),
            Civico is EtaA+EtaB+EtaC.
treRisposte(EtaA,EtaB,EtaC,Civico):-
            dueRisposte(EtaA,EtaB,EtaC,Civico),
            EtaB #< EtaC.

nonUnaSolaSoluzione(Civico) :-
            setof((EtaA,EtaB,EtaC),
                  dueRisposte(EtaA,EtaB,EtaC,Civico),
                  L),
            length(L,N),
            N>1.

treFiglie([EtaA,EtaB,EtaC]) :-
            nonUnaSolaSoluzione(Civico),
            treRisposte(EtaA,EtaB,EtaC,Civico).
```

In questo programma, la prima clausola codifica l'informazione fornita dalle prime due risposte date dalla signora. La seconda clausola codifica invece l'informazione fornita dalle tre risposte. Il predicato principale `treFiglie` fornirà una soluzione solamente se l'informazione ricavabile dalle tre risposte consente di stabilire le tre età, mentre dall'informazione

fornita dalle sole due risposte è possibile ottenere più di una possibile tripla di età. Questa ultima condizione viene imposta tramite il predicato `nonUnaSolaSoluzione`.

11. Esercizi dal Capitolo 17

Esercizio 17.1: Per risolvere questo esercizio assumiamo di rappresentare il grafo tramite: una lista di variabili per rappresentare i nodi e una lista di coppie di variabili per rappresentare gli archi. Così facendo possiamo risolvere l'esercizio con il seguente programma:

```
maxCut(Nodi, Archi) :- domain(Nodi, 0, 1),
                       maximize((labeling([],Nodi),peso(Archi,P)),P).

peso([],0).
peso([[X,Y]|As], P) :- X == Y, peso(As,P).
peso([[X,Y]|As], P1) :- X \== Y, peso(As,P), P1 is P+1.
```

Bibliografia

[Apt97] Krzysztof R. Apt. *From Logic Programming to Prolog*. International Series in Computer Science. Prentice Hall, 1997.

[Apt03] Krzysztof R. Apt. *Principles of Constraint Programming*. Cambridge, 2003.

[Bar04] Chitta Baral. *Knowledge representation, reasoning and declarative problem solving*. Cambridge University Press, 2004.

[BS98] Franz Baader and Klaus U. Schulz. Unification theory. In W. Bibel and P.H. Schmidt, editors, *Automated Deduction — A Basis for Applications, Vol. I: Foundations — Calculi and Methods*, volume 8 of *Applied Logic Series*, pages 225–263. Kluwer Academic Publishers, 1998.

[CB01] Peter Clote and Rolf Backofen. *Computational Molecular Biology: An Introduction*. John Wiley & Sons, 2001.

[CG89] Nicholas Carriero and David Gelernter. Linda in context. *Communications of the ACM*, 32(4):444–458, 1989.

[CL73] Chin-Liang Chang and Richard C.-T. Lee. *Symbolic Logic and Mechanical Theorem Proving*. Academic Press, 1973.

[Cla78] Keith L. Clark. Negation as failure. In H. Gallaire and J. Minker, editors, *Logics and Data Bases*, pages 293–322. Plenum Press, 1978.

[Dij76] Edsger W. Dijkstra. *A Discipline of Programming*. Prentice-Hall, 1976.

[DPPR00] Agostino Dovier, Carla Piazza, Enrico Pontelli, and Gianfranco Rossi. Sets and constraint logic programming. *Transaction on Programming Language and Systems*, 22(5):861–931, 2000.

[DPR00] Agostino Dovier, Enrico Pontelli, and Gianfranco Rossi. A necessary condition for constructive negation in constraint logic programming. *Inf. Process. Lett.*, 74(3–4):147–156, 2000.

[End72] Herbert B. Enderton. *A Mathematical Introduction to Logic*. Academic Press, New York and London, 1972.

[GJ79] Michael R. Garey and David S. Johnson. *Computers and Intractability. A guide to the Theory of NP-Completeness*. Books in the Mathematical Sciences. W.H.Freeman and Company, New York, 1979.

[GL88] Michael Gelfond and Vladimir Lifschitz. The stable model semantics for logic programming. In Robert A. Kowalski and Kenneth A. Bowen, editors, *Logic Programming, Proceedings of the Fifth International Conference and Symposium*, pages 1070–1080, 1988.

[GL92] Michael Gelfond and Vladimir Lifschitz. Representing actions in extended logic programs. In K. Apt, editor, *Logic Programming, Proceedings of the Joint International Conference and Symposium*, pages 559–573. MIT Press, 1992.

[Her30] Jacques Herbrand. Recherches sur la theorie de la demonstration. Master's thesis, Université de Paris, 1930. Also in *Ecrits logiques de Jacques Herbrand*, PUF, Paris, 1968.

[HK73] John E. Hopcroft and Richard M. Karp. An $n^{5/2}$ algorithm for maximum matchings in bipartite graphs. *SIAM Journal on Computing*, 2(4):225–231, 1973.

[JL86] Joxan Jaffar and Jean-Louis Lassez. Constraint Logic Programming. Tech. rep., Department of Computer Science, Monash University, June 1986.

[JM94] Joxan Jaffar and Michael J. Maher. Constraint Logic Programming: A Survey. *Journal of Logic Programming*, 19–20:503–581, 1994.

[Kow74] Robert A. Kowalski. Predicate logic as programming language. In Jack L. Rosenfeld, editor, *Information Processing 74, Proceedings of IFIP Congress 74*, pages 569–574. North Holland, 1974.

[Lie05] Yuliya Lierler. Cmodels - SAT-based disjunctive answer set solver. In *LPNMR*, pages 447–451. Springer Verlag, 2005.

[Lif99] Vladimir Lifschitz. Answer set planning. In M. Gelfond, N. Leone, and G. Pfeifer, editors, *Proceedings of the 5th International Conference on Logic Programming and Nonmonotonic Reasoning (LPNMR-99)*, volume 1730 of *LNAI*, pages 373–374, Berlin, December 1999. Springer.

[Llo87] John W. Lloyd. *Foundations of Logic Programming*. Springer-Verlag, 2nd edition, 1987.

[LM04] Yuliya Lierler and Marco Maratea. Cmodels-2: SAT-based answer set solver enhanced to non-tight programs. In *LPNMR*, pages 346–350. Springer Verlag, 2004.

[Lov78] Donald W. Loveland. *Automated Theorem Proving*. North-Holland, Amsterdam, 1978.

[Mac77] Alan K. Mackworth. Consistency in networks of relations. *Artificial Intelligence*, 8(1):99–118, 1977.

[Men79] Elliott Mendelson. *Introduction to Mathematical Logic*. Van Nostrand, New York, second edition, 1979.

[MM82] Alberto Martelli and Ugo Montanari. An efficient unification algorithm. *ACM Transactions on Programming Languages and Systems*, 4:258–282, 1982.

[MS98] Kim Marriott and Peter Stuckey. *Programming with Constraints*. The MIT Press, 1998.

[PMG98] David Poole, Alan Mackworth, and Randy Goebel. *Computational Intelligence: A Logical Approach*. Oxford University Press, Oxford, 1998.

[PS98] Christos H. Papadimitriou and Ken Steiglitz. *Combinatorial optimization: Algorithms and Complexity*. Dover, second edition, 1998.

[PW78] Mike Paterson and Mark N. Wegman. Linear unification. *Journal of Computer System Science*, 16(2):158–167, 1978.

[Rég94] Jean-Charles Régin. A filtering algorithm for constraints of difference in CSPs. In *AAAI–Proceedings of the 12th National Conference on Artificial Intelligence*, pages 362–367, 1994.

[Rei78] Raymond Reiter. On closed world data bases. In H. Gallaire and J. Minker, editors, *Logic and Data Bases*, pages 55–76. Plenum, New York / London, 1978.

[RN03] Stuart J. Russell and Peter Norvig. *Artificial Intelligence: a modern approach*. Prentice Hall, 2nd international edition edition, 2003.

[Rob65] John. A. Robinson. A machine-oriented logic based on the resolution principle. *Journal of the ACM*, 12(1), 1965.

[Rob68] John. A. Robinson. The generalized resolution principle. *Machine Intelligence*, 3:77–93, 1968.

[Sar00] Vijay A. Saraswat. Concurrent constraint programming. In *17th Annual ACM Symposium on Principles of Programming Languages*, pages 232–245, 2000.

[Sho67] Joseph R. Shoenfield. *Mathematical Logic*. Addison Wesley, 1967.

[Sim00] Patrik Simons. *Extending and Implementing the Stable Model Semantics*. PhD thesis, Helsinki University of Technology, May 2000.

[SNS02] Patrik Simons, Ilkka Niemelä, and Timo Soininen. Extending and implementing the stable model semantics. *Artificial Intelligence*, 138(1–2):181–234, 2002.

[SRP01] Vijay A. Saraswat, Martin C. Rinard, and Prakash Panangaden. Semantic foundations of concurrent constraint programming. In *18th Annual ACM Symposium on Principles of Programming Languages*, pages 333–352, 2001.

[SS97] Leon Sterling and Ehud Shapiro. *The Art of Prolog*. The MIT Press, 2nd edition, 1997.

[Stu95] Peter J. Stuckey. Negation and constraint logic programming. *Information and Computation*, 118(1):12–33, April 1995.

[Syr01] Tommi Syrjänen. Lparse 1.0. user's manual. Technical report, Helsinki University of Technology, 2001. Disponibile in `http://www.tcs.hut.fi/Software/smodels`.

[War80] David H. D. Warren. Logic programming and compiler writing. *Software — Practice and Experience*, 10(2):97–125, 1980.

[Wei05] Eric W. Weisstein. Schur Number, 2005. From MathWorld–A Wolfram Web Resource, `http://mathworld.wolfram.com/SchurNumber.html`.