Linguaggi e Tecniche Speciali di Programmazione

Agostino Dovier

Dipartimento Scientifico-Tecnologico, Università di Verona, Strada Le Grazie 15, 37134 Verona, Italy
E-mail address: dovier@sci.univr.it

Contents

Chapter 1. Introduzione 5

Chapter 2. Logica del prim'ordine 9
1. Sintassi 9
2. Semantica 10
3. Sostituzioni 12

Chapter 3. Unificazione 17
1. Preliminari 17
2. Il problema dell'Unificazione 18
3. Algoritmo di unificazione 19
4. Osservazioni circa l'algoritmo Unify 22
5. E-unificazione 26

Chapter 4. Programmazione con clausole di Horn 31
1. Esempi di Programmi Definiti 31
2. Semantica operazionale del Logic Programming 35
3. Indipendenza del non-determinismo nell'SLD 38
4. Semantica Osservazionale dei programmi definiti 47
5. Turing Completezza di Prolog 50
6. Semantica logica (modellistica) di programmi definiti 51
7. Semantica di punto fisso di programmi definiti 55

Chapter 5. Programmazione in PROLOG 59
1. Liste 59
2. alberi 60
3. Grafi 61
4. Automi 62
5. Esempi 63
6. Predicati built-in 64
7. Predicati di tipo 64
8. Predicati metalogici o extralogici: 65
9. Il CUT 66
10. FAIL 68
11. Operatori 69

Chapter 6. Programmi Generali 71
1. Problemi 71
2. Proposte operazionali 72

3. Confronti tra le tre regole 75
4. Negazione Costruttiva 76
5. Implementazione della NaF 77

Chapter 7. Programmazione Dichiarativa 79
1. Principi di base 79
2. Generate and test 80
3. N-regine 81
4. Map Coloring 83
5. Commesso Viaggiatore 84
6. Predicati di secondo ordine 85
7. Metainterpreti 87
8. Quiz 88
9. Mondo dei blocchi 90
10. Giochi 92

Chapter 8. Programmi logici con vincoli 97
1. Sintassi e semantica operazionale 97
2. $CLP(\mathbb{R})$ in SICStus 100
3. $CLP(\mathcal{F}D)$ in SICStus 101
4. Metodologia Constrain & generate 106
5. N-Regine 106
6. Map Coloring 107
7. Marriage Problem 108
8. SEND + MORE = MONEY 108
9. Uso del cumulative 109
10. Allocazione di compiti 110
11. Constraints reificati 111

Bibliography 113

CHAPTER 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 scarsa. 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 ciasuna 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. Poichè 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 essre 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, ovvero 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 [**2**], 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) [**8**] fu resa possibile dagli studi sull'algoritmo di unificazione, in particolare la procedura di Paterson-Wegman [**6**] e l'algoritmo di Martelli-Montanari [**5**] 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, 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* (constraints). Si introdurranno pertanto le capacità delle moderne versioni di PROLOG di programmazione con vincoli e si mostrerà come risolvere in modo dichiarativo, ma con efficienza comparabile a qualunque altro metodo, diversi problemi pratici.

Il contenuto del corso si basa ma non esclusivamente su diversi testi [**1, 4, 3, 7**] 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. La stesura della dispensa ha beneficiato della collaborazione degli studenti del V anno di Informatica dell'Università di Verona A.A. 1999/2000, che di seguito elenco e ringrazio per il lavoro svolto: Claudio Arduini, Mario Ballarin, Lorenzo Benamati, Marco Bersellini, Matteo Burato, Stefano Fiore, Matteo Galavotti, Giuliano Grigolato, Daria Girelli, Matteo Lotto, Alessio Merlo, Francesco Orsolato, Gianluca Rambaldelli, Emanuele Vicentini, Stefano Zanetti, e Simone Zuliani.

CHAPTER 2

Logica del prim'ordine

1. Sintassi

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

- $ar(X) = 0 \; \forall X \in \mathcal{V}$;
- $ar(f) \geq 0 \; \forall f \in \mathcal{F}$. Se $c \in \mathcal{F}$ è un simbolo di costante, allora $ar(c) = 0$;
- $ar(p) \geq 0 \; \forall p \in \Pi$.

Parentesi aperte e chiuse, virgole, e i connettivi logici (,), $\wedge$, $\vee$, $\neg$, $\rightarrow$, $\leftrightarrow$, $\forall$, $\exists$ 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 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 2.3. $f(X, f(a, b))$ con $ar(f) = 2$ e $ar(b) = ar(a) = 0$ è un termine. $+(1, \cdot(3, 5))$ è un termine. ab non è un termine.

DEFINIZIONE 2.4. s è un *sottotermine* di t se s è una sottostringa di t che è anche un termine. Un sottotermine si dice *proprio* se è un sottotermine e non è il termine stesso.

ESEMPIO 2.5. Dato il termine $f(f(a, b), g(X, b))$, sono suoi sottotermini: $a, b, X, f(a, b)$, $g(X, b)$, oltre al termine stesso. $f(f(a$, non è un sottotermine.

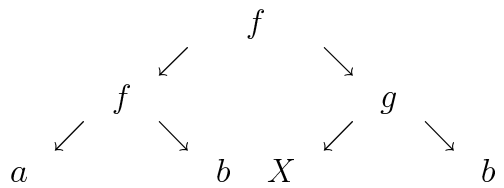


FIGURE 1. Albero associato ad un termine

E' naturale associare un albero finito ad un termine. Costruendo tale albero (cf. Figura 1) si evince una relazione tra sottotermini e sottoalberi dell'albero.

Due termini possono essere sintatticamente uguali ($s \equiv t$) o diversi ($s \not\equiv t$). Dato un termine t con $vars(t)$ denotiamo l'insieme delle variabili in t. Se $vars(t) = \emptyset$ allora t è detto *ground*. $T(\mathcal{F})$ è l'insieme dei termini ground. $T(\mathcal{F}, \mathcal{V})$ è l'insieme dei termini.

DEFINIZIONE 2.6. Se $p \in \Pi$, $ar(p) = n$ 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 ground, allora $p(t_1, \ldots, t_n)$ è detta *ground*.

Le formule si definiscono in modo induttivo:

DEFINIZIONE 2.7.
- 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:

- $\varphi \wedge \psi$ 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 $\neg\, \exists X \neg\varphi$

DEFINIZIONE 2.8. Diciamo che una variabile X *occorre libera* in una formula φ se:

- 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 $\psi \vee \eta$ e X occorre libera in ψ o X occorre libera in η
- φ è $\exists Y\psi$ se X occorre libera in ψ e $X \neq Y$.

Se X è presente in φ e X non è libera in φ allora è *legata*.

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*.

2. Semantica

DEFINIZIONE 2.9. Un'*interpretazione* (struttura) $\mathcal{A} = \langle A, (\cdot)^{\mathcal{A}} \rangle$ per un alfabeto Σ consiste di:

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

DEFINIZIONE 2.10. Sia $\mathcal{A} = \langle A, (\cdot)^{\mathcal{A}} \rangle$ struttura, t termine, $B \supseteq vars(t)$, e $\sigma : B \longrightarrow A$ un assegnamento di elementi del dominio A alle variabili di B. Si definisce la valutazione

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

$$(t\sigma)^{\mathcal{A}} = \begin{cases} \sigma(X) & \text{se } t \equiv X \in \mathcal{V} \\ (c)^{\mathcal{A}} & \text{se } t \equiv c \in \mathcal{F},\ ar(c) = 0 \\ (f)^{\mathcal{A}}((t_1\sigma)^{\mathcal{A}}, \ldots, (t_n\sigma)^{\mathcal{A}}) & \text{se } t \equiv f(t_1, \ldots, t_n) \end{cases}$$

DEFINIZIONE 2.11. Sia $\mathcal{A}$ struttura, $p(t_1, \ldots, t_n)$ formula atomica, $B \supseteq \mathit{vars}(p(t_1, \ldots, t_n))$ e $\sigma : B \longrightarrow A$. Allora il *valore di verità* della formula atomica è definito:

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

Data una formula φ ed un assegnamento σ il fatto che la formula abbia valore di verità true in $\mathcal{A}$ viene denotato come $\mathcal{A} \models \varphi\sigma$. Il valore false come $\mathcal{A} \not\models \varphi\sigma$.

Diamo ricorsivamente la regola per stabilire se $\mathcal{A} \models \varphi\sigma$ per una generica formula.

DEFINIZIONE 2.12.
- $\mathcal{A} \models p(t_1, \ldots, t_n)\sigma$ sse $V^{\mathcal{A}}(r(t_1, \ldots, t_n)) = \mathsf{true}$
- $\mathcal{A} \models \neg\varphi\sigma$ sse $\mathcal{A} \not\models \varphi\sigma$
- $\mathcal{A} \models (\varphi \vee \psi)\sigma$ sse $\mathcal{A} \models \varphi\sigma$ oppure $\mathcal{A} \models \psi\sigma$
- $\mathcal{A} \models (\exists X\varphi)\sigma$ sse esiste un elemento $a \in A$ t.c. $\mathcal{A} \models \varphi[X/a]\sigma$.

Poiché nessun assegnamento di variabili ha effetto sugli enunciati, abbiamo stabilito il significato della notazione $\mathcal{A} \models \varphi$ dato un generico enunciato φ. Essa viene in genere indicata con le tre equivalenti diciture:

- φ è vera in $\mathcal{A}$;
- φ è soddisfatta in $\mathcal{A}$;
- $\mathcal{A}$ è modello di φ.

DEFINIZIONE 2.13. Sia φ una formula tale che $\mathit{vars}(\varphi) = \{X_1, \ldots, X_n\}$. Allora si definisce $\mathcal{A} \models \varphi$ se e solo se esistono $a_1, \ldots, a_n \in \mathcal{A}$ t.c. $\mathcal{A} \models \varphi[X_1/a_1, \ldots, X_n/a_n]$.

Si osservi che $\mathcal{A} \models \varphi$ se e solo se $\mathcal{A} \models \exists X_1 \cdots \exists X_n\varphi$.
Nel caso in cui φ contenga variabili libere, si dice inoltre che φ è soddisfacibile in $\mathcal{A}$.

DEFINIZIONE 2.14. 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 un'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 2.15. 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 è *insoddisfacibile* o contraddittorio.
- φ è una *conseguenza logica* di Γ se per ogni modello $\mathcal{A}$ di Γ si ha che se $\mathcal{A} \models \Gamma$ allora $\mathcal{A} \models \varphi$. Con abuso di notazione ciò si indica con $\Gamma \models \varphi$.
- Se $\Gamma = \emptyset$, la notazione $\models \varphi$ esprime la validità di φ.

LEMMA 2.16. *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 2.17. Sia Σ una segnatura 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:

$$\begin{aligned} (0)^{\mathcal{N}} &= 0 \\ (s)^{\mathcal{N}} &= \lambda x.x + 1 \\ (+)^{\mathcal{N}} &= \lambda x, y.x + y \end{aligned}$$

Si ha che:

- $\mathcal{N} \models \forall Y \exists X(X = s(Y))$
- $\mathcal{N} \not\models \forall Y \exists X(Y = s(X))$ (si istanzi Y con 0).

Il secondo enunciato sarà invece vero usando un'interpretazione sugli interi $\mathcal{Z} = \langle \mathbb{Z}, (\cdot)^{\mathcal{Z}} \rangle$. In tal caso 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 della segnatura, gli elementi del dominio che la soddisfano (ad esempio -1).

DEFINIZIONE 2.18. Dato un alfabeto $\Sigma = (\mathcal{F}, \Pi, \mathcal{V})$, la *(pre)-interpretazione di Herbrand* $\mathcal{H}$ si definisce nel modo seguente:

- il dominio è $T(\mathcal{F})$, detto *Universo di Herbrand*;
- la funzione di interpretazione di Herbrand manda ogni termine ground in se stesso ed è così definita:

$$\begin{cases} c^{\mathcal{H}} &= c \\ f^{\mathcal{H}}(t_1, \cdots, t_n) &= f(t_1^{\mathcal{H}}, \cdots, t_n^{\mathcal{H}}) \end{cases}$$

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

Nel caso in cui l'alfabeto sia privo di simboli di funzione, $\mathcal{F} = \emptyset$, allora anche l'universo di Herbrand ad esso associato è vuoto, $T(\mathcal{F}) = \emptyset$. Se, invece, esistono nell'alfabeto almeno un simbolo di costante ed un simbolo di funzione, allora $T(\mathcal{F})$ ha cardinalità infinita.

ESEMPIO 2.19. Proviamo a descrivere l'universo di Herbrand dell'alfabeto dell'esempio 2.17 .

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

ESEMPIO 2.20. I seguenti enunciati vengono soddisfatti dall'interpretazione di Herbrand:

$$\begin{aligned} &\mathcal{H} \models \forall y \exists x(x = s(y)) \\ &\mathcal{H} \not\models \forall y \exists x(y = s(x)) \\ &\mathcal{H} \not\models 0 + s(0) = s(0) + 0 \end{aligned}$$

mentre per quello che avevamo visto in precedenza, $\mathcal{N} \models 0 + s(0) = s(0) + 0$.

3. Sostituzioni

DEFINIZIONE 2.21. Una *sostituzione* è una funzione $\sigma : \mathcal{V} \longrightarrow T(\mathcal{F}, \mathcal{V})$ tale che $dom(\sigma) \equiv \{X \in \mathcal{V} : \sigma(X) \not\equiv X\}$ è un insieme finito.

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 $i = 1, \ldots, n$.

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

DEFINIZIONE 2.22.
- Se $t_1, \ldots, t_n$ sono tutte variabili allora σ è detta *sostituzione di variabili*.
- Se inoltre $t_1, \ldots, t_n$ sono tutte distinte, allora la sostituzione è detta *rinomina* (renaming);
- Se inoltre $dom(\sigma) = ran(\sigma)$ ovvero $\{X_1, \ldots, X_n\} = \{t_1, \ldots, t_n\}$, allora è detta *variante* (permutazione di variabili).
- Se $t_1, \ldots, t_n$ sono tutti ground allora σ è detta *ground*.

ESEMPIO 2.23. $[X/a, Y/b], [X/f(X,Y), Y/g(Z,X)]$ sono sostituzioni. $[X/Z, Y/Z]$ è una sostituzione di variabili (ma non una rinomina). $[X/Z, Y/W]$ è un rinomina. $[X/Y, Y/X]$ è una variante.

LEMMA 2.24. *Sia σ una rinomina allora esiste una variante $\sigma' \supseteq \sigma$ (σ' estende σ) tale che $dom(\sigma') = ran(\sigma') = \text{vars}(\sigma)$.*

PROOF. Per semplicità, si ragioni pensando da un esempio specifico (si pensi a $\sigma = [X_3/X_1, X_4/X_2, X_5/X_3, X_6/X_4, X_7/X_5]$).

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' (ad esempio $g = [X_1/X_6, X_2/X_7]$ per ottenere $\sigma' = \sigma \cup [X_1/X_6, X_2/X_7]$). σ' è una variante per costruzione . □

DEFINIZIONE 2.25. L'*applicazione di una sostituzione* σ ad un termine t, $t\sigma$ è definita nel modo seguente:

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

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 2.26. 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 2.27. Un termine s è un *istanza* di un termine t se esiste σ tale che $s = t\sigma$. Si potrebbe, senza perdere di generalità assumere che $dom(\sigma) \subseteq vars(t)$.

ESEMPIO 2.28. $f(a,a)$ è istanza di $f(X,Y)$, poichè $f(a,a) = f(X,Y)[X/a, Y/a]$. Avremmo anche potuto scrivere $f(a,a) = f(X,Y)[X/a, Y/a, Z/b]$. $f(Z,Z)$ è istanza di $f(X,Y)$ ma non viceversa.

DEFINIZIONE 2.29. Il termine s è una *variante* del termine t se esiste σ, variante, tale che $s = t\sigma$.

ESEMPIO 2.30. $f(X,Y)$ è una variante di $f(Y,X)$, poichè $f(X,Y) = f(Y,X)[X/Y, Y/X]$. $f(X,Y)$ è una variante di $f(X,Z)$, poichè $f(X,Y) = f(X,Z)[Z/Y, Y/Z]$.

$f(X,X)$ non è una variante di $f(X,Y)$. Infatti $f(X,X) = f(X,Y)[Y/X]$ ma $[Y/X]$ non è una variante. Bisognerebbe aggiungere $[X/?]$. Tuttavia, facendolo, non si otterrebbe più $f(X,X)$.

DEFINIZIONE 2.31. Dato $V \subseteq \mathcal{V}$ e σ definiamo $\sigma\,|_V$ (σ *ristretto a* V)

$$\sigma\,|_V\,(X) = \begin{cases} \sigma(X) & \text{se } X \in \mathcal{V} \\ X & \text{altrimenti.} \end{cases}$$

Date θ e η sostituzioni definiamo la *composizione* $\theta\eta$ di sostituzioni come: $(\theta\eta)X = (X\theta)\eta$.

LEMMA 2.32. *Siano*

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

Allora si ha che $\theta\eta$ 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 $[V/V]$.

PROOF. Esercizio. □

ESERCIZIO 2.33. Dimostrare che:

(1) $\theta\varepsilon = \varepsilon\theta = \theta$, dove ε è la sostituzione vuota e θ è una sostituzione qualsiasi.
(2) Per ogni variante θ esiste esattamente una sostituzione γ tale che $\theta\gamma = \gamma\theta = \varepsilon$. Questa γ la chiamiamo θ^{-1}.
(3) Se $t = t\theta$ allora $\theta|_{vars(t)} = \varepsilon$.
(4) 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.

LEMMA 2.34. (1) *Per ogni coppia di sostituzioni θ e η e per ogni termine s si ha che $(s\theta)\eta = s(\theta\eta)$.*
(2) *Per ogni terna di sostituzioni θ, η e γ si ha che $(\theta\eta)\gamma = \theta(\eta\gamma)$.*

PROOF. La seconda proprietà è banale. Per la prima, induciamo sulla struttura del termine s.
Base: $s \equiv X$: per definizione di composizione.
Passo:

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

□

LEMMA 2.35 (Varianti). *(i) s è variante di t se e solo se s è istanza di t e t è istanza di s,*

(ii) se s è variante di t allora $s = t\theta$ per almeno una variante θ t.c. $vars(\theta) \subseteq vars(s) \cup vars(t)$.

PROOF. Dimostrando il punto *(i)*, viene automaticamente dimostrato anche l'altro.

($\Rightarrow$) Se s è variante di t, allora per definizione esiste θ variante t.c. $s = t\theta$.

Sappiamo (Esercizio 2.33.2) che data una variante esiste un'unica sostituzione che è la sua inversa tale per cui $\theta\theta^{-1} = \theta^{-1}\theta = \varepsilon$. Nel nostro caso, partendo da $s = t\theta$, dove θ è una biiezione da variabili in variabili, se noi 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 pedita 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 2.33).

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

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), poichè $X \in dom(\theta) \subseteq vars(t)$. 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 2.24 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: assurdo per l'iniettività di θ su $vars(t)$.

Pertanto, da $s = t\theta$ consegue che $s = t\theta'$ in quanto nuovi bindings 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 come è costruita, si ha subito che $vars(\theta') \subseteq vars(s) \cup vars(t)$ (come nell'enunciato *(ii)*). □

DEFINIZIONE 2.36. Date θ e τ sostituzioni, θ è *più generale* di τ (si indica con $\theta \leq \tau$) se $\exists\eta$ sostituzione t.c. $\tau = \theta\eta$.

ESEMPIO 2.37. (1) Considero $[X/Y]$, $[X/a, Y/a]$. La seconda sostituzione si può ottenere applicando alla prima $[Y/a]$: $[X/Y][Y/a] = [X/a, Y/a]$, 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 può valere anche il viceversa. Mentre nell'esempio precedente la sostituzione rimpiazzava variabili con termini ground, in questo caso abbiamo invece due variabili: $[X/Z, Y/Z][Z/Y] = [X/Y, Y/Y, Z/Y] = [X/Y, Z/Y]$. Non abbiamo ottenuto $[X/Y]$; il problema vero è che è ineliminabile la presenza di Z nel dominio della sostituzione risultante.

(3) ε è più generale di ogni sostituzione. Prendendo una generica sostituzione θ si ha $\theta = \varepsilon\theta$. Si può pensare a una forma di ordinamento (pre-ordine) tra sostituzioni in base alla relazione $\leq$, dove nel bottom c'è ε e sopra stanno le altre meno generali.

(4) $[X/f(Y, Z)]$ è più generale di $[X/f(a, a)]$? Applicando $[Y/a, Z/a]$ alla prima otteniamo: $[X/f(Y, Z)][Y/a, Z/a] \neq [X/f(a, a)]$. Le due sostituzioni non sono confrontabili con $\leq$.

(5) $[X/Y, Y/Z, Z/X] \leq [X/Z, Y/X, Z/Y]$. Infatti $[X/Y, Y/Z, Z/X][Y/Z, Z/X, X/Y] = [X/Z, Y/X, Z/Y]$. Se però partiamo dalla seconda sostituzione, otteniamo:

$$[X/Z, Y/X, Z/Y][Z/Y, X/Z, Y/X] = [X/Y, Y/Z, Z/X]$$

Le due sostituzioni sono reciprocamente più generali una 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*.

LEMMA 2.38. *Se $\theta \leq \tau$, allora, per ogni termine t, $t\tau$ è istanza di $t\theta$.*

PROOF. Consideriamo $\theta \leq \tau$. Allora, per definizione, esiste η t.c. $\tau = \theta\eta$. Predndo un termine t qualsiasi: $t\tau \stackrel{def}{=} t(\theta\eta) \stackrel{A}{=} (t\theta)\eta$. Questo dimostra che $t\tau$ è istanza di $t\theta$. □

LEMMA 2.39. *$\theta \leq \tau$ e $\tau \leq \theta$ se e solo se $\exists\gamma$ t.c. γ è una variante e $vars(\gamma) \subseteq vars(\theta) \cup vars(\tau)$ e $\tau = \theta\gamma$.*

PROOF. Esercizio. Suggerimento: si segua la dimostrazione del Lemma 2.35. □

CHAPTER 3

Unificazione

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

1. Preliminari

DEFINIZIONE 3.1. Dati due termini s e t,

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

ESEMPIO 3.2. (1) Consideriamo i termini $f(g(X,a),Z)$ e $f(Y,b)$. Un loro unificatore è $[Y/g(c,a),X/c,Z/b]$; un altro: $[Y/g(h(W),a),X/h(W),Z/b]$. I due unificatori sono tra loro indipendenti rispetto a $\leq$. L'*m.g.u.* (in questo caso unico) è: $[Y/g(X,a),Z/b]$.

(2) $f(a,b)$ e $f(X,b)$: l'unificatore è $[X/a]$.

(3) $f(a,b)$ e $f(X,a)$: non esiste nessun unificatore per questi due termini perché comunque sostituisca X, $b \neq a$. Questo esempio si può capire meglio rappresentando i termini con la struttura ad albero di figura 1. L'unificabilità si può infatti ridurre ad un problema di ricerca di isomorfismo di alberi etichettati.

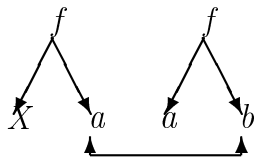


FIGURE 1.

(4) Consideriamo X e $g(X)$. Supponiamo che l'unificatore sia la sostituzione $[X/g(X)]$. Quello che otteniamo dopo averla applicata è $g(X)$ e $g(g(X))$. Se applico infinite volte $g(g(\cdots(X)\cdots))$ ottengo due termini di lunghezza infinita che differiscono per un $g(X)$ (e sono dunque uguali). La soluzione è comunque inaccettabile perché noi consideriamo solo termini di lunghezza finita.

Si osservi pertanto che se s è sottotermine proprio di t non sarà mai unificabile con t.

DEFINIZIONE 3.3. $C \equiv s_1 = t_1 \wedge \cdots \wedge s_n = t_n$ (con s_i, t_i termini) si dice *sistema di equazioni*.

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

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 generico.
(3) Verificare che non esiste un unificatore, altrimenti fornire l'*m.g.u.*.

Si noti che saper risolvere (2) implica la soluzione di (1). Inoltre, se l'esistenza di un mgu a partire dall'unificabilità è garantita (come di fatto lo è), allora (3) implica (2). L'algoritmo di unificazione risolve direttamente il punto (3).

L'algoritmo che vedremo noi deriva da quello presente nella tesi di dottorato di Herbrand [**?**]. [1]

DEFINIZIONE 3.4. 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 che non occorrono mai in $t_1, \ldots, t_n$. In altri termini, C ha la forma $X_1 = t_1 \wedge \cdots \wedge X_n = t_n$ t.c. $\forall i \neq j$, $X_i \not\equiv X_j$ 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 vale che $dom(\theta) \cap ran(\theta) = \emptyset$. Tale condizione assicura l'idempotenza, ovvero $t\theta = t\theta\theta$ per ogni termine t.

LEMMA 3.5. *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 mgu di C.*

PROOF. Dobbiamo prima dimostrare che θ è un unificatore. Per fare questo devo dimostrare che per $i = 1, \ldots, n$: che $X_i\theta = t_i\theta$. Ma $X_i\theta = t_i$ per definizione di θ e $t_i\theta = t_i$ poichè per ipotesi (forma risolta) $X_i \notin vars(t_i)$.

Mostriamo ora che è *mgu*, ovvero che se σ è un unificatore allora $\exists\eta$ t.c. $\sigma = \theta\eta$. Mostriamo una proprietà più forte: se σ è un unificatore allora $\sigma = \theta\sigma$. Distinguiamo due casi, le variabili $X_1, \ldots, X_n$ e tutte le altre.

- Per $i = 1, \ldots, n$: $X_i\sigma = t_i\sigma$ (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 banalmente: $V_i\sigma = V_i\theta\sigma = V_i\sigma$ in quanto $V_i\theta = V_i$.

□

LEMMA 3.6. *[Binding] Sia θ una sostituzione. Se $X\theta = t\theta$ allora $\theta = [X/t]\theta$.*

PROOF. Deve valere per i seguenti casi:

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

□

[1] Il primo algoritmo di unificazione che fu inserito in una procedura risolutiva è quello di Robinson (1965). Martelli e Montanari in [**5**] svilupparono un'ottimizzazione dell'algoritmo di unificazione di Herbrand in grado di eseguire l'unificazione in tempo quaso lineare. Paterson e Wegman in [**?**] presentarono una tecnica completamente diversa (distruttiva) che permette di effettuare l'unificazione in tempo lineare.

3. Algoritmo di unificazione

Questo algoritmo, dovuto ad Herbrand, è composto da sei regole di riscrittura, che vengono ripetute in modo non deterministico finché è possibile:

(1)	$f(s_1,\dots,s_n) = f(t_1,\dots,t_n) \wedge C$	$\mapsto$	$s_1 = t_1 \wedge \cdots \wedge s_n = t_n \wedge C$
(2)	$f(s_1,\dots,s_m) = g(t_1,\dots,t_n) \wedge C$ $f \neq g$ 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)	$X = t \wedge C$ $X \notin \mathit{vars}(t), X \in \mathit{vars}(C)$ $X \notin \mathit{vars}(t) \wedge X \in \mathit{vars}(C)$	$\mapsto$	$C[X/t] \wedge X = t$
(6)	$X = t \wedge C$ $X \in \mathit{vars}(t), X \not\equiv t$	$\mapsto$	false

La regola (1) include implicitamente anche $a = a \wedge C \mapsto C$ per ogni termine costituito dalla costante a. Si osservi che questo algoritmo non introduce nuove variabili.

Prima di dimostrare il teorema 3.13 ci fa comodo introdurre alcuni concetti:

Definizione 3.7. Una relazione d'ordine $\prec \subseteq A \times A$ è un *buon ordine* quando non esistono infiniti $a_1, a_2, \dots \in A$ t.c. $\cdots \prec a_4 \prec a_3 \prec a_2 \prec a_1$ (catena discendente infinita).

Esempio 3.8. $\langle \mathbb{N}, \prec \rangle$ è un buon ordine. $\langle \mathbb{Z}, \prec \rangle$ non è un buon ordine.

Definizione 3.9. 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. L'ordine lessicografico $\langle A \times B, \prec \rangle$ (che altro non è che l'ordinamento del vocabolario: si confronta prima il primo carattere, poi il secondo e cosí via) è definito come:

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

L'idea puó facilmente estendere a terne, quaterne e cosí via. Si puó intuitivamente dimostrare che questo 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 minori 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 non posso costruire una catena discendente infinita.

Assumiamo la nozione primitiva di multiinsieme, come quella di un insieme in cui le ripetizioni "contano". Pertanto, ad esempio, $\{\!\{5,4,5,5,3,2\}\!\} \neq \{\!\{5,4,3,2\}\!\}$. Questo tipo di oggetti sono utili per modellare contenitori di entità che "si consumano". I multiinsiemi che useremo sono *finiti* e *piatti*, ovvero contengono solo elementi presi da un insieme A (non contengono altri multiinsiemi). Se $\langle A, < \rangle$ è un buon ordine, allora un buon ordinamento per

i multiinsiemi 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 $n \geq 0$ (potrei anche non mettere nessun t_i).

A parole, un multiinsieme è piú piccolo di un'altro se è ottenuto dal primo eliminando un elemento o sostituendolo uno o piú elementi piú piccoli.

ESEMPIO 3.10. $\{\!\{5,1,1,1,2,2,2,0,0,0,2,1,2\}\!\} \prec \{\!\{5,3,2,1,2\}\!\}$: In questo caso il primo è stato ottenuto dal secondo sostituendo a 3 gli elementi. 1,1,1,2,2,2,0,0,0. $\{\!\{5,3,1,2\}\!\} \prec \{\!\{5,3,2,1,2\}\!\}$: In questo caso il primo è stato ottenuto dal secondo eliminando il terzo elemento.

Introduciamo anche una misura per i termini:

DEFINIZIONE 3.11. Sia t un termine, allora

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

La *size* conta il numero di occorrenze di simboli di $\mathcal{F}$ in t. Infatti, $size(X) = 0$ se X è una variabile. $size(f(a, b, f(X, c, Y))) = 5$. Considerando la somma delle *size* di tutti i termini presenti in un sistema, si osserva che alcune azioni dell'algoritmo fanno calare tale valore. Ciò tuttavia non vale per tutte le azioni: se sostituisco ad X il termine $f(f(f(a)))$ (la cui *size* vale 4), aumenta la *size* totale del sistema di un numero legato alle occorrenze di X (che prima non pesava sul sistema in quanto variabile). Cerchiamo un'altra misura che cali in questo caso:

DEFINIZIONE 3.12. Una variabile $X \in vars(C)$ è detta *risolta* se X occorre in C solo come $X = t \wedge C'$ con $X \notin vars(t) \wedge X \notin vars(C')$. Dato un sistema C, definiamo l'insieme *Unsolved fi* C nel modo seguente:

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

TEOREMA 3.13. (1) Unify(C) *termina per qualsiasi sistema* C.
(2) Unify(C) *restituisce* C' *che è* false *oppure un sistema in forma risolta* C'.
(3) θ *è unificatore di* C *se e solo se* θ *è unificatore di* C'.

PROOF. 1) Unify(C) termina per qualsiasi sistema C.

Dimostriamo che qualunque passo di Unify fa calare una misura complessità del sistema. La misura è la seguente:

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

ove l'ordinamento più esterno è quello lessicografico, quello sul primo elemento è il $<$ su $\mathbb{N}$, quello sul secondo è l'ordinamento di multiinsiemi di numeri naturali. Poiché questo è un buon ordine, se mostro che ogni passo fa calare la complessità, ne seguirà la terminazione dell'algoritmo.

Per farlo analizzo il comportamento di tutte le regole di Unify:

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 $s_1 = t_1 \wedge \cdots \wedge s_n = t_n$ potrebbe essere del tipo $X = t_i$ con X non

altrove, quindi $Uns(C)$ cala o resta costante. Di certo non puó aumentare. Sicuramente il multiinsieme cala perchè $1 + \sum_{i=1}^{n} size(s_i)$ viene eliminato e sostituito con $size(s_1), \ldots, size(s_n)$.

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

Regola (3): $X = X \wedge C \mapsto C$. Questa regola fa sparire l'equazione $X = X$. Può essere che Uns cali o rimanga costante, ma anche questa volta cala il multiinsieme perchè $size(X = X) = 0$ e questa occorrenza di 0 viene buttato via.

Regola (4): $t = X \wedge C \mapsto X = t \wedge C$. Anche in questo caso Uns puó calare o rimanere costante ma il multiinsieme cala perchè visto che t non è variabile $size(t) > 0$, quindi nel multiinsieme rimpiazzo $size(t)$ con zero.

Regola (5): $X = t \wedge C \mapsto C[X/t] \wedge X = t$. X non è in t ma è in C. X è diventata risolta, quindi calano le non risolte Uns.

2) Sappiamo che Unify termina. Se non ha terminato con false, e non è in forma risolta, si vede, osservando le azioni possibili, che almeno una di esse risulta applicabile, contraddicendo l'ipotesi.

3) Per mostrare che θ è unificatore di C se e solo se lo è di C', mostriamo la proprietà passo per passo.

(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 $s_i\theta = t_i\theta$ per $i \in \{1, \ldots, n\}$.

(2) Per definizione di unificatore, non esiste nessun unificatore di $f(\ldots)$ e $g(\ldots)$ con f diverso da g.

(3) Ogni sostituzione è unificatore di $X = X$. Banale.

(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 3.6 ho che 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 Binding lemma (Lemma 3.6) $\theta = [X/t]\theta$. 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: non saranno mai unificabili.

□

COROLLARIO 3.14. Se Unify(C) = false allora C non ammette unificatori. Se Unify$(C) \equiv X_1 = t_1 \wedge \cdots \wedge X_n = t_n$ é in forma risolta, allora $[X_1/t_1, \ldots, X_n/t_n]$ é mgu di C.

NOTA 3.15. Per ragioni di semplicità implementativa, in SICStus Prolog l'unificazione è implementata senza il controllo "occur-check". Si può tuttavia richiederlo esplicitamente, chiamando la libreria terms ed il predicato built-in unify_with_occurs_check. In dettaglio, vanno dati i comandi:

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

che può anche essere messo in un file come direttiva e poi, se S e T sono i due termini da unificare, si chiama:

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

Per applicarlo a sistemi di equazioni sarà sufficiente definire ricorsivamente un predicato che richiama quest'ultimo come caso base.

4. Osservazioni circa l'algoritmo Unify

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. Abbiamo trascurato questo aspetto non deterministico dell'algoritmo, perchè fino ad ora ci interessava analizzare solo l'aspetto riguardante il risultato finale. Vediamo ora invece di chiarire questo concetto con degli esempi.

Ammettiamo di avere un sistema C, supponiamo che si possano applicare a C le regole (i) e (j). Nel primo caso otteniamo un sistema C' mentre nel secondo un sistema C''. Se poi a questi applichiamo rispettivamente le regole (j) e (i) all'altro, il risultato che si trova, sará lo stesso? In altri termini, le strade non deterministiche che si aprono, alla fine si chiudono? In generale no.

$$\begin{array}{ccc} & C & \\ (i)\swarrow & & \searrow(j) \\ C' & & C'' \\ (j)\searrow & & \swarrow(i) \\ & ?? & \end{array}$$

ESEMPIO 3.16. In questo esempio verrà risolta un'equazione seguendo due strade distinte, e mostreremo cosí come sia possibile giungere a mgu distinti.

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

Scegliamo a caso tra una delle 3 equazioni, prendiamo la prima $X = Y$. Si applica quindi il caso (5) dell'algoritmo di unificazione, ovvero si applica $[X/Y]$ e si giunge a:

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

Selezioniamo ora la seconda equazione; la regola (5) fa applicare la sostituzione $[Y/Z]$ ottenendo:

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

$Z = Z$ viene eliminata dal passo (3): abbiamo quindi trovato un sistema in forma risolta, con $\theta = [X/Z, Y/Z]$.

Consideriamo ora una diversa scelta non deterministica all'inizio: scegliamo la seconda equazione da $X = Y \wedge Y = Z \wedge Z = X$, applicando con la regola (5) la sostituzione $[Y/Z]$:

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

Seleziono la terza equazione applicando $[Z/X]$, giungendo a

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

Si elimina $X = X$ con il passo (3) ottenendo il sistema in forma risolta con $\theta' = [Y/X, Z/X]$.

$$\begin{array}{ccc} & C & \\ \swarrow & & \searrow \\ \theta & & \theta' \end{array}$$

Ciò che possiamo però osservare è che i due mgu trovati sono delle varianti, infatti:

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

TEOREMA 3.17. *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' tra loro varianti.*

PROOF. Dal Teorema 3.13 conosciamo l'equivalenza di unificatori tra C e θ_1' e tra C e θ_2'. Inoltre θ_1' e θ_2' sono entrambe m.g.u. Dunque varrà che: $\theta_1' \leq \theta_2'$ e $\theta_2' \leq \theta_1'$. Per il Lemma 2.39 θ_1' e θ_2' sono varianti. □

4.2. Aspetti Computazionali di Unify. Ci chiediamo ora quali sono le caratteristiche computazionali dell'algoritmo Unify.

ESEMPIO 3.18. Prendiamo il sistema di equazioni:

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

Selezioniamo la quarta equazione, e applichiamo la sostituzione indotta

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

Ora possiamo applicare la sostituzione per X_3:

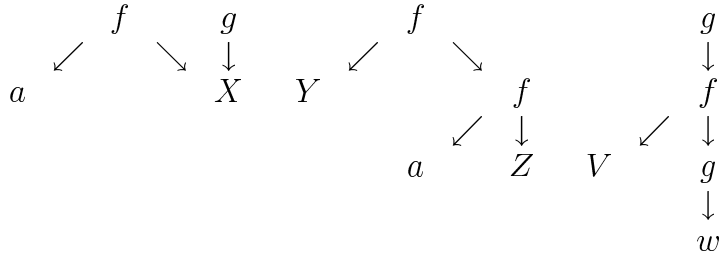
$$X_1 = f(X_2, X_2) \wedge X_2 = f(f(f(X_5, X_5), f(X_5, X_5)), f(f(X_5, X_5), f(X_5, X_5))) \wedge$$
$$X_3 = f(f(X_5, X_5), f(X_5, X_5)) \wedge X_4 = f(X_5, X_5)$$

La dimensione dei termini cresce molto velocemente, il termine X_4 conta 1, X_3 conta 3, X_2 conta 7 ed X_1 conta 15. Dunque l'applicazione di questo algoritmo conduce alla costruzione di termini che cresce in maniera esponenziale rispetto alle dimensioni del sistema iniziale. Usare Unify nel modo descritto genera pertanto il problema della crescita esponenziale dei termini (tuttavia, la visione non-deterministica con applicazione esplicita di sostituzioni lo rende adatto alle dimostrazioni di correttezza, completezza e terminazione). Vedremo ora quali tecniche e strutture dati ci permetteranno di ovviare a tale problema.

Si consideri il sistema di equazioni:

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

Costruiamo un grafo associando un nodo ad ogni occorrenza di simbolo funzionale e un unico nodo per ogni variabile. 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 può a questo punto essere mimato sul grafo costruito. Si terrà conto delle equazioni presenti. Si seleziona una equazione $\ell = r$ e si collassano i due nodi radice associati ai termini ℓ e r. Se i due nodi sono etichettati:

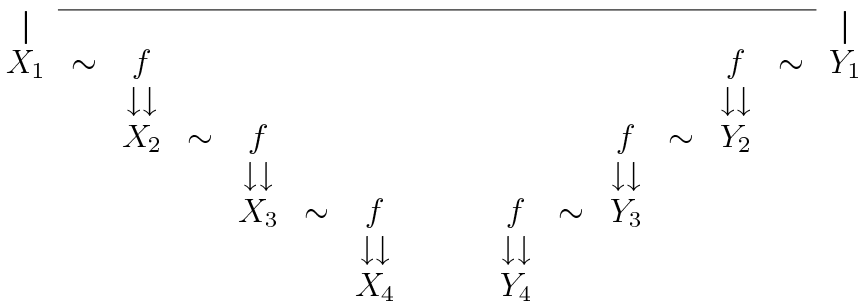
- con simboli funzionali diversi, allora si finisce con fallimento.
- con lo stesso simbolo funzionale f, allora si aggiungono equazioni per i figli senza ripetere la stessa equazione,
- uno con una variabile e l'altro con un simbolo f, allora si effettua l'*occur-check* (si verifica l'aciclicità del grafo ottenuto).

.

ESEMPIO 3.19. Consideriamo il sistema

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

Su questo sistema l'algoritmo di Herbrand ha un comportamento esponenziale. Costruiamo il grafo associato:



L'esecuzione dell'algoritmo sul grafo avviene in tempo polinomiale.

4.3. Significato dell'unificazione. Ci chiediamo ora cosa significhi per due termini "unificare"? I due termini $f(1,2)$ e $f(1,2)$ sono sintatticamente uguali; sappiamo in particolare che unificano. Qualunque sia l'interpretazione di $f, 1, 2$ gli oggetti identificati da quei termini saranno uguali.

I termini $f(1,2)$ e $f(2,1)$, non unificano peró se f viene interpretato come un simbolo funzionale commutativo (ad esempio '+') gli oggetti identificati da quei due termini possono essere uguali. Come altro esempio, si considerino i termini $f(1,X)$ e $f(X, f(1,1))$. Essi non unificano, tuttavia, se f viene interpretato ad esempio come $\cup$ (unione insiemistica), $\vee$ (OR logico) e 1 come insieme universo o true, allora è del tutto lecito che quest'equazione abbia degli unificatori. (ad esempio, nel caso dell'OR, si avrebbe che $1 \vee X$ ha lo stesso significato di $X \vee 1 \vee 1$).

Quindi se l'unificabilitá garantisce che comunque si interpretino i simboli, due oggetti diventeranno in qualche modo uguali, la non unificabilitá non garantisce il contrario. L'esempio 2.20 mostra come l'interpretazione di Herbrand mantenga fortemente alcune proprietà proprie della unificabilità dei termini. Tale intuizione è confermata dal seguente teorema. Esso stabilisce che l'esistenza di un unificatore per un insieme di equazioni ne implica la soddisfacibilità in ogni interpretazione. Inoltre, esso dimostra che se un insieme di equazioni è soddisfacibile nell'universo di Herbrand allora lo è per qualsiasi interpretazione.

TEOREMA 3.20. *Sia C un sistema di equazioni, $vars(C) = \{X_1, \ldots, X_n\}$. Se $\mathcal{F}$ contiene almeno un simbolo di costante, i tre seguenti enunciati sono sono equivalenti:*

(1) *C è unificabile*

(2) $\models \exists X_1 \cdots X_n\, C$
(3) $\mathcal{H} \models \exists X_1 \cdots X_n\, C$

$1 \Rightarrow 2$. 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 $vars(C\theta) = \{Y_1, \cdots, Y_n\}$ e sia 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 $\mathcal{A} \models \exists X_1 \cdots X_n C$.

$[2 \Rightarrow 3]$ Banale, per definizione $\models \exists X_1 \cdots X_n(C)$ significa che la formula é soddisfatta per ogni interpretazione. Quindi anche per $\mathcal{H}$.

$[3 \Rightarrow 1]$ $\mathcal{H} \models \exists X_1 \cdots X_n(C)$ significa, per definizione, che esistono $t_1, \ldots, t_n$ 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 $(s = t)$ di C, $s\theta \equiv t\theta$, con $\theta = [X_1/t_1, \cdots, X_n/t_n]$. Il che significa che θ é un unificatore di C. □

Esempio 3.21. Poniamoci ora il problema di stabilire in quale modo si collegano logicamente le formule:

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

L'implicazione $(\leftarrow)$ è sempre vera per l'assioma dell'uguaglianza: *Per ogni formula φ:*

$$X = Y \to (\varphi \leftrightarrow \varphi[X/Y])$$

Invece, l'implicazione $(\rightarrow)$ non è sempre verificata. Questo si può vedere nel seguente esempio:

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

Avremo pertanto che:

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

Questo non è tuttavia del tutto soddisfacente: quello che si vorebbe ottenere è un ambiente in cui l'algoritmo Unify termina con successo se e solo se l'insieme di equazioni è soddisfacibile. Per questo Clark nel '78, riprendendo un lavoro di Mal'cev propose una teoria equazionale per cui l'algoritmo di unificazione risulta corretto e completo.

Definizione 3.22 (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 oggetto X diverso da ogni oggetto t che contenga X come sottotermine proprio.

Teorema 3.23. *Sia C un sistema di equazioni. Allora $CET \models \overrightarrow{\forall}(C \Leftrightarrow \mathsf{Unify}(C))$.*

5. E-unificazione

Abbiamo detto che due termini sono unificabili se esiste una sostituzione che li rende sintatticamente identici. In presenza di una teoria E che coinvolga i termini, potremmo pensare ad un concetto più vasto, 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":

(1) un modo potrebbe essere quello di verificare se esiste una sequenza finita di applicazioni di regole deducibili da E tali che $s \longrightarrow_* t$;
(2) un altro quello di partizionare l'universo di Herbrand in classi di equivalenza in accordo con E e stabilire pertanto se s e t appartengano alla stessa classe di equivalenza.

Non studieremo tali problematiche in teorie del prim'ordine qualsiasi, ma solo in una classe di teorie dette *equazionali*.

Esempio 3.24. 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 $b \cup a$. L'unificazione sintattica (Unify) direbbe di no. Tuttavia, nella teoria ACI si ha che i due termini sono equivalenti. Infatti $a \cup b$ si riscrive in $b \cup a$ per la regola desunta dall'assioma di commutatività C.

Similmente, $(a \cup b) \cup b \stackrel{ACI}{=} b \cup a$:

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

La nozione di E-unificabilità estende pertanto quella di unificabilità sintattica vista nei paragrafi precedenti. La sua generalità riapre delle domande risolte nel caso sintattico.

Esempio 3.25. Cerchiamo gli unificatori della seguente equazione:

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

Tale equazione ammette gli unificatori

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

(in realtà ne ammette anche molti altri, come vedremo nell'esempio 3.32). Si noti come i due unificatori siano tra loro indipendenti. Inoltre non appare possibile avere un unificatore più generale di nessuno dei due (e dunque nessuno più generale di entrambi). Parrebbe che, per questa teoria, descrivere tutte le soluzioni con un mgu, unico a meno di varianti, non sia possibile.

Inoltre, i seguenti:

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

sembrano essere infiniti ACI-unificatori del problema di partenza. Il numero di unificatori è infinito; esisterà un insieme se non unitario, per lo meno finito di unificatori in grado di descriverli tutti?

Formalizziamo ora alcuni concetti dell'E-unificazione.

DEFINIZIONE 3.26. 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 3.27. 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. Viene suggerita dalle precedenti definizioni una interpretazione

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

che manda 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:

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

DEFINIZIONE 3.28. Un *E-unificatore* di un sistema C è una sostituzione θ tale che

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

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

ESEMPIO 3.30. Sia E la teoria ACI e consideriamo il sistema: $X \cup a = X \cup b$.

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

Esisterà in questo caso un unificatore più generale di tutti e due? Consideriamo $\theta = [X/a \cup b \cup N]$ con N *nuova variabile*. Avremo che:

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

La presenza della variabile N, non presente nelle variabili iniziali del problema non ci permette di ottenere esattamente le sostituzioni (1) e (3) come istanze. Tuttavia è molto più

accettabile richiedere che la nozione di istanza sia ristretta ad un insieme di variabili (tipicamente l'insieme delle variabili del sistema) piuttosto che necessitare di insiemi infiniti di unificatori per collezionare le risposte. Ciò giustifica la $\mathcal{X}$ nella definizione sopra.

DEFINIZIONE 3.31. 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) $\forall \theta \in \mathcal{U}_E(C)$ 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:

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

ESEMPIO 3.32. Sia $X \cup Y = a \cup b$ come nell'Esempio 3.32. 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) $[X/a \cup b, Y/a \cup b], [X/a \cup b \cup a, Y/a \cup b], \ldots$

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

DEFINIZIONE 3.33. 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. $\mathcal{U}_E(C) \neq \emptyset$ (ammette almeno un E-unificatore) allora esiste $\mu\mathcal{U}_E(C)$ tale che $|\mu\mathcal{U}_E(C)| = 1$ (é rappresentabile con 1 m,g.u.).
ω: (finitario) se per ogni C t.c. $\mathcal{U}_E(C) \neq \emptyset$, esiste $\mu\mathcal{U}_E(C)$ di cardinalitá 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 cardinalitá infinita (ed esiste almeno un 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 3.34. Per ognuno dei quattro *tipi* si fornisca almeno un esempio di teoria. Nell'ultimo caso (tipo 0) si consiglia la lettura della letteratura (p.es. [**?**]).

5.1. ACI-Unificazione. Per concludere questo capitolo, analizziamo il problema decisionale relativamente alla teoria ACI. Consideriamo il problema di decisione prima nel caso in cui l'alfabeto contiene solo simboli di costante e poi nel caso in cui contenga simboli di funzione (almeno uno). Assumiamo pertanto che $\mathcal{F}$ contenga solo simboli di costante, oltre a $\cup$ di aritá 2. 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 assegno il termine costituito dall'unione di tutte le costanti che trovo:

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

Tale sostituzione é un ACI-unificatore! In presenza di variabili in entrambi i termini, tale situazione può facilmente essere generalizzata. Cerchiamo pertanto di risolvere il problema di decisione di un sistema fatto 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 sse ogni costante in ℓ è presente anche in r.
(3) $vars(\ell) = \emptyset$ e $vars(r) \neq \emptyset$: si veda il punto precedente, con i ruoli di ℓ e r invertiti.
(4) $vars(\ell) = \emptyset$ e $vars(r) = \emptyset$: unifica se i termini ℓ e r contengono esattamente le stesse costanti.

Pertanto, sulla base del ragionamento sopra, il problema di decisione dell'unificazione ACI nel caso di segnatura con costanti ammette una soluzione in tempo polinomiale.

Vediamo come cambia la situazione quando nel linguaggio ammettiamo la presenza di simboli di funzione. 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 3 letterali, determinare se esiste un assegnamento di verità per le variabili che la renda soddisfacibile.*

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

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

Con il problema di unificazione:

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

si vincolano le variabili ad assumere valori mutamente esclusivi 0 o 1 (si pensi all'interpretazione insiemistica dei termini). Pertanto si avrà che Y_i è vincolata ad essere $\neg X_i$.

Il seguente problema di unificazione modella esattamente 3-SAT:

$$\{\{\{X_i\} \cup \{Y_i\}\} \cup \{\{X_i\} \cup \{Y_i\}\} \cup \{\{X_i\} \cup \{Y_i\}\} \cup$$
$$\{X_1\} \cup \{Y_2\} \cup \{X_3\} \cup \{0\}\} \cup \{\{Y_1\} \cup \{X_2\} \cup \{X_3\} \cup \{0\}\} \cup \{\{X_1\} \cup \{Y_2\} \cup \{Y_3\} \cup \{0\}\} =$$
$$\{\{0\} \cup \{1\}\}$$

Ogni soluzione del problema di 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 1) è NP-completo.

CHAPTER 4

Programmazione con clausole di Horn

DEFINIZIONE 4.1. Dato un alfabeto $\Sigma = \langle \Pi, \mathcal{F}, \mathcal{V} \rangle$

- Se $H, A_1, \ldots, A_n$ sono atomi, $n \geq 0$, allora $H \leftarrow A_1, \ldots, A_n$ è una *regola*. H è detta *testa* della regola. $A_1, \ldots, A_n$ invece è detto *corpo* della regola.
- Se $n = 0$, $H \leftarrow$ 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 $n = 0$, indichiamo con $\leftarrow \Box$ il goal vuoto.
- Un *programma* è un insieme finito di regole.

Cerchiamo di dare un significato alla Regola

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

La virgola corrisponde all'and logico ($\wedge$). Pertanto vorremmo indicare che quando $A_1, \ldots, A_n$ sono vere, lo dev'essere H. Poichè $P \rightarrow Q \equiv \neg P \vee Q$, allora $H \leftarrow A_1, \ldots, A_n$ è equivalente ad una disgiunzione di letterali con esattamente un letterale positivo:

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

Una disgiunzione di letterali viene chiamata clausola in logica, le clausole con al piú un letterale positivo sono dette di *Horn*. Una disgiunzione di letterali con esattamente un letterale positivo è detta *clausola definita*.

Se $\{X_1, \ldots, X_n\} = \mathit{vars}(H, A_1, \ldots, A_n)$, allora la clausola $H \leftarrow A_1, \ldots, A_n$ equivale a

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

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

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

1. Esempi di Programmi Definiti

1.1. Programmi con dominio vuoto—proposizionali. Si consideri il programma:

```
estate ←
caldo ← estate
caldo ← sole
sudato ← estate, caldo.
```

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

Una volta scritto il programma, si può farlo leggere all'interprete PROLOG 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 la lettera minuscola, mentre le variabili con la lettera maiuscola (oppure con il trattino _).

1.2. Programmi con dominio finito - database programming. Per rappresentare in PROLOG un albero genealogico come quello di Figura 1, è naturale usare il seguente insieme di fatti:

```
padre(a,b) ←
padre(a,c) ←
padre(b,d) ←
padre(b,e) ←
```

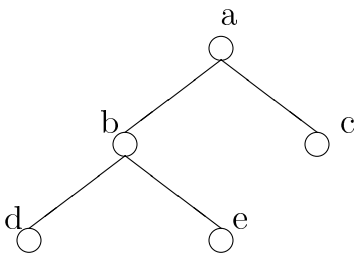


FIGURE 1. Albero genealogico

Come prima, si fa leggere il programma all'interprete prolog e poi si chiede se un goal è vero o falso:

```
?- padre(a,b).
yes
?- padre(a,e).
no
?- padre(a,Y).
yes Y=b ?
```

[1]SICStus Prolog nella configurazione *default* fornisce errore quando cerca di dimostrare un atomo costruito con un simbolo predicativo mai presente come testa di una regola del programma. Per ovviare a ciò è necessario dare il comando:

`prolog_flag(unknown, error, fail).`

oppure metterlo come *direttiva* (ovvero anticipato da `:-`) all'inizio del file contenente il programma stesso.

Quest'ultimo goal contiene la variabile Y. In questo modo si chiede non solo se è vero o falso bensí se c'è qualche figlio Y il cui padre è a. Il primo figlio trovato dall'interprete è b. Poi un punto di domanda chiede se si vogliono altre risposte, per vedere le quali si deve digitare il tasto "punto e virgola".

```
?- padre(X,c).
yes X=a
```

Questo goal chiede chi è il padre di c. Per avere tutte le coppie padri-figli si scrive

```
?- padre(X,Y).
yes X=a,Y=b ?  (si digita ;)
yes X=a,Y=c ?  ···
```

Inseriamo adesso qualche regola nel programma, definendo un nuovo predicato in maniera intensionale (ovvero non specificando elemento per elemento):

```
figlio(X,Y)←padre(Y,X).
```

Alla domanda

```
?- figlio(Y,b)
```

otteniamo come risposta `yes` $Y = d$; $Y = e$

Una regola del tipo:

```
nonno(X,Y)←padre(X,Z),padre(Z,Y).
```

si caratterizza per il fatto che il corpo contiene una variabile che non c'è nella testa. Si può intendere in due modi:

(1) X è nonno di Y se esiste un individuo Z tale che X è padre di Z e Z è padre di Y, oppure
(2) qualunque siano X, Y, Z, se l'individuo X è padre di Z e Z è padre di Y, allora X è nonno di Y.

```
?- nonno(b,d).
no
?- nonno(a,e).
yes
```

Cerchiamo i nipoti di a:

```
?- nonno(a,Y).
yes Y = d ?  (si digita ;)
yes Y = e ?  (si digita ;)
no
```

Finora non si è sfruttata la ricorsività del linguaggio Prolog. Si possono infatti scrivere formule del tipo:

```
antenato(X,Y) ← padre(X,Y).
antenato(X,Y) ← antenato(X,Z), padre(Z,Y).
```

Allora:

```
?-antenato(a,Y)
yes Y=b ?  ;
yes Y=c ?  ;
yes Y=d ?  ;
```

```
yes Y=e ?  ;
no
```

1.3. Programmi con dominio infinito. Supponiamo che $\mathcal{F}_P$,ovvero l' insieme dei simboli di funzione usati nel programma P contenga simboli di funzione ($ar \geq 1$).

In questo modo possiamo, ad esempio, definire il concetto di numerale:

```
num(0).
num(s(X)) ← num(X).
```

Nel fatto si definisce che zero è un numero, mentre nella regola si definisce che il successore di qualcosa è un numero se quel qualcosa è ancora un numero. Chiediamo all'interprete Prolog se $\underline{2}$ è un numerale:

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

La domanda naturalmente deve essere posta dopo aver fatto leggere all' interprete il programma. La risposta è chiaramente positiva.
Un goal del tipo:

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

ha come risposta:

```
yes Z=0 ?  (si digita ;)
yes Z=s(0) ?  (sigita ;)
yes Z=s(s(0)) ?  (si digita ;) ...
```

NOTA 4.2. La scrittura dei numerali in questo modo può risultare piuttosto noiosa. 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 $ssss0$. 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 (p. es., []).

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) :- nat(Y).`

1.4. Definizione di Funzioni. Ricordiamo che una funzione $f : A_1 \times \cdots A_n \longrightarrow B$ non è altro che un insieme di ennuple $\langle x_1, \ldots, x_n, y\rangle \in A_1 \times \cdots A_n \times B$. Poiché siamo in un contesto predicativo, nel definire le funzioni, cercheremo di caratterizzare l'insieme di

ennuple caratterizzanti. In altri termini, per definire una funzione n-aria, definiremo un predicato $n+1$-ario.

Per comodità indichiamo con $s^i(0)$ il termine $\underbrace{s(\cdots(s}_{i}(0)\cdots))$. Definiamo alcune funzioni sui naturali:

- Successore
  ```
  succ(X, s(X)).
  ```
- Somma
  ```
  plus(X, 0, X).
  plus(X, s(Y), s(Z)) :- plus(X, Y, Z).
  ```
  SI osservi che, attraverso il goal $\leftarrow plus(s(0), X, s^3(0))$ si ottiene l'inversa della somma, cioè la sottrazione $X = 3 - 1$.
- 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).
  ```
  Attraverso le interrogazioni:
  (1) $\leftarrow exp(X, s^3(0), s^8(0))$ si ottiene la radice cubica di 8,
  (2) $\leftarrow exp(s^2(0), Y, s^8(0))$ 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).
  ```

2. Semantica operazionale del Logic Programming

È basata sulla SLD derivazione (risoluzione) dove S sta per *Selection* function, L sta per *Linear* resolution e D sta per *Definite* clauses.

DEFINIZIONE 4.3. Dato un atomo A_i ed una clausola $C \equiv H \leftarrow B_1, B_2, \ldots, B_n$ in un programma P, si dice che C è applicabile ad A_i se presa comunque una variante $H' \leftarrow B'_1, B'_2, \ldots, B'_n$ di C con variabili non occorrenti in A_i esiste $\mathsf{Unify}(A_i = H') \neq \mathsf{false}$.

La notazione $\mathsf{Unify}(A_i = H')$ non è del tutto corretta. A_i e H' sono atomi e non termini. Possiamo tuttavia, per semplicità notazionale, pensare ad essi come a termini e permetterci questo abuso di notazione.

Il passo di SLD derivazione è il punto chiave del funzionamento dell'interprete del logic programming.

DEFINIZIONE 4.4.

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

Il passo SLD seleziona un atomo (in questo esempio è stato scelto l'i-esimo atomo) ed una clausola C applicabile ad A_i. C viene ridenominata con variabili non presenti nel goal. Il passo SLD calcola dunque l'mgu di A_i con la testa della clausola ridenominata H e restituisce

un nuovo goal, detto la SLD *risolvente* di $\leftarrow A_1, \ldots, A_n$ e $H \leftarrow B_1, \ldots, B_m$ rispetto ad A_i, con mgu θ.

Ci sono diversi punti in cui possono essere fatte delle scelte per la SLD risoluzione:

(1) Scelta del letterale del goal (*Selection rule*)
(2) Scelta della clausola di input (*Computation rule*)
(3) Scelta della variante e variabili nuove
(4) Scelta dell'mgu.

Dopo aver fatto queste scelte si tratta di effettuare deterministicamente un rimpiazzamento (al posto di A_i si mette la variante) e una istanziazione (parte finale con sostituzione θ). Tra tutte le cose sopra, come vedremo, la cosa più difficile da realizzare è la parte di unificazione che abbiamo studiato nel Capitolo 3.

Il passo suddetto può essere sintetizzato con la seguente notazione: $G \overset{\theta}{\Longrightarrow}_C G'$, con G' goal ottenuto dal passo. Solitamente C è la clausola come appare nel programma (senza dunque esplicitare la ridenominazione).

Esempio 4.5. Supponiamo di avere a disposizione il seguente programma:

$$\mathsf{num(s(X))} \leftarrow \mathsf{num(X)}$$

e di voler operare sul goal G: $\leftarrow \mathsf{num(s(0))}$.

Poiché il programma consta di una sola regola, è chiaro che la clausola che c'è è l'unica clausola che si può prendere in considerazione e che il passo di SLD-derivazione sarà il seguente:

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

Definizione 4.6. 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:

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

che può essere finita o infinita e t.c.:

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

Per "massimale" si deve intendere che qualora esista almeno una clausola che unifica con il goal dato, l'unificazione deve essere operata sul letterale preso in considerazione e quindi il passo di SLD-derivazione eseguito.

Definizione 4.7. La *lunghezza* di una SLD-derivazione è il numero di passi eseguiti durante la derivazione (può essere finita o infinita).

Si osservi che se una SLD-derivazione ha lunghezza n finita allora vale uno dei seguenti casi:

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

DEFINIZIONE 4.8. Sia data una SLD-derivazione finita del tipo $\xi \equiv G \overset{\theta_1}{\Longrightarrow}_{C_1} G_1 \overset{\theta_2}{\Longrightarrow}_{C_2} G_2 \overset{\theta_3}{\Longrightarrow}_{C_3} \cdots \overset{\theta_n}{\Longrightarrow}_{C_n} G_n$. Si dice che ξ è di *successo* se G_n è $\leftarrow \square$. In questo caso la sostituzione $\theta_1 \circ \theta_2 \circ \ldots \circ \theta_n \mid_{vars(G)}$ si dice (sostituzione di) *risposta calcolata (cas).*

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

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

ESEMPIO 4.9. Consideriamo ora alcune derivazioni a partire dal seguente programma P:

```
padre(a, b)    ←
padre(a, c)    ←
padre(b, d)    ←
padre(b, e)    ←
figlio(X, Y)   ← padre(Y, X)
nonno(X, Y)    ← padre(X, Z), padre(Z, Y)
antenato(X, Y) ← padre(X, Y)
antenato(X, Y) ← antenato(X, Z), padre(Z, Y)
```

(1) Sia $G \equiv \leftarrow nonno(a, d)$.

$$\frac{\leftarrow nonno(a,d) \quad nonno(X_1,Y_1) \leftarrow padre(X_1,Z_1), padre(Z_1,Y_1)}{\leftarrow (padre(X_1,Z_1), padre(Z_1,Y_1))\theta_1 = \leftarrow padre(a,Z_1), padre(Z_1,d)}$$

con $\theta_1 = [X_1/a, Y_1/d]$ unificatore calcolato tra l'atomo `nonno(a, d)` e la testa della clausola `nonno(`X_1`,` Y_1`)`. Proseguiamo con altri passi possibili.

$$\frac{\leftarrow padre(a,Z_1), padre(Z_1,d) \qquad padre(a,b) \leftarrow}{\leftarrow padre(b,d)} \qquad \theta_2 = [Z_1/b]$$

$$\frac{\leftarrow padre(b,d) \qquad padre(b,d) \leftarrow}{\leftarrow \square} \qquad \theta_3 = \varepsilon$$

La derivazione è finita e di successo. La risposta calcolata è: $\theta_1\theta_2\theta_3 \mid_{vars(G)} = [X_1/a, Y_1/d, Z_1/b] \mid_{\emptyset} = \varepsilon$.

(2) Consideriamo il goal G $\equiv \leftarrow nonno(X, d)$.

$$\frac{\leftarrow nonno(X,d) \qquad nonno(X_1,Y_1) \leftarrow padre(X_1,Z_1), padre(Z_1,Y_1)}{\leftarrow padre(X_1,Z_1), padre(Z_1,d)}$$

con $\theta_1 = [X/X_1, Y_1/d]$.

$$\frac{\leftarrow padre(X_1,Z_1), padre(Z_1,d) \qquad padre(a,b) \leftarrow}{\leftarrow padre(b,d)} \qquad \theta_2 = [X_1/a, Z_1/b]$$

$$\frac{\leftarrow padre(b,d) \qquad padre(b,d) \leftarrow}{\leftarrow \square} \qquad \theta_3 = \varepsilon$$

La derivazione è chiaramente finita e di successo, quindi finita. La risposta calcolata è: $[X/a, X_1/a, Y_1/d, Z_1/b] \mid_X = [X/a]$.

ESERCIZIO 4.10. Supponiamo che al primo passo la sostituzione restituita da Unify fosse $\theta_1 = [X_1/X, Y_1/d]$. Verificare che si ottiene la stessa risposta calcolata.

(3) Consideriamo ora lo stesso goal del caso precedente $\leftarrow nonno(X, d)$, ma vediamo come la scelta di clausole diverse pregiudichi la derivazione.

$$\frac{\leftarrow nonno(X,d) \qquad nonno(X_1,Y_1) \leftarrow padre(X_1,Z_1), padre(Z_1,Y_1)}{\leftarrow padre(X,Z_1), padre(Z_1,d)}$$

con l'unificatore $\theta_1 = [X_1/X, Y_1/d]$.

$$\frac{\leftarrow padre(X,Z_1), padre(Z_1,d) \qquad padre(a,c) \leftarrow}{\leftarrow padre(c,d)} \quad \theta_2 = [X/a, Z_1/c]$$

Come si vede, a questo punto non esiste alcuna clausola del programma che permetta di unificare con $\leftarrow padre(c, d)$ e quindi si ha una derivazione finita ma di fallimento.

(4) Supponiamo ora che il goal da considerare sia $G \equiv\leftarrow antenato(X, Y)$ (tutti gli antenati X di Y).[2]

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

con $\theta_1 = [X/X_1, Y/Y_1]$. Supponiamo di considerare l'atomo $ant(Z_1, Y_1)$ del nuovo goal.

$$\frac{\leftarrow padre(X_1,Z_1), ant(Z_1,Y_1) \quad ant(X_2,Y_2) \leftarrow padre(X_2,Z_2), ant(Z_2,Y_2)}{\leftarrow padre(X_1,X_2), padre(X_2,Z_2), ant(Z_2,Y_2)}$$

con $\theta_2 = [Z_1/X_2, Y_1/Y_2]$.
Come si può notare, scegliendo la clausola `antenato(X, Y)` del programma si hanno goals sempre più complessi. La derivazione che stiamo costruendo è infinita.

3. Indipendenza del non-determinismo nell'SLD

3.1. Indipendenza dal renaming della Clausula. Relativamente al programma dell'esempio 4.9 consideriamo il goal:

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

e costruiamo due alberi di derivazione ξ e ξ' che si differenziano per la scelta della rinomina delle variabili presenti nelle clausole.

Primo Albero di derivazione:

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

[2]Per comodità di impaginazione, il predicato `antenato` viene abbreviato in `ant`.

E' una derivazione di successo con risposta calcolata $[A/a, B/b]$

Secondo Albero di derivazione:

$$\dfrac{\dfrac{\leftarrow nonno(A,B) \qquad nonno(X_2,Y_2) \leftarrow padre(X_2,Z_2), padre(Z_2,Y_2)}{[A/X_1, B/Y_1] \qquad \dfrac{\leftarrow padre(X_2,Z_2), padre(Z_2,Y_2) \qquad padre(a,b)}{[X_2/a, Z_2/b] \qquad \dfrac{\leftarrow padre(a,Y_2) \qquad padre(b,d)}{\leftarrow \square}}}}$$

La scelta della rinomina ha condotto, in questo caso, alla stessa risposta calcolata.

Si prenda ora come esempio il programma

```
member( X , [ X | Z ] ).
member( X , [ Y | Z ] :-
   member( X , Z ).
```

Con due derivazioni identiche salvo la rinomina delle clausole, a partire dal goal `:- member( A , B )` possiamo ottenere le risposte calcolate:

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

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.

3.2. Indipendenza della scelta dell'mgu. Consideriamo nuovamente il goal $\leftarrow nonno(A,B)$ e la seconda tra le due derivazioni analizzate sopra. Calcoliamo un diverso mgu al primo passo.

$$\dfrac{\dfrac{\leftarrow nonno(A,B) \qquad nonno(X_2,Y_2) \leftarrow padre(X_2,Z_2), padre(Z_2,Y_2)}{[X_2/A, Y_2/B] \qquad \dfrac{\leftarrow padre(A,Z_2), padre(Z_2,B) \qquad padre(a,b)}{[A/a, Z_2/b] \qquad \dfrac{\leftarrow padre(b,B) \qquad padre(b,d)}{\leftarrow \square}}}}$$

Otteniamo una risposta identica alla precedente.

Più in generale, si osserva che diverse scelte portano alle stesse risposte calcolate o quantomeno a varianti. Cerchiamo di formalizzare questa intuzione.

DEFINIZIONE 4.11. Consideriamo le due SLD-derivazioni:

$$\begin{aligned} \xi &= G_0 \overset{\theta_1}{\Rightarrow}_{c_1} G_1 \overset{\theta_2}{\Rightarrow}_{c_2} G_2 \cdots \\ \xi' &= G'_0 \overset{\theta'_1}{\Rightarrow}_{c_1} G'_1 \overset{\theta'_2}{\Rightarrow}_{c_2} G_2 \cdots \end{aligned}$$

(si osservi che le clausole utilizzate sono le stesse nelle due derivazioni). La derivazione ξ' è un *lift* di ξ se

(1) ξ ha lunghezza minore o uguale a ξ' (se ξ è infinita, allora lo è anche ξ'),
(2) G_0 è un'istanza di G'_0.
(3) per $i \geq 0$ in G_i e G'_i vengono selezionati gli atomi nella stessa posizione.

ESEMPIO 4.12. Consideriamo le derivazioni:

$$\begin{aligned} \xi &= \leftarrow p(a) \overset{[X'/a]}{\Longrightarrow}_{p(X)\leftarrow q(X)} q(a) \overset{\varepsilon}{\Longrightarrow}_{q(a)\leftarrow} \square \\ \xi' &= \leftarrow p(X) \overset{[X'/X]}{\Longrightarrow}_{p(X)\leftarrow q(X)} q(X) \overset{[X/a]}{\Longrightarrow}_{q(a)\leftarrow} \square \\ \xi'' &= \leftarrow p(V) \overset{[V/X']}{\Longrightarrow}_{p(X)\leftarrow q(X)} q(X') \overset{[X'/a]}{\Longrightarrow}_{q(a)\leftarrow} \square \end{aligned}$$

ξ' e ξ'' sono entrambe lift di ξ. ξ' è inoltre lift di ξ'' e viceversa.

LEMMA 4.13. *Siano ξ e ξ' due derivazioni come nella definizione 4.11. Sia ξ' lift di ξ. Allora, per ogni $i \geq 0$ si ha che:*

(1) *G_i ha lo stesso numero di atomi di G'_i.*
(2) *G_i è istanza di G'_i.*
(3) *Se ξ e ξ' sono di successo e lunghe n, allora $G_i\theta_{i+1}\cdots\theta_n$ è istanza di $G'_i\theta'_{i+1}\cdots\theta'_n$.*

PROOF. La dimostrazione del punto (2) implica il punto (1). Il punto (3) è lasciato per esercizio (la dimostrazione è simile a quella del punto (2)). Dimostriamo il punto (2) per induzione su $i \geq 0$.
Base: $i = 0$ Per ipotesi, ξ' è lift di ξ. Allora G_0 è istanza di G'_0 per definizione.
Passo: Assumiamo G_i istanza di G'_i per ipotesi induttiva. Pertanto esiste una sostituzione μ tale che

$$G_i = G'_i\mu \tag{3.1}$$

Dobbiamo mostrare che G_{i+1} è istanza di G'_{i+1}.

Sia $H \leftarrow \bar{B}$ la clausola impiegata nel passo di derivazione. Sia $H' \leftarrow \bar{B}'$ la sua rinomina in ξ e $H'' \leftarrow \bar{B}''$ la sua rinomina in ξ'. Avremo che, per una qualche variante γ,

$$(H' \leftarrow \bar{B}') = (H'' \leftarrow \bar{B}'')\gamma \tag{3.2}$$

Siano A_k e A'_k gli atomi selezionati nelle derivazioni ξ e ξ' rispettivamente. Si ha che

$$A_k = A'_k\mu \tag{3.3}$$

Avremo inoltre che $\theta_{i+1} = mgu(A_k, H')$ mentre $\theta'_{i+1} = mgu(A'_k, H'')$.

G_{i+1} si ottiene, per definizione di passo di derivazione, da $(G_i, \bar{B}')\theta_{i+1}$ rimuovendo il letterale $A_k\theta_{i+1}$. Similmente, G'_{i+1} si ottiene da $(G'_i, \bar{B}'')\theta'_{i+1}$ rimuovendo il letterale $A'_k\theta_{i+1}$. Dimostreremo che $(G_i, \bar{B}')\theta_{i+1}$ è istanza di $(G'_i, \bar{B}'')\theta'_{i+1}$. Il risultato cercato sarà una conseguenza.

Ora, poiché $dom(\gamma)$ consiste di variabili nuove, da 3.1 e 3.2, ho che:

$$(G_i, \bar{B}')\theta_{i+1} = (G'_i, \bar{B}'')\mu\gamma\theta_{i+1} \tag{3.4}$$

Ci siamo ricondotti a dover mostrare che $\mu\gamma\theta_{i+1}$ è istanza di θ'_{i+1}.

Il non-determinismo insito in Unify ci assicura che

$$\mathsf{Unify}(A_k = H') = \mathsf{Unify}(A'_k = H'' \wedge \hat{\mu} \wedge \hat{\gamma}) \tag{3.5}$$

dove con $\hat{\mu}$ e $\hat{\gamma}$ si denotano i sistemi in forma risolta associati alle sostituzioni μ e γ. Pertanto θ_{i+1} che è mgu di $\mathsf{Unify}(A_k = H')$ sarà istanza di θ'_{i+1} che è mgu di $\mathsf{Unify}(A'_k = H'')$. □

DEFINIZIONE 4.14. Due SLD-derivazioni ξ e ξ':

$$\begin{aligned} \xi &= G_0 \overset{\theta_1}{\Rightarrow}_{c_1} G_1 \overset{\theta_2}{\Rightarrow}_{c_2} G_2 \cdots \\ \xi' &= G'_0 \overset{\theta'_1}{\Rightarrow}_{c_1} G'_1 \overset{\theta'_2}{\Rightarrow}_{c_2} G_2 \cdots \end{aligned}$$

sono *simili* se:

(1) ξ e ξ' hanno la stessa lunghezza;
(2) G_0 e G'_0 sono varianti;
(3) per ogni $i \geq 0$ in G_i e G'_i sono selezionati atomi nella stessa posizione.

ESEMPIO 4.15. Riferendoci all'esempio 4.12, ξ non é simile né a ξ' né a ξ''. ξ' é simile a ξ''.

TEOREMA 4.16. *Siano ξ e ξ' come in Def. 4.14 due derivazioni simili. Allora per ogni $i \geq 0$ si ha che G_i è variante di $G_i{}'$.*

PROOF. Se ξ e ξ' sono simili allora ξ è lift di ξ' e viceversa. Dal Lemma 4.13, punto (2) abbiamo dunque che per ogni $i \geq 0$ G_i è istanza di G'_i e viceversa. Per il Lemma sulle varianti (Lemma 2.35) G_i e G'_i sono varianti. □

TEOREMA 4.17. *Siano due derivazioni simili di successo per un goal G con risposta calcolata θ e θ'. Allora $G\theta$ e $G\theta'$ sono varianti.*

PROOF. Deriva dal Lemma 4.13, punto (3). □

I due seguenti risultati, non necessari agli scopi di questa sezione, servono per la prova di completezza della SLD-risoluzione. La loro dimostrazione non è molto diversa da quella di alcuni dei risultati visti in questa sezione.

TEOREMA 4.18 (Lifting). *Per ogni SLD-derivazione ξ di $P \cup \{G\eta\}$ esiste una SLD-derivazione di $P \cup G$ che è un lift di ξ.*

COROLLARIO 4.19. Per ogni SLD_derivazione ξ di successo di P∪Gη con risposta θ esiste una SLD_derivazione per ξ' di successo di P∪G con risposta θ' tale che:

(1) ξ'è lift di ξ e ha la stessa lunghezza
(2) $G\theta'$ é piú generale di $G\theta$

Ad esempio, dal fatto che `member(a,[a,b,c])` abbia una derivazione di successo e risposta calcolata nulla, questi risultati assicurano che `member(X,[a,b,c])` avrà una derivazione con successo e risposta calcolata più generale (ovvero $[X/a]$).

3.3. Scelta dell'atomo nel goal. Per prima cosa definiamo formalmente il concetto di regola di selezione.

DEFINIZIONE 4.20. Sia *IF (Initial Fragment)* l'insieme dei frammenti iniziali di tutte le SLD-derivazioni (dato un certo alfabeto) con ultimo goal non vuoto. R è una *regola di selezione* se é una funzione che, applicata a IF, indica esattamente un atomo nell'ultimo goal. R dunque si basa su tutta la storia della derivazione. Diciamo che una SLD-derivazione ξ è *via* R se tutte le selezioni di un atomo in essa sono in accordo con R.

Si osservi che:

- Ogni SLD-derivazione è via R per una qualche R (ad ogni passo avviene una scelta di un qualche atomo. La regola R la posso scrivere a posteriori).
- Se dimostriamo che vi è indipendenza dalla scelta della regola R, questo ci permette di scegliere 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 4.21. Consideriamo il goal $\leftarrow padre(X,Z), padre(Z,Y)$
Una prima derivazione, con risposta $[X/a, Y/d, Z/b]$, è:

$$\begin{array}{lll} [X/a,Z/b] & \underline{\leftarrow \underline{padre(X,Z)}, padre(Z,Y) \qquad \leftarrow padre(a,b)} & \\ [Y/d] & \underline{\leftarrow padre(b,Y) \qquad\qquad padre(b,d)} & \\ \leftarrow \square & & \end{array}$$

Una seconda, con la stessa risposta calcolata, è:

$$\begin{array}{lll} [Z/b,Y/d] & \underline{\leftarrow padre(X,Z), \underline{padre(Z,Y)} \qquad \leftarrow padre(b,d)} & \\ [X/a] & \underline{\leftarrow padre(X,b) \qquad\qquad padre(a,b)} & \\ \leftarrow \square & & \end{array}$$

Si osservi l'ordine inverso di applicazione delle medesime clausole.

LEMMA 4.22 (Switching). *Sia ξ una SLD-derivazione tale che:*

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

- *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:

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

- *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}.*

PROOF. Cerchiamo di esplicitare la parte che ci interessa della derivazione. Data una derivazione:

$$\begin{array}{lll} [\theta_{n+1} = mgu(A_i, A'_i)] & \underline{\leftarrow A_1, \ldots, A_i, \ldots, A_j, \ldots, A_n \qquad A'_i \leftarrow \overline{B_1}} & \\ [\theta_{n+2} = mgu(A_j\theta_{n+1}, A'_j)] & \underline{(\leftarrow A_1, \ldots, \overline{B}_1, \ldots, \overline{A}_j, \ldots, A_n)\theta_{n+1} \qquad A'_j \leftarrow \overline{B}_2} & \\ & \leftarrow (A_1, \ldots, \overline{B}_1, \ldots, \overline{B}_2, \ldots, A_n)\theta_{n+1}\theta_{n+2} \qquad \ldots & \end{array}$$

Dobbiamo dimostrare che ne esiste una della forma:

$$\begin{array}{lll} [\theta'_{n+1} = mgu(A_j, A'_j)] & \underline{\leftarrow A_1, \ldots, A_i, \ldots, A_j, \ldots, A_n \qquad A'_j \leftarrow \overline{B_2}} & \\ [\theta'_{n+2} = mgu(A_i\theta'_{n+1}, A'_i)] & \underline{(\leftarrow A_1, \ldots, \overline{A}_i, \ldots, \overline{B}_2, \ldots, A_n)\theta'_{n+1} \qquad A'_i \leftarrow \overline{B}_1} & \end{array}$$

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

tale che:

$$(3.6)\quad (A_1, \ldots, \overline{B}_1, \ldots, \overline{B}_2, \ldots, A_n)\theta_{n+1}\theta_{n+2} \;=\; (A_1, \ldots, \overline{B}_1, \ldots, \overline{B}_2, \ldots, A_n)\theta'_{n+1}\theta'_{n+2}$$

In altri termini che, relativamene alle variabili in gioco,

$$(3.7)\qquad \theta_{n+1}\theta_{n+2} \;=\; \theta'_{n+1}\theta'_{n+2}$$

Ora, pensiamo ai passi di unificazione effettuati nei due casi:

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

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

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

Sappiamo pertanto, dal Teorema 3.17, 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:

$$(3.8)\qquad \theta_{n+1}\theta_{n+2} \;=\; \theta'_{n+1}\theta'_{n+2}\gamma$$

Rimane da mostrare che Unify applicata due volte in ξ' può restituire esattamente $\theta_{n+1}\theta_{n+2}$ (anche se in fondo trovare la variante sarebbe sufficiente!). Ora, possiamo spezzare γ in γ' e γ'' a seconda che le variabili riguardino $vars(A'_i)$ e $vars(A_i) \setminus vars(A_j)\theta'_{n+1}$ e le altre rispettivamente. I due pezzi saranno a loro volta varianti. Avremo che

$$(3.9)\qquad \theta'_{n+1}\theta'_{n+2}\gamma = \theta'_{n+1}\gamma'\theta'_{n+2}\gamma''$$

Ora, $\theta'_{n+1}\gamma'$ sarà variante di θ'_{n+1} mentre $= \theta'_{n+2}\gamma''$ lo sarà di θ'_{n+2}. A questo punto sappiamo che la prima può essere restituita da Unify$(A_i = A'_i)$ mentre la seconda da Unify$(A_j\theta'_{n+1}\gamma' = A'_j)$. □

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

Teorema 4.23 (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.*

Proof. Consideriamo la derivazione ξ:

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

Cerchiamo di costruire ξ' via R. Assumo che fino a G_i la computazione sia via R. In G_i è selezionato l'atomo A_h mentre R direbbe di selezionare A_k. Lo stesso A_k viene poi

selezionato al passo G_{i+j}. L'idea è che con un numero finito j di applicazioni dello scambio porto la selezione di A_k al punto i-esimo, come vuole R. La correttezza è garantita dallo switching lemma.

Tale procedura va però iterata. Per dimostrare che l'algoritmo termina associo alla situazione sopra una coppia $\langle n-i, j\rangle$ che dice la lunghezza del tratto differente tra ξ e ξ' e la distanza fra l'applicazione dei due atomi A_h e A_k. La coppia rappresenta la valutazione della "differenza" fra la prima derivazione e quella che devo ottenere. La coppia $\langle 0, -\rangle$ è rappresenta la differenza nulla, e quindi il raggiungimento dell'obbiettivo; coppie lessicograficamente più prossime a $\langle 0, -\rangle$ rappresentano derivazioni piú vicine a ξ. Di conseguenza si possono presentare le due seguenti situazioni:

(1) $\langle 0, -\rangle$ quindi ξ era via R;
(2) $\langle i, j\rangle$ applico lo *Switching Lemma* ottenendo ξ'' e una delle seguenti situazioni:
   (a) $\langle i, j-1\rangle$;
   (b) $\langle i-1, -\rangle$ se sono già su quello dopo G_i.

L'applicazione di un passo di calcolo porta da una coppia a un'altra lessicograficamente minore. Ciò dimostra che mi basta applicare un numero finito di volte lo *Switching Lemma*: l'algoritmo termina correttamente. □

Abbiamo dimostrato che la scelta di una regola di selezione non è significativa per quanto riguarda le derivazioni di successo. Ma per le derivazioni di fallimento vi è invece un risvolto computazionale: infatti il fallimento avviene quando l'atomo scelto non unifica con la testa di nessuna clausola del programma, e quindi prima trova questo atomo, prima la computazione termina. Per esempio supponiamo di avere il seguente goal, e che solo per l'ultimo atomo $p(a)$ non vi sia una clausola applicabile:

$$\leftarrow A_1, \ldots, A_{999}, p(a).$$

con regola *leftmost* devo attendere di computare tutti gli atomi prima di terminare, mentre con *rightmost* computo solo $p(a)$. Ne segue 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.

3.4. Indipendenza dalla scelta della clausola. Analizziamo ora l'ultimo punto di non-determinismo (Punto (b)): la scelta della clausola nel programma. Assumiamo che la regola di selezione R sia fissata.

Definizione 4.24. Un *albero SLD* 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 la risolvente di G e C rispetto ad A.

Le (eventuali) foglie dell'albero sono i goal vuoti oppure i goal di fallimento (atomi senza nessuna clausola applicabile).

Definizione 4.25. Un'albero SLD è:

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

Esempio 4.26. Supponiamo di avere il seguente programma:

C_1: `antenato(X,Y)`←`padre(X,Y).`
C_2: `antenato(X,Y)`←`antenato(X,Z),padre(Z,Y).`
C_3: ←`padre(a,b).`

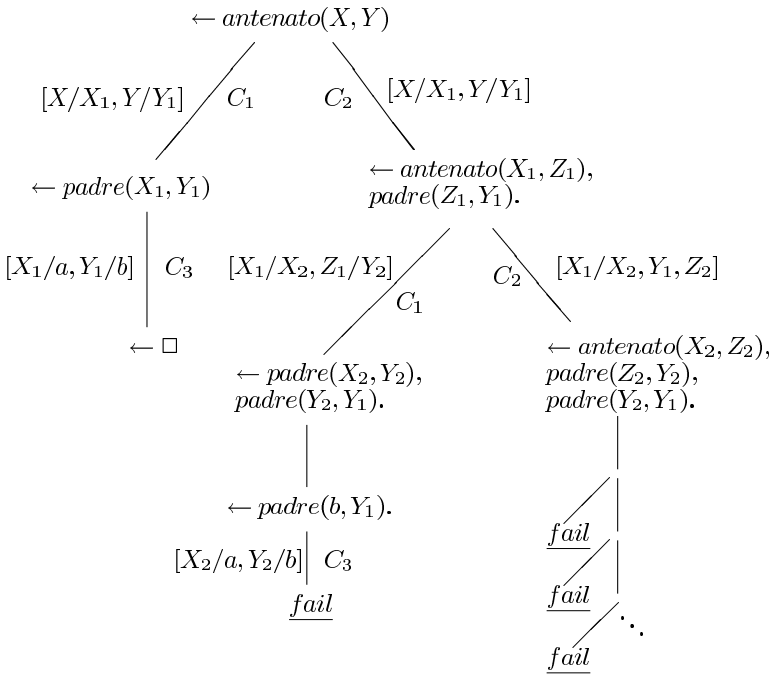


FIGURE 2. Albero SLD con $R \equiv Leftmost$

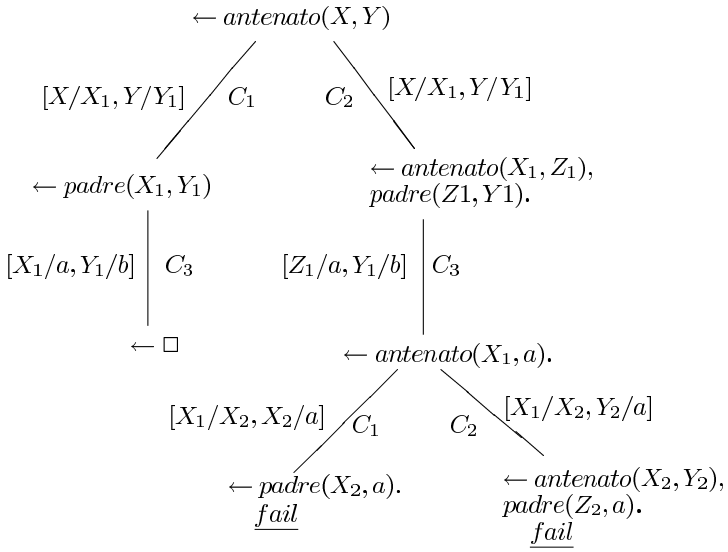


FIGURE 3. Albero SLD con $R \equiv Rightmost$

Costruiamo l'albero SLD prima con regola di selezione leftmost, poi con regola rightmost: come posso notare ottengo due alberi SLD di successo, entrambi con risposta calcolata $[X/a, Y/b]$, ma mentre il primo (fig. 2) è infinito il secondo (fig. 3) é finito.

NOTA 4.27. Essendo la regola di selezione di Prolog *leftmost*, con quel goal darebbe subito una risposta, ma chiedendo altre soluzioni (battendo ";") andrebbe in loop. Questo dipende dal fatto che nella clausola C_2 il predicato che opera la chiamata ricorsiva si trova al primo posto. L'esempio precedente suggerisce dunque un'altra regola di programmazione: *mai mettere un predicato ricorsivo al primo posto.* Nel significato logico-dichiarativo questa regola non ha senso, ma siccome l'interprete di Prolog utilizza algoritmi di risoluzione si deve tener conto anche dell'aspetto computazionale.

Vedremo tuttavia che gli alberi SLD possono essere diversi per regole di selezione diverse, ma le derivazioni di successo presenti negli alberi sono sempre le stesse.

Questo principio è sancito dal seguente teorema:

TEOREMA 4.28. *Fissata una regola di selezione R, se un albero SLD per $P \cup \{G\}$ è di successo allora tutti gli alberi SLD (al variare di R) saranno di successo.*

PROOF. Traccia: ogni SLD-derivazione è presente nell'albero e quelle di successo sono presenti con tutte le R. □

3.5. Costruzione dell'albero mediante back-tracking. Abbiamo visto che la presenza di derivazioni di successo in un albero è indipendente dalla regola di computazione. Tuttavia alcune danno alberi finiti e altri infiniti. Vediamo ora come venga costruito l'albero (se fosse costruito in ampiezza, le derivazioni di successo sarebbero sempre fornite tutte, ma vi sarebbero inaccettabili problemi di efficienza). La *Computation Rule* di Prolog è: *scegliere la prima clausola applicabile*, in caso di fallimento o di richiesta dell'utente (;) si faccia back-tracking. Proviamo per esempio a costruire l'albero della computazione del programma precedente con regola *Leftmost*.

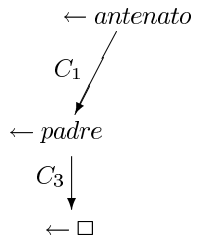


FIGURE 4. Primo passo della costruzione dell'albero SLD: trovata la prima soluzione, Prolog attende la richiesta di altre soluzioni.

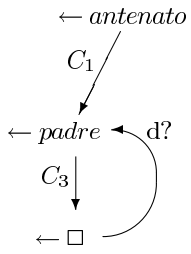


FIGURE 5. Backtracking al choice point precedente.

Dopo aver trovato la prima soluzione, Prolog attende la battitura del ";"(fig 4) fa *back-tracking* tornando al *choice point* precedente (ovvero al più vicino nodo dell'albero da cui

era possibile prendere un'altra strada), e chiedendosi se ha usato tutte le clausole unificabili con il goal (domanda d) (fig. 5). Evidentemente, per ogni nodo deve tenere una lista delle clausole già usate.

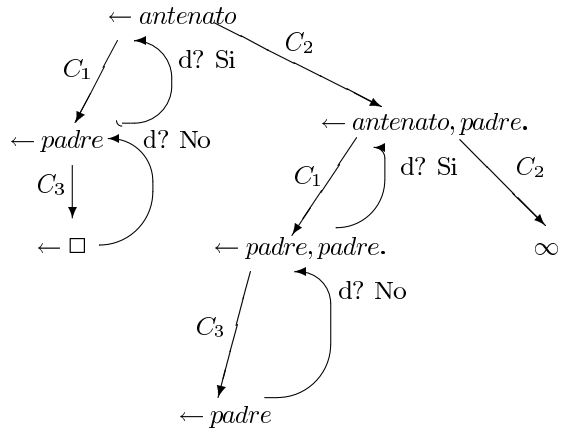


FIGURE 6. Albero SLD completo con Backtracking

Ogni volta che che non ci sono più clausole da usare torna al choice point precedente. Una nuova clausola usata invece, apre un nuovo ramo della computazione. Se l'albero costruito è finito il processo termina quando si esauriscono le clausole unificabili con la prima query (la radice dell'albero). In questo caso (fig.6) l'albero ha una computazione infinita; per ottenere almeno una soluzione è stato fondamentale mettere il caso base all'inizio della definizione induttiva.

ESEMPIO 4.29. Consideriamo il seguente programma Prolog P:

C_1: `r(b).`
C_2: `p(a).`
C_3: `p(b).`
C_4: `q(X):-p(X),r(X).`
C_5: `q(X):-q'(a).`

e costruiamo l'albero SLD per il goal $\leftarrow q(X)$.

ESEMPIO 4.30. Si consideri il programma P:

C_1: `num(0).`
C_2: `num(s(X)):-num(X).`

con il goal $\leftarrow num(X)$ restituisce come risposta "tutti" i naturali: ad ogni ";" costruisce un nuovo ramo di albero e da una nuova risposta.

4. Semantica Osservazionale dei programmi definiti

L'aver definito la procedura risolutiva ci permette di dare un significato operazionale ad un programma definito. In particolare si potremme definire come semantica operazionale di un programma definito, per ogni goal nel linguaggio del programma, l'intero albero di SLD-derivazione associato. Le semantiche descritte in questa sezione si possono vedere come approssimazioni di tale semantica.

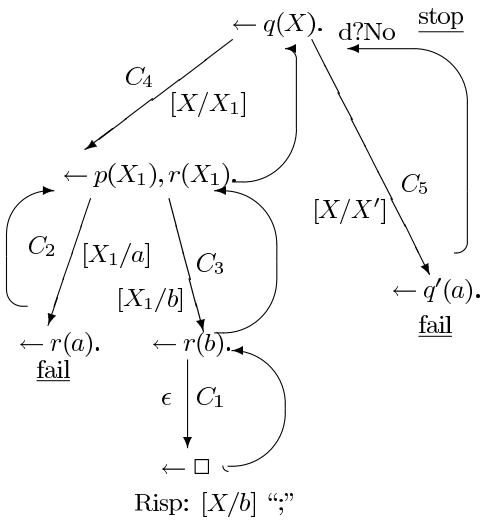


FIGURE 7. Albero SLD finito del programma all'esempio 4.29

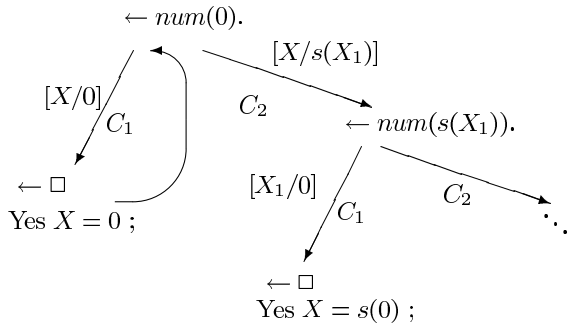


FIGURE 8. Albero SLD del programma all'esempio 4.30

DEFINIZIONE 4.31. Sia P un programma definito. Si dice osservabile di P l'insieme:

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

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. Gli *atomi ground* useranno come simboli l'insieme Π_P dei simboli di predicato presenti in P e l'insieme $\mathcal{F}_P$ dei simboli di costante e di funzione presenti in P. Qualora non vi fosse nessun simbolo di costante in $\mathcal{F}_P$, lo aggiungiamo.

ESEMPIO 4.32. 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 loop se gli si fornisce il goal $\leftarrow p(a)$. L'implementazione della SLD-risoluzione in Prolog è pertanto incompleta rispetto alla semantica osservazionale Oss.

ESERCIZIO 4.33. Si descriva $Oss(P)$ per i programmi seguenti:

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

(2) p(a). q(X) :− p(X). p(X) :− q(X).

La semantica operazionale Oss può essere considerata poco precisa per classificare programmi equivalenti. Ad esempio consideriamo:

(1) p(a). q(a).
(2) p(a). q(a). q(X).
(3) p(a). q(X).

L'universo di Herbrand dei tre i programmi è lo stesso e costituito dall'insieme contente solo la costante $\{a\}$. Gli atomi costruibili su tale universo sono: $\leftarrow p(a)$ ed $\leftarrow q(a)$. Per tutti e tre i programmi l'osservabile Oss è l'insieme $\{p(a), q(a)\}$. Questo significa che per la semantica specificata i tre programmi sono equivalenti. Tuttavia i programmi si comportano in modo diverso al goal $\leftarrow p(X)$. Hanno pertanto un diverso comportamento operazionale.

Consideriamo invece la semantica definita come segue:

DEFINIZIONE 4.34.

$$O_2(P) = \{A : \begin{array}{l} A \text{ è atomo e esiste una SLD derivazione per } \leftarrow A, \\ \text{con risposta calcolata } \varepsilon \text{ o una sua variante}\} \end{array}$$

è la semantica dell'*insieme di successo non ground*.

Per i tre programmi sopra, i goals da testare diventano:

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

L'osservabile dei tre programmi ora è rispettivamente:

(1) $\{p(a). \quad q(a).\}$
(2) $\{p(a). \quad q(a). \quad q(X).\}$
(3) $\{p(a). \quad q(a). \quad q(X).\}$

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

Un'ulteriore semantica è:

DEFINIZIONE 4.35. La *semantica di risposta calcolata* di un programma P è definita:

$$O_3(P) = \{A : \begin{array}{l} A \text{ è atomo e esiste } p \in \Pi, ar(p) = n \text{ ed} \\ \text{esistono } X_1, \ldots, X_n \in \mathcal{V} \text{ distinte t.c.} \\ \text{esiste una derivazione per } \leftarrow p(X_1, \ldots, X_n) \text{ con c.a.s. } \theta \text{ e} \\ A = p(X_1, \ldots, X_n)\theta\} \end{array}$$

I possibili goals da testare sono ora solamente

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

Per il primo programma $O_3(P) = \{p(a), q(a)\}$, per il secondo $O_3(P) = \{p(a), q(a), q(X)\}$, infine per il terzo è $O_3(P) = \{p(a), q(X)\}$. 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.

Tuttavia, per ragioni che diventeranno evidenti nelle prossime sezioni, la semantica osservazionale di un programma definito viene usualmente descritta dall'insieme Oss.

5. Turing Completezza di Prolog

Dimostreremo in questa sezione che mediante programmi di clausole definite e usando la SLD-derivazione, si dispone di un formalismo equivalente a quello della Macchina di Turing. Per far ciò, iniziamo dimostrando che nel formalismo Prolog si possono costruire funzioni ricorsive primitive e le funzioni parziali ricorsive.

TEOREMA 4.36. *Se una funzione $f : \mathbb{N}^n \longrightarrow \mathbb{N}$ è ricorsiva (parziale), allora f è definibile da un programma definito e calcolabile via SLD-risoluzione.*

PROOF. Dobbiamo saper definire le funzioni di base:

Funzione costante zero: $zero(X_1, \ldots, X_n, 0)$.
Funzione successore: $succ(X, s(X))$.
Funzione di proiezione: Per ogni n e $i \leq n$ possiamo definire:

$$pi_n_i(X_1, \ldots, X_n, X_i).$$

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.

Composizione: Supponiamo di aver definito le funzioni $g_1, \ldots, g_n, f$ mediante i predicati:

$$p_g_1(X_1, \ldots, X_m, Y_1), \ldots, p_g_n(X_1, \ldots, X_m, Y_m)$$

e la funzione:

$$p_f(Y_1, \ldots, Y_n, Y)$$

Definiamo la funzione h che le compone:

$$\begin{array}{rcl} p_h(X_1, \ldots, X_m, Y) & \leftarrow & p_g_1(X_1, \ldots, X_m, Y_1), \\ & & \vdots \\ & & p_g_n(X_1, \ldots, X_m, Y_n), \\ & & p_f(Y_1, \ldots, Y_n, Y). \end{array}$$

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

$$p_g(X_1, \ldots, X_n, Y) \text{ e } p_f(X_1, \ldots, X_n, Y_1, Y)$$

Definiamo, per ricorsione primitiva, la funzione h:

$$\begin{array}{rcl} p_h(X_1, \ldots, X_n, 0, Y) & \leftarrow & p_g(X_1, \ldots, X_n, Y). \\ p_h(X_1, \ldots, X_n, s(X), Y) & \leftarrow & p_h(X_1, \ldots, X_n, X, Y_1), p_f(X_1, \ldots, X_n, Y_1, Y). \end{array}$$

μ-operatore di minimalizzazione: Supponiamo di aver definito la funzione f mediante il predicato p_f. Dobbiamo definire la funzione h tale che $h(X_1, \ldots, X_n) = \mu Y\,(f(X_1, \ldots, X_n, Y) = 0)$.

$$\begin{array}{rcl} p_h(X_1, \ldots, X_n, 0) & \leftarrow & p_f(X_1, \ldots, X_n, 0, 0). \\ p_h(X_1, \ldots, X_n, s(Y)) & \leftarrow & p_f(X_1, \ldots, X_n, s(Y), 0), \\ & & mai_prima(X_1, \ldots, X_n, Y). \\ mai_prima(X_1, \ldots, X_n, 0) & \leftarrow & p_f(X_1, \ldots, X_n, 0, s(_)) \\ mai_prima(X_1, \ldots, X_n, s(Y)) & \leftarrow & p_f(X_1, \ldots, X_n, s(Y), s(_)), \\ & & mai_prima(X_1, \ldots, X_n, Y). \end{array}$$

□

Dunque programmi definiti + SLD-derivazione sono almeno espressivi quanto la macchina di Turing. Mostriamo l'implicazione inversa.

TEOREMA 4.37. *Sia $f : \mathbb{N}^n \longrightarrow \mathbb{N}$ una funzione calcolata da un programma definito P con SLD-risoluzione. Allora f è Turing calcolabile.*

PROOF. Se ci fidiamo della tesi di Church non è necessario dire altro. Dato che f è calcolabile, f è turing calcolabile.

Volendo dimostrare altrimenti il teorema, sia $f : \mathbb{N}^n \longrightarrow \mathbb{N}$ definita da un programma P. Ciò significa che, $p_f(X_1, \ldots, X_n, Y) \in Oss(P)$ se e solo se $Y = f(X_1, \ldots, X_n)$. Dire se $p_f(X_1, \ldots, X_n, Y) \in Oss(P)$ può essere fatto con una visita in ampiezza dell'albero SLD. Ciò è r.e.. □

6. Semantica logica (modellistica) di programmi definiti

6.1. Modelli di Herbrand. Nella definizione di pre-interpretazione di Herbrand (Def. 2.18) il dominio e l'interpretazione dei termini sono fissati univocamente. Nessuna posizione viene invece presa riguardo i simboli predicativi.

DEFINIZIONE 4.38. 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. Pertanto alla preinterpretazione H_Σ è sufficiente aggiungere un insieme di atomi veri

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

Poiché dominio e interpretazione dei termini è prefissata, l'insieme I da solo rappresenta univocamente una interpretazione.

L'insieme

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

è detto *base di Herbrand*. Se $\Pi, \mathcal{F}$ sono desunti da un programma/teoria P allora si indcherà semplicemente con $\mathcal{B}_P$.

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

Ricordiamo che, data una teoria T o un programma P, è possibile dedurre gli insiemi associati $\mathcal{F}$ e Π.

ESEMPIO 4.39. Dato il programma P

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

abbiamo $\mathcal{F}_P = \{a_{/0}, b_{/0}, f_{/1}\}$ e $\Pi_P = \{p, q, r\}$.

Data la teoria T:

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

abbiamo $\mathcal{F}_T = \{0_{/0}$ e $\Pi_T = \{<, =\}$.

Abbiamo, al solito, implicitamente assunto che $\mathcal{F}$ contenga sempre almeno un simbolo di costante; in caso contrario se ne dovrà aggiungere almeno uno, altrimenti l'universo di Herbrand sarebbe vuoto.

ESEMPIO 4.40. Dato il programma P:

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

Si ha che $\Pi_P = \{p, q, r\}, \mathcal{F}_P = \{a, b\}, T(\mathcal{F}_P) = \{a, b\}, \mathcal{B}_P = \{p(a), p(b), q(a), q(b), r(a), r(b)\}$.

Una possibile interpretazione di Herbrand è $I = \{p(a), r(b), q(b)\}$. I non è un modello di P in quanto l'ultima clausola non è verificata.

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

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

$I_4 = \{p(a), r(b), q(a)\}$ è ancora modello e sembra essere un modello *minimo*.

ESEMPIO 4.41. 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) $I_1 = \emptyset$ ma non è un modello di T
(2) $I_2 = B_T = \{p(a)\}$, ma l'unico elemento del dominio è a, quindi nemmeno questo è un modello di T.

T non ammette dunque modelli di Herbrand. È possibile però costruire un modello per T se si pone il dominio pari a $\{a, b\}$ e $I = \{p(a)\}$.

6.2. Teorie di clausole. L'esistenza di un modello non implica in generale quella di un modello di Herbrand (si veda Esempio 4.41). Se una teoria T è di tipo particolare, ovvero un insieme di *clausole*, vale invece il seguente risultato:

TEOREMA 4.42. *Se T è un insieme di clausole soddisfacibile allora ha modelli di Herbrand.*

PROOF. Sia $\mathcal{D} = \langle D, I^{\mathcal{D}} \rangle$ un modello di T. Costruiamo un'interpretazione $\mathcal{D}' = \langle D', I' \rangle$ dove:

- $D' = T(\mathcal{F})$ con F insieme di simboli di funzioni e di costanti usati in T (più eventualmente un simbolo di costante "fittizio")
- $I' = \{p(t_1, \ldots, t_n) : p \in \Pi, t_1, \ldots, t_n \in T(\mathcal{F}), \mathcal{D} \models p(t_1, \ldots, t_n)\}$

$\mathcal{D}'$ è un modello di Herbrand T. □

COROLLARIO 4.43. Sia T un insieme di clausole. T è insoddisfacibile se e solo se T non ammette modelli di Herbrand.

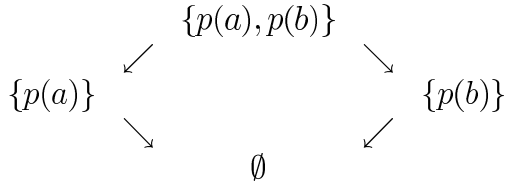
PROOF. ($\rightarrow$) Se T ammette modelli di Herbrand allora ammette modelli, quindi è soddisfacibile.
($\leftarrow$) se T è soddisfacibile allora ammette modelli, quindi per il teorema 4.42 ammette modelli di Herbrand. □

TEOREMA 4.44. *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.*

PROOF. ($\rightarrow$) Per definizione: $T \models A$, cioè A è vero in tutti i modelli di T, in particolare nei modelli di Herbrand.
($\leftarrow$) A è vero in tutti i modelli di Herbrand, quindi $T \cup \{\neg A\}$ non ha modelli di Herbrand, dunque $T \cup \{\neg A\}$ è insoddisfacibile perciò (Lemma 2.16) $T \models A$. □

COROLLARIO 4.45. 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.

ESEMPIO 4.46. Dati $T = \{p(a) \vee p(b)\}$ e $B_P = \{p(a), p(b)\}$ possiamo costruire più modelli di Herbrand:



Si osservi che:

(1) L'insieme di modelli di Herbrand mostrati in figura forma un reticolo $\langle \mathcal{P}(B_P), \subseteq \rangle$.
(2) Non esiste un modello minimo, in quanto l'inf. del reticolo, che è l'intersezione di $\{p(a)\}$ e $\{p(b)\}$ non è un modello di T.
(3) L'intersezione di due modelli non è necessariamente un modello.

ESEMPIO 4.47. Si consideri T definita come:

$$\begin{array}{ccccc} (p(a) \vee p(b)) & \wedge & (\neg p(a) \vee p(b)) & \wedge \\ (p(a) \vee \neg p(b)) & \wedge & (\neg p(a) \vee \neg p(b)) \end{array}$$

E' una teoria fatta di clausole di Horn ma è insoddisfacibile!

6.3. Clausole Definite. I due esempi *negativi* 4.46 e 4.47 riguardavano teorie fatte da clausole ma non di Horn (non definite). Studiamo cosa succede per programmi definiti.

TEOREMA 4.48. *Sia dato un programma di clausole definite P. La base di Herbrand B_P è* sempre *un modello per P.*

PROOF. Le clausole in P sono implicazioni della forma:

$$H \leftarrow B_1 \wedge \cdots \wedge B_n$$

Per ogni assegnamento delle variabili H risulta sempre vero e dunque l'implicazione è vera a sua volta. □

TEOREMA 4.49. *Sia P un programma di clausole definite; siano $\mathcal{D}_1, \mathcal{D}_2, \ldots$ modelli di Herbrand di P. Allora $\bigcap_{i \geq 1} \mathcal{D}_i$ è un modello di Herbrand di P.*

PROOF. Consideriamo un'istanza ground di una clausola di P: $A \leftarrow B_1 \wedge \ldots \wedge B_n$. Se i vari B_i appartengono ad un modello allora anche A vi deve appartenere. Supponiamo che $\bigwedge_{j=1}^{n} B_j \in \bigcap_{i \geq 1} \mathcal{D}_i$. Allora, per ogni i avremo che $\bigwedge_{j=1}^{n} B_j \in \mathcal{D}_i$, e poiché $\mathcal{D}_i$ è un modello allora anche $A \in \mathcal{D}_i$; quindi $A \in \bigcap_{i \geq 1} \mathcal{D}_i$. □

COROLLARIO 4.50. Sia P un programma di clausole definite. Siano $\mathcal{D}_1, \mathcal{D}_2, \ldots$ tutti i modelli di Herbrand di P; allora $M_P = \bigcap_{i \geq 1} \mathcal{D}_i$ è un modello di Herbrand. M_P è il minimo modello di Herbrand.

DEFINIZIONE 4.51. Pertanto, $M_P = \{A \in \mathcal{B}_P : P \models A\}$ viene definito come la semantica logica (o modellistica) del programma P.

TEOREMA 4.52 (Correttezza e Completezza della SLD-derivazione). *Sia P un programma di clausole definite. Allora $Oss(P) = M_P$.*

PROOF. Omessa. □

6.4. Risoluzione. La SLD risoluzione, usata per programmi definiti, è un caso particolare della procedura di risoluzione, che lavora su programmi composti genericamente da clausole.

Studiamo il programma P:

$$\begin{array}{lll} p(a) & \leftarrow & p(b) \\ p(b). & & \end{array}$$

Esso si può riscrivere come:

$$\begin{array}{lll} p(a) & \vee & \neg p(b) \\ p(b). & & \end{array}$$

La derivazione del goal $\leftarrow p(a), p(b)$ si può vedere come la risoluzione di:

$$\dfrac{\dfrac{\neg p(a) \vee \neg p(b) \qquad p(a) \vee \neg p(b)}{\neg p(b) \vee \neg p(b) \qquad p(b)}}{\neg p(b) \qquad p(b)}$$
$$\square$$

In generale, la ricerca di una derivazione di A da P si può vedere come la dimostrazione di insoddisfacibilità di $P \cup \{A\}$. Per questa ragione i goals avevano la forma $\leftarrow A_1, \ldots, A_n$. Il significato logico era $\neg(A_1 \wedge \cdots \wedge A_n)$.

In quest'ottica, si può estendere il discorso a programmi fatti di clausole non necessariamente di Horn. Si pensi tuttavia di dimostrare l'insoddisfacibilità del programma

$$\begin{array}{lll} (p(a) \vee p(b)) & \wedge & (\neg p(a) \vee p(b)) \quad \wedge \\ (p(a) \vee \neg p(b)) & \wedge & (\neg p(a) \vee \neg p(b)) \end{array}$$

dell'esempio 4.47.

Partendo da

$$\dfrac{\dfrac{\neg p(a) \vee \neg p(b) \qquad p(a) \vee \neg p(b)}{\neg p(b) \vee \neg p(b) \qquad ???}}{\vdots}$$

Si continuano a costruire risolventi con due letterali. Non si riuscirà a dimostrare nulla.

E' pertanto indispensabile introdurre una ulteriore regola che contrae i letterali uguali (si ha una visione insiemistica). Anche in tal modo tuttavia, non riusciamo ad avere una deduzione dello stesso tipo di quello usato per la SLD:

$$\dfrac{\dfrac{\neg p(a) \vee \neg p(b) \qquad p(a) \vee \neg p(b)}{\neg p(b)} \qquad \dfrac{p(a) \vee p(b) \qquad \neg p(a) \vee p(b)}{p(b)}}{\square}$$

La letteratura sulla risoluzione è molto vasta. Tuttavia solo il frammento di Horn risulta completo per la famiglia di SLD-derivazioni (SLD risoluzione). La forma della SLD-erivazione è sempre a "lista" con in input una clausola del programma. In generale questa restrizione è troppo forte, come si vede dall'esempio sopra.

7. Semantica di punto fisso di programmi definiti

7.1. Prerequisiti di teoria dei reticoli. Una relazione $\leq$ su un insieme A è un *ordine parziale* su A se è riflessiva, antisimmetrica e transitiva. Nozioni di Inf ($\wedge$) Sup $\vee$ massimo e minimo. CPO e reticoli completi. Operatori monotoni, finitari e continui. Punti fissi. Iterazione di un operatore:

DEFINIZIONE 4.53. Sia $\langle A, \leq \rangle$ un insieme parzialmente ordinato. Sia $T : A \longrightarrow A$ un operatore. Definiamo:

$$\begin{cases} T \uparrow 0(I) &= I \\ T \uparrow (n+1) &= T(T \uparrow n(I)) \\ T \uparrow \omega(I) &= \bigvee_{i \geq 0} T \uparrow i(I) \end{cases}$$

Se $I = \bot$ allora $T \uparrow \alpha(I)$ si denota semplicemente con $T \uparrow \alpha$.

TEOREMA 4.54 (Tarski). *Sia T operatore continuo su un CPO. Allora $T \uparrow \omega$ esiste ed è minimo punto fisso di T.*

7.2. Operatore sui programmi.

DEFINIZIONE 4.55. Dato un programma definito P, assumiamo che esista almeno un simbolo di costante in $\mathcal{F}_P$ (altrimenti l'aggiungiamo). Definiamo:

$$ground(P) = \{(A \leftarrow B_1, \ldots, B_n)\theta : \begin{array}{l} c \equiv A \leftarrow B_1, \ldots, B_n \in P, \\ \theta \text{ è sostituzione ground per tutte le variabili di } c \} \end{array}$$

DEFINIZIONE 4.56. Dato un programma definito P, assunto che esista almeno un simbolo di costante in $\mathcal{F}_P$ (altrimenti l'aggiungiamo), definiamo l'operatore di *conseguenza immediata* $T_P : \wp(\mathcal{B}_P) \longrightarrow \wp(\mathcal{B}_P)$ che trasforma interpretazioni in interpretazioni, nel modo seguente:

$$T_P(I) = \{a : \begin{array}{l} a \leftarrow b_1, \ldots, b_n \in ground(P), \text{e} \\ b_1 \in I, \ldots, b_n \in I\} \end{array}$$

ESEMPIO 4.57. Sia P così definito:
$r(b)$.
$p(a)$.
$q(x) \leftarrow r(x), p(x)$.

Allora:

$$\begin{aligned} T_P(\{r(a), r(b), p(b)\}) &= \{r(b), p(a), q(b)\} \\ T_P(\{r(b), p(a), q(b)\}) &= \{r(b), p(a)\} \end{aligned}$$

TEOREMA 4.58. *Sia P un programma definito. Allora I è modello di Herbrand di P se e solo se $T_P(I) \subseteq I$.*

PROOF. ($\rightarrow$) Sia $x \in T_P(I)$; allora esiste un'istanza ground di una clausola $x \leftarrow b_1, \ldots, b_n \in ground(P)$, con $b_1, \ldots, b_n \in I$. Se I è un modello, allora $x \in I$.

($\leftarrow$) Sia $x \leftarrow b_1, \ldots, b_n \in ground(P)$. Assumendo che $b_1, \ldots, b_n \in I$, si ha che $x \in T_P(I)$; essendo $T_P(I) \subseteq I$ per ipotesi, allora $x \in I$ ed I è un modello. □

COROLLARIO 4.59. 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.

PROOF. (1) $T_P(\mathcal{B}_P)$ è sempre sottoinsieme di $\mathcal{B}_P$;
(2) $T_P(I) = I$ implica che $T_P(I) \subseteq I$.

□

L'operatore di conseguenza immediata gode delle seguenti proprietà:

LEMMA 4.60. (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*

$$T_P\left(\bigcup_{i \geq 0} I_i\right) = \bigcup_{i \geq 0} T_P(I_i)$$

PROOF. Esercizio. □

TEOREMA 4.61. *Sia P un programma definito. Allora $M_P = T_P \uparrow \omega$.*

PROOF. ($\subseteq$) $T_P \uparrow \omega$ 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 poicé M_P è il modello minimo. □

COROLLARIO 4.62. Sia P un programma definito. Allora $M_P = T_P \uparrow \omega = Oss(P)$.

7.3. Semantica bottom-up di DATALOG. DATALOG è un linguaggio per l'interrogazione di basi di dati 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.

Ciò garantisce, dato un programma (o DB) P che $T(\mathcal{F}_P)$ è un insieme finito, così come $\mathcal{B}_P$.

Per tali programmi è pensabile fornire una procedura operazionale basata sull'operatore T_P. Dato un goal

$$\leftarrow p(\bar{t})$$

si costruisce $T_P \uparrow \omega$. Tale costruzione è però finita in quanto ad ogni passo o si aggiunge un nuovo atomo (ma in tutto sono $|\mathcal{B}_P|$) oppure si termina perché si ha raggiunto il punto fisso. A tal punto si vede se $p(\bar{t})$ appartiene o meno all'insieme costruito.

Tale procedura sembra molto più lenta di quella top-down implementata dalla SLD-risoluzione. Tuttavia può, ad esempio, essere effettuata off-line e rieseguita solo in seguito ad aggionamenti del DB. In tal modo rspondere ad un goal è equivalente a verificare l'esistenza di un atomo in un insieme di atomi.

Inoltre è sempre terminante, mentre la SLD-derivazione potrebbe entrare in loop anche su programmi per basi di dati, quali ad esempio:

padre(X,Y) $\leftarrow$ figlio(Y,X).
figlio(X,Y) $\leftarrow$ padre(Y,X).
padre(pinco,pallino).

7.4. Operatori all'ingiù e asimmetrie. Dato un operatore T, si può anche definire la sua iterazione *all'ingiù* nel modo seguente:

DEFINIZIONE 4.63. Sia $\langle A, \leq \rangle$ un insieme parzialmente ordinato. Sia $T : A \longrightarrow A$ un operatore. Definiamo:

$$\begin{cases} T \downarrow 0(I) &= I \\ T \downarrow (n+1)(I) &= T(T \uparrow n(I)) \\ T \downarrow \omega(I) &= \bigwedge_{i \geq 0} T \uparrow i(I) \end{cases}$$

Se $\top$ è il massimo del reticolo, allora $T \downarrow \alpha(\top)$ si denota semplicemente con $T \downarrow \alpha$. Si consideri il seguente programma:

$$p(0) \leftarrow q(X)$$
$$q(s(X)) \leftarrow q(X)$$

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 ottengo $T_P(\{p(0)\}) = \emptyset = T_P(\emptyset)$.

Il fatto che l'operatore sia continuo e che si lavori su un reticolo completo (che in particolare ha il massimo) non assicura che con $T_P \downarrow \omega$ si raggiunga un punto fisso. Tuttavia esiste un risultato che assicura l'esistenza di un ordinale α tale che $T_P \downarrow \alpha$ è il massimo punto fisso.

Assodato dunque che $T_P \uparrow \omega$ non coincide necessariamente con $T_P \downarrow \omega(\mathcal{B}_P)$, ci chiediamo se sia possibile che massimo e minimo punto fisso siano distinti.

Si consideri il seguente esempio:

$$q(a) \leftarrow$$
$$q(s(X)) \leftarrow q(s(X))$$

Allora $M_P = T_P \uparrow \omega = \{q(a)\}$ mentre il massimo punto fisso è $\mathcal{B}_P = T_P \downarrow \omega(\mathcal{B}_P)$.

ESERCIZIO 4.64. Si verifichi che tra minimo emassimo punto fisso dell'esempio sopra ci sono una quantità più che numerabile di punti fissi.

CHAPTER 5

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, la lista $[a, b, c]$ si può rappresentare come:

$$f(a, f(b, f(c, \texttt{nil})))$$

Vista la gran necessità di lavorare con le liste nella programmazione ricorsiva, tuttavia, nel linguaggio è prevista una facilitazione. Anzichè il generico simbolo f usato sopra vi è a disposizione il simbolo funzionale binario $[\cdot|\cdot]$ e anzichè `nil` il simbolo di costante []. Dunque, la lista $[a, b, c]$ viene indicata con

$$[a \mid [b \mid [c \mid [\,] \,] \,] \,]$$

Vi è inoltre un secondo livello di astrazione. La lista suddetta si può rappresentare direttamente con $[a, b, c]$. Più in generale, la notazione $[s_1, \ldots, s_n|t]$ denota il termine

$$[s_1|[s_2|\ldots[s_n|t]\ldots]]$$

Quando t è `nil`, si può scrivere semplicemente $[s_1, \ldots, s_n]$ per $[s_1|[s_2|\ldots[s_n|[]]\ldots]]$.

Si osservi bene la differenza tra

- $[X, Y]$ che denota il termine $[X|[Y|[]]]$ ovvero la lista che contiene esattamente i due elementi X e Y, e
- $[X|Y]$ che è la lista che aggiunge in testa alla lista Y (che a priori non sappiamo quanto sia lunga) l'elemento X.

Definiamo ora alcuni predicati sulle liste:

- `member(X, [X | Resto]).`
  `member(X, [A | Resto])` ← `member(X, Resto).`
  Esiste in Prolog una *variabile muta*, cioè un simbolo (`_`) che può essere sostituito ad una variabile il cui uso non è specificato nella clausola. Ad esempio la variabile `Resto` della prima clausola può essere sostituita dalla variabile muta, così come `A` nella seconda.
- `list([ ]).`
  `list([A | B]) :- list(B).`
- `prefix([ ], _ ).`
  `prefix([A | R], [A | S]) :- prefix(R, S).`
- `sublist(X, Y) :- prefix(X, Y).`
  `sublist(X, [B | S]) :- sublist(X, S).`
- `append([ ], X, X).`
  `append([A | R], Y, [A | S]) :- append(R, Y, S).`

Un'altra interessante funzione relativa alle liste è il *reverse*. In Prolog possiamo scrivere due differenti predicati che calcolano il reverse di una lista:

(1)
```
reversenaive([ ], [ ]).
reversenaive([A | R], S) :- reversenaive(R, Rrev),
                            append(Rrev, [A], S).
```
Il reverse basato su questa definizione ha un costo pari a $\frac{n^2}{2}$, con n dimensione della lista.

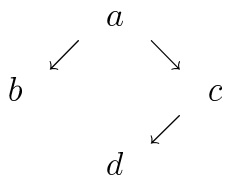
(2)
```
reverse(L, Lrev) :- rev(L, [ ], Lrev).
rev([X | R], Acc, Y) :- rev(R, [X | Acc], Y).
rev([ ], Y, Y).
```
Quest'ultima definizione permette di costruire derivazioni di successo desiderate con lunghezza molto minore. *Acc* viene chiamato accumulatore. L'idea è quella di usarlo com uno stack entro cui mettiamo via via gli elementi letti dalla testa alla coda. Alla fine la lista sarà invertita.

ESEMPIO 5.1. : Supponiamo di voler invertire la lista `[a, b, c]` con usando il predicato `reverse`. Otteniamo le seguenti chiamate interne:
←`reverse([a, b, c], Lrev)` (il goal)
`rev([a, b, c], [ ], Lrev)`
`rev([b, c], [a, [ ] ], Lrev)`
`rev([c], [b, a], Lrev)`
`rev([ ], [c, b, a], Lrev)` e con l'unificazione $\theta = [\ Lrev/[c, b, a]\ ]$
$\leftarrow \square$.

Per esercizio si scriva un predicato binario che riordina una lista di numerali.

2. alberi

Per gli alberi non vi è presente nessun simbolo particolare. Stabiliamo di usare tree (ternario) come costruttore di alberi binari e `nil` per l'albero vuoto. L'albero



Si potrà rappresentare con il termine

$$\mathsf{tree}(\mathsf{a}, \mathsf{tree}(\mathsf{b}, \mathtt{nil}, \mathtt{nil}), \mathsf{tree}(\mathsf{c}, \mathsf{tree}(\mathsf{d}, \mathtt{nil}, \mathtt{nil}), \mathtt{nil})))$$

In tal modo possiamo definire alcuni predicati:

- il predicato istree che dice se un termine è un albero:
  istree(`nil`).
  istree(tree(Label, L , R) :- istree(L), istree(R).
- il predicato membertree che dice se un elemento occorre in un albero:
  membertree(X, tree(X,_,_)).
  membertree(X, tree(_,L,R)) :- membertree(X,L).
  membertree(X, tree(_,L,R)) :- membertree(X,R).

- Proviamo ora ad effetturare una visita in preordine dell'albero stesso. Il risultato lo mettiamo in una lista:

```
preorder( tree(X,L,R), Out) :-
    preorder( L, LList),
    preorder( R, RList),
    append( LList, [X — RList], Out ).
```

Per esercizio si scrivano i predicati relativi alle vistite in post-ordine e in in-ordine.

3. Grafi

Vi sono due principali tecniche di rappresentazione/manipolazione di grafi in Prolog.

Nella prima i vari archi vengono forniti come insieme di fatti (e dunque inseriti dentro il programma). Ad esempio un grafo potrà essere:

```
edge( a , b ).
edge( b , d ).
edge( a , c ).
edge( c , e ).
edge( d , f ).
edge( e , f ).
```

A questo punto si possono scrivere predicati operanti su grafi, quali ad esempio il predicato che stabilisce se due nodi o meno siano connessi:

```
connceted( N , N ).
connceted( N1, N2 ) :-
    edge( N1 , Int ).
    connected( Int , N2 ),
```

Esercizio 5.2. Modificando il predicato ora descritto si definisca un predicato in grado di stabilire la lista dei nodi che occorrono nel cammino trovato per andare da $N1$ a $N2$. Qualora la lista risultasse invertita, si utilizzi la tecnica dell'accumulatore per costruirla invece nell'ordine desiderato.

La seconda tecnica per operare sui grafi è quella di passare il grafo in una lista di archi come argomento. Nell'esempio precedente, la lista:

$$[\mathsf{edge(a,b)}, \mathsf{edge(b,d)}, \mathsf{edge(a,c)}, \mathsf{edge(c,e)}, \mathsf{edge(d,f)}, \mathsf{edge(e,f)}]$$

Fatto ciò, il predicato connected andrebbe definito come:

```
connceted( G , N , N ). connceted( G , N1, N2 ) :-
    member( edge( N1 , Int ), G),
    connected( G, Int , N2 ),
```

Quali sono le principali differenze tra le due versioni:

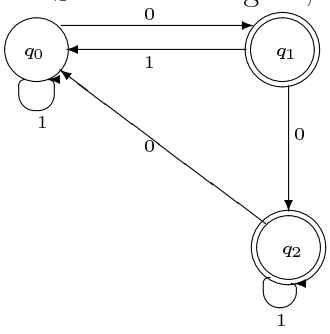
- La seconda vede il grafo effettivamente come dato ed è dunque più pulita.
- La prima però permette di accedere ai vari archi senza dover utilizzare la member che deve scandire una lista per verificare la presenza di un arco. Con questo approccio la verifica di esistenza dell'arco è rimandata al risolutore di Prolog che tipicamente ottimizza queste ricerche, ad esempio utilizzando tabelle di Hash.

Esercizio 5.3. Grafi con archi pesati (etichettati) possono essere rappresentati da fatti del tipo: edge(a , 16 , b). Si scrivano:

(1) una procedura che stabilisce la connessione in questo caso.
(2) una procedura che oltra a far ciò calcola il *costo* del cammino trovato.
(3) una procedura che verifica l'esistenza di un cammino di lunghezza minore ad un certo K passato come input.

4. Automi

Similmante a grafi, 'automa:



può essere rappresentato dai seguenti fatti:

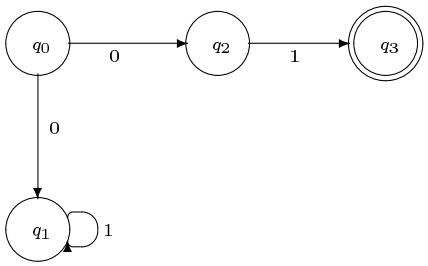
```
%funzione di transizione%
delta(q0,0,q1).
delta(q0,1,q0).
...
%stato iniziale%
initial(q0).
%stati finale%
final(q1).
final(q2).
```

A questo punto si possono descrivere i predicati che chiudono transitivamente e riflessivamente la δ e 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 semplice e dichiarativa. Inoltre, lo stesso programma funziona perfettamente anche nel caso di automi non deterministici!!!

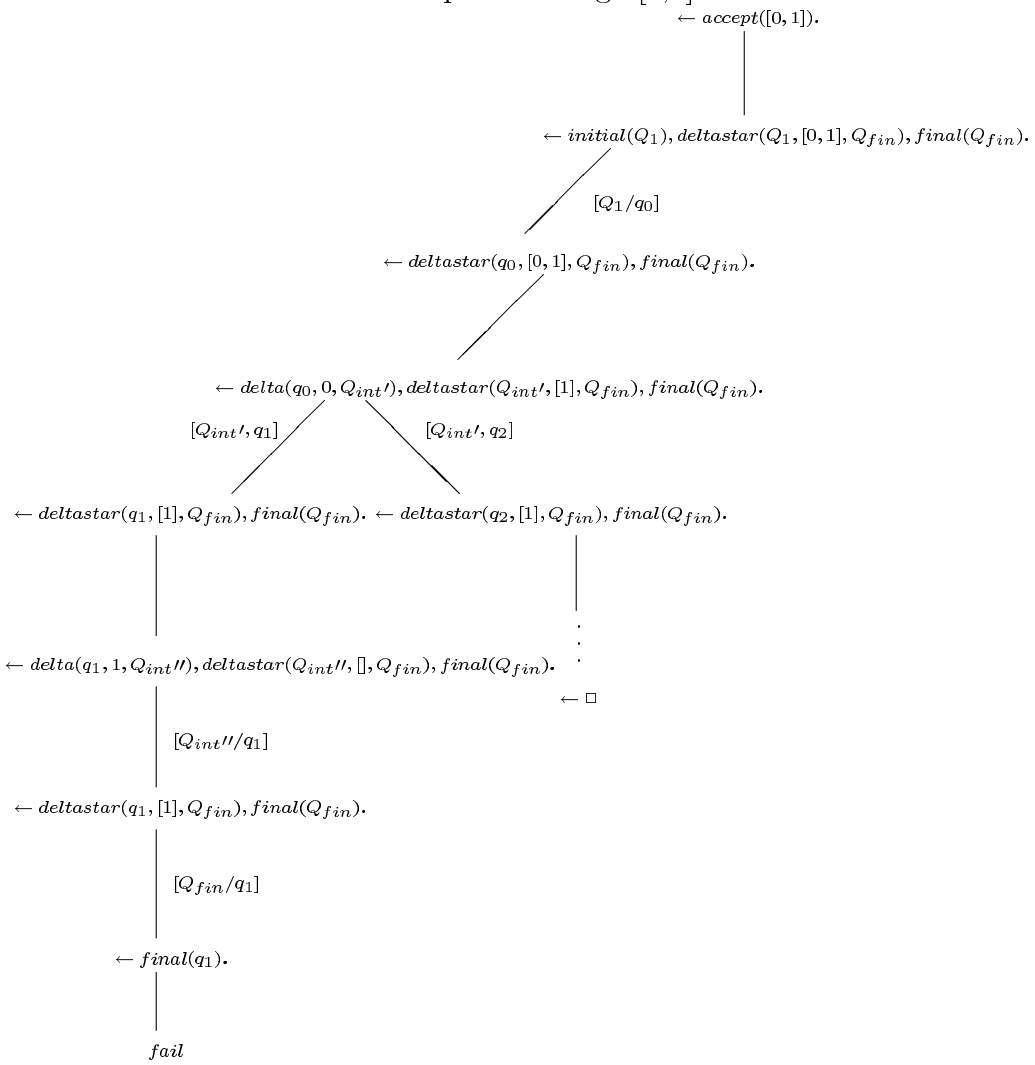
Prendiamo ad esempio l'automa:



rappresentato come:

```
delta(q0,0,q1).
delta(q0,0,q2).
delta(q1,1,q1).
delta(q2,1,q3).
initial(q0).
final(q3).
```

Vediamo l'albero di derivazione per la stringa [0,1]:



5. Esempi

Come ulteriore esempio consideriamo il celeberrimo problema della torre di Hanoi e vediamo come scrivere in Prolog un predicato che lo risolve:

```
:-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).
```

6. Predicati built-in

Descriviamo ora qualche comando di Prolog già presente nel linguaggio. La Turing completezza di Prolog assicura che tali predicati si possono realizzare in Prolog. Tuttavia averli già pronti semplifica di molto le cose.

- "is" valuta l'espressione numerica a destra e ne restituisce il valore a sinistra. Quando anche a sinistra c'è un valore verifica che sia lo stesso valore di ciò che c'è a destra. Le variabili a destra devono sempre essere istanziate. Esempi:
  $\leftarrow 3$ is $1+2$ risponde yes
  $\leftarrow X$ is $1+2$ risponde $X=3$
  $\leftarrow 2+1$ is $1+2$ risponde no
  $\leftarrow 8$ is $X*2$ risponde instantiation error
  $\leftarrow X$ is X risponde instantiation error
- "$<$" è ovviamente il predicato minore ed ha un comportamento simile al precedente, ad esempio:
  $\leftarrow 2<3$ risponde yes
  $\leftarrow 3<2$ risponde no
  $\leftarrow X<3$ risponde instantiation error
- "$s=t$" questo predicato controlla se le gli atomi s e t possono essere unificati:
  $\leftarrow 2=2$ risponde yes
  $\leftarrow a=b$ risponde no
  $\leftarrow X=2$ risponde dando l'mgu cioè $[X/2]$

- La negazione di questo predicato è data da "$s\backslash=t$": s non unifica con t.
- "s==t" corrisponde all'uguaglianza sintattica:
  $\leftarrow 2==2$ risponde yes
  $\leftarrow a==b$ risponde no
  $\leftarrow X==2$ risponde no
- La negazione di questo predicato è data da "$s\backslash==t$".
- "=:=" è l'uguaglianza di espressioni nomeriche ground, cioè:
  $\leftarrow 2+1=:=4-1$ risponde yes
  $\leftarrow X=:=2$ risponde instantiation error
- La diseguaglianza è data da "$=\backslash=$".

7. Predicati di tipo

- `integer(N):` Controlla se un termine è un intero.
  - integer(0) $\Rightarrow$ *True*
  - integer(1) $\Rightarrow$ *True*
  - integer(1.2) $\Rightarrow$ *False*
  - integer(1+2) $\Rightarrow$ *False*
    Anche se la somma restituisce un intero, il risultato è *False* in quanto 1+2 è un'espressione ma non un numero intero
- `real(N) :` Controlla se un termine è un reale.[1]
  - real($(1.2)^3$) $\Rightarrow$ *True*

[1] In SICStus il comando real non esiste.

  - real(c) ⇒ *False*
- `atom(c)` : Controlla se il termine `c` è un termine fatto con una costante (ma non numerica!)
  - atom(c) ⇒ *True*
  - atom(p(c)) ⇒ *False*
  - atom(1) ⇒ *False*
- `number(N)` : Restuisce il tipo del numero N inserito(real, integer, ...).
  - number(1) ⇒ *integer*
  - number(1.2) ⇒ *real*
- `compound(f(X,Y))` : Controlla se il termine è composto.
  - compound(f(X,Y)) ⇒ *True*
  - compound(c) ⇒ *False*
- `functor(Term,F,Arity)` : Controlla se un termine è una funzione.
  - functor(f(a,b,c),f,3) ⇒ *True*
  - functor(f(a,b,c),F,N) ⇒ *True,* $F = f, N = 3$
  - functor(g(a),f,2) ⇒ *False*
- `arg(N,Term,Arg)` : Restituisce l'argomento di una funzione.
  - arg(2,f(a,b,c),b) ⇒ *True*
  - arg(3,f(a,b,c),X) ⇒ *True,* $X = c$
  - arg(N,f(a,b,c),B) ⇒ *True,* $N = 2$
- `=..` : Operatore *univ*.
  - f(a,b,g(X,Y)) =.. [f,a,b,g(X,Y)] ⇒ *True*
  - f(a,b,g(X,Y)) =.. [F,X,b,g(X,Y)] ⇒ *True,* $F = f, X = a$

8. Predicati metalogici o extralogici:

- `var(Term)`: Dice se un termine è una variabile
- `nonvar(Term)`: Dice se un termine non è una variabile
  - var(2) ⇒ *False*
  - nonvar(2) ⇒ *True*
  - var(X) ⇒ *True*
  - var(f(X)) ⇒ *False*
  - nonvar(X) ⇒ *False*

Quindi con i due nuovi comandi visti sopra aggiustiamo il Plus:

```
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 5.4. Definire in Prolog il predicato che:

(1) calcola la *size* di un termine (numero occorrenze di simboli funzionali e di costante in esso).
(2) l'altezza dell'albero associato ad un termine.
(3) scrivere, senza usare il built-in omonimo, un predicato che dica quando un termine è ground.
(4) scrivere, senza usare il built-in omonimo, un predicato che dica quando un termine è non-ground.
(5) effettua l'unificazione (con occur check) di due termini.

(6) stabilisce (e trova le evenuali soluzioni) se una formula logica proposizionale sia o meno soddisfacibile.

In SICSstus esiste un comando di nome ground con la funzione di restituirmi True se il termine è ground , False altrimenti.
ground(f(a,b)) $\Rightarrow$ *True*
ground(X) $\Rightarrow$ *False*

9. Il CUT

La funzione del CUT (indicato come **!** all'interno del codice) è quella di bloccare il backtracking.

ESEMPIO 5.5. Al Programma

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

viene associato l'albero SLD di figura 1. Fornisce pertanto le soluzioni $X = b$ e $X = c$.

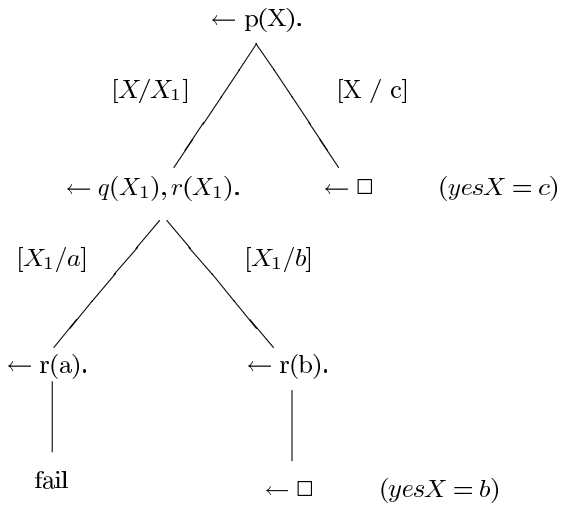


FIGURE 1. SLD-tree senza cuts

Consideriamo ora il programma

```
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 quella $X = b$. Una spiegazione diagrammatica del funzionamento la si può avere inserendo nell'albero SLD degli archi monodirezionali (indicati con la notazione del *diodo*) che si possono attraversare solo in un verso (Fig. 2).

Dal programma seguente non si ottiene invece nessuna risposta.

```
p(X):- q(X), !,  r(X).
p(c).
q(a).
q(b).
r(b).
```

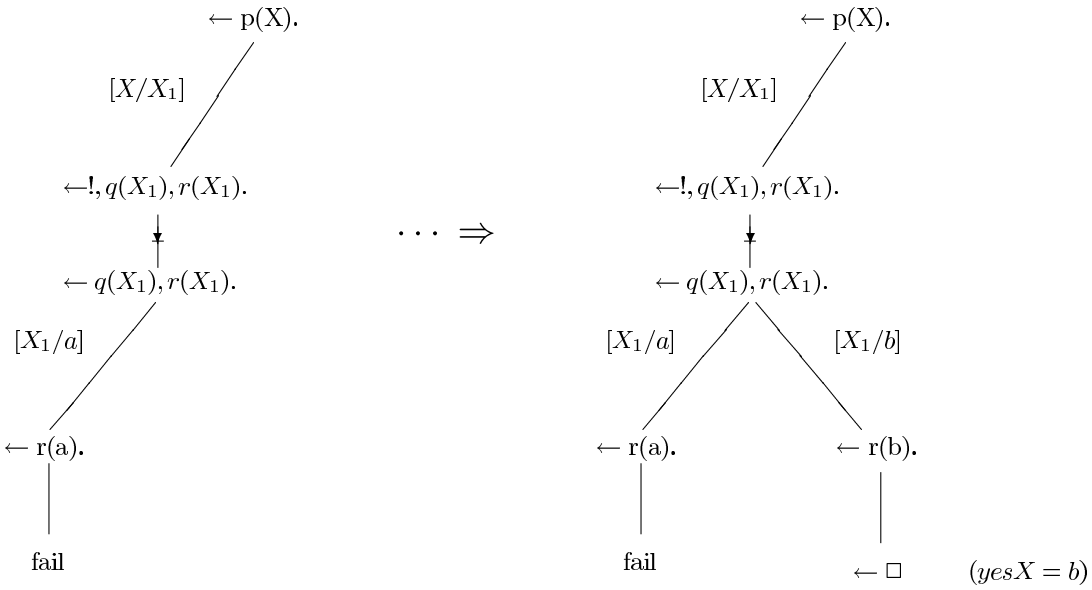


FIGURE 2. Albero con CUT

9.1. CUT VERDE. Aggiunge determinismo al programma. L'inserimento di un CUT all'inizio di ogni possibile scelta permette di evitare di visitare rami inutili dell'albero una volta trovata una soluzione.

$$\begin{array}{rll} \texttt{merge}([X|X_s],[Y|Y_s],[X|Z_s]) & \leftarrow & X < Y, !, \texttt{merge}(X_s,[Y|Y_s],Z_s]. \\ \texttt{merge}([X|X_s],[Y|Y_s],[X,Y|Z_s] & \leftarrow & X == Y, !, \texttt{merge}(X_s,Y_s,Z_s]. \\ \texttt{merge}([X|X_s],[Y|Y_s],[Y|Z_s]) & \leftarrow Y < X, !, \texttt{merge}([X,X_s],Y_s,Z_s]. & \end{array}$$

9.2. CUT ROSSO. Permette di omettere condizioni esplicite. Programmi che sembrano uguali possono avere semantiche completamente diverse!

$$\begin{array}{rcl} \texttt{norep_member}(X,[X|X_s]). & & \\ \texttt{norep_member}(X,[Y|Y_s]) & \leftarrow & X \backslash == Y, \texttt{norep_member}(X,Y_s). \end{array}$$

In questo caso il member cerca solo la prima soluzione e poi si ferma.

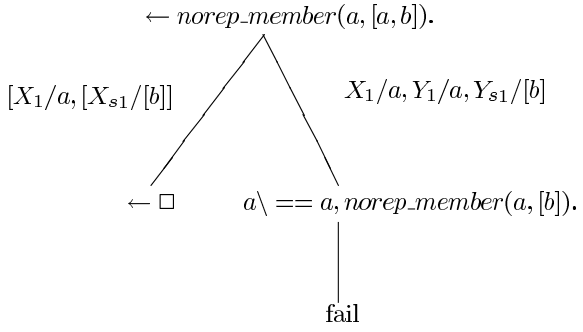


FIGURE 3. Codice con *norep_member*

Una volta trovata la prima soluzione si può evitare di trovare le altre soluzioni.
Si può così riscrivere il programma :
`norep_member`$(X, [X|X_s]) \leftarrow !$
`norep_member`$(X, [Y|Y_s]) \leftarrow$ `norep_member`(X, Y_s).
Non potrò mai entrare nella seconda clausola con $X == Y$; viene così persa l'informazione di $X == Y$.

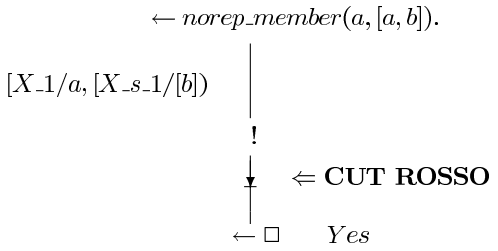


FIGURE 4. Codice con CUT ROSSO

E' molto efficiente ma non si capisce la differenza tra questo member ed il member standard.

10. FAIL

E' un predicato nullario che fallisce sempre, è come un atomo il cui valore è sempre *False*.

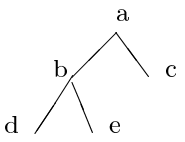


FIGURE 5. Albero genealogico

Vorrei scrivere un predicato che mi dia tutte le coppie padre figlio di figura 5.
`padri ← padre(X,Y), write(X,'padre di',Y),fail.`

Albero di derivazione:

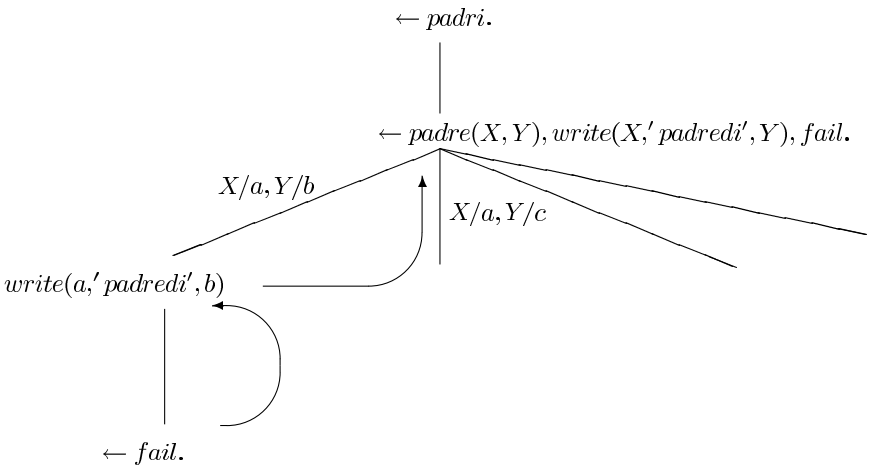


FIGURE 6. Codice con Fail

Alla fine l'output sarà *False* ma avrò comunque stampato tutti i padri; se avessi voluto come risposta True avrei dovuto aggiungere la clausola padri all'insieme delle clausole iniziali.

11. Operatori

Come già accennato, all'inizio del programma si possono definire degli operatori in questo modo:

$$:-op(numero, tipologia, nome_operazione).$$

dove *numero* è la priorità dell'operatore che si stà definendo (un numero alto corrisponde ad una bassa priorità), con *tipologia* invece si definiscono alcune propietà dell'operatore, ad esempio prefisso, infisso oppure sufisso. In breve:

- xfx infisso, non si può annidare (comporre l'operatore con altri)
- yfx infisso, si può annidare a sinistra
- xfy infisso, si può annidare a destra
- fx sufisso, non si può annidare a destra
- fy sufisso, si può annidare
- xf prefisso, non si può annidare
- yf prefisso, si può annidare

CHAPTER 6

Programmi Generali

Nel definire i predicati `studente_single` e disgiunte risulta utile (anche se non indispensabile) disporre di una forma di negazione dei letterali in un goal.

```
studente(bill).
sposato(joe).
studente_single(X) :- studente(X), not sposato(X).

interseca(X, Y) :- member(Z, X),member(Z, Y).
disgiunte(X, Y) :- not interseca(X, Y).
```

DEFINIZIONE 6.1. 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.

1. Problemi

I programmi generali non godono di quasi nessuna delle eleganti proprietà dei programmi definiti studiati nel Capitolo 4. In particolare,

(1) Le clausole non sono più di Horn. Si pensi infatti a

$$p(a) \leftarrow \neg q(a)$$

che è equivalente a $p(a) \vee q(a)$.

(2) L'intersezione di modelli non è più un modello (nel caso sopra si pensi ai due modelli $\{p(a)\}$ e $\{q(a)\}$.

(3) Può non esistere un modello minimo (si veda sopra).

(4) Si può definire anche qui un operatore T_P. Il modo naturale è:

$$\begin{array}{rcll} T_P(I) & = & \{a: & a \leftarrow b_1, \ldots, b_m, \neg c_1, \ldots, \neg c_n \in \mathit{ground}(P) \\ & & & b_1 \in I, \ldots, b_m \in I, \\ & & & c_1 \notin I, \ldots, c_n \notin I\} \end{array}$$

Si osservi che $T_P(\emptyset) = \{p(a)\}$ e $T_P(\{q(a)\}) = \emptyset$. Pertanto T_P non è nemmeno monotono.

(5) Si pensi al programma:

$$\begin{array}{rcl} p(a) & \leftarrow & p(b) \\ p(a) & \leftarrow & \neg p(b) \\ p(b) & \leftarrow & p(a) \end{array} \quad \text{ovvero:} \quad \begin{array}{l} p(a) \vee \neg p(b) \\ p(a) \vee p(b) \\ \neg p(a) \vee p(b) \end{array}$$

assieme al goal

$$\leftarrow p(a), p(b)$$

(ovvero $\neg p(a) \vee \neg p(b)$). Siamo nelle stesse situazioni dell'esempio della sezione 6.4 dove si era visto come fosse necessario disporre della procedura di *risoluzione* (non SLD) per giungere a refutare il goal.

Pertanto non si cerca di affrontare in modo totale i programmi generali. Si cerca piuttosto di definire delle semantiche operazionali che approssimino il più possibile il comportamento desiderato.

2. Proposte operazionali

Vi sono tre proposte operazionali per manipolare i programmi e goals generali senza dover usare tutta la risoluzione: la *Closed World Assumption*, la *Negation as (finite) failure* e la *regola di Herbrand* (basata sulla nozione di completamento di un programma). Per semplicità, assumiamo che il programma P sia un programma definito e lo interroghiamo con un goal generale.

2.1. Chosed World Assumption. Ci chiediamo quando si deve fornire risposta positiva ad un goal:

$$\leftarrow \neg A$$

La regola Closed World Assumption (in breve, CWA) risponde *yes* quando non esiste una SLD-derivazione per $\leftarrow A$ dal programma P. Ovvero, per i risultati di equivalenza studiati, quando

$$A \notin T_P \uparrow \omega$$

Questo primo metodo, introdotto da Reiter nelle basi di dati deduttive, non è applicabile ai programmi Prolog, in quanto la proprietà su cui è basato è non ricorsivamente enumerabile.

2.2. Negation as Failure. La regola di Negazione per fallimento finito (in breve, NaF), introdotta da Clark, fornisce *yes* quando l'albero SLD per $\leftarrow A$ fallisce finitamente. È una sorta di approssimazione, con una regola ora ricorsivamente enumerabile, del primo approccio. Vediamo degli esempi:

Esempio 6.2. Consideriamo il programma:

$$\begin{array}{lcl} p(a). & & \\ p(b). & & \\ q(a). & & \\ r(X) & \leftarrow & p(X), q(X). \end{array}$$

e analizziamo il goal $\leftarrow \neg r(b)$.

- CWA risponde *yes*, perché *r(b)* non appartiene al modello minimo

$$M_P = \{p(a), p(b), q(a), r(a)\}$$

- NaF agisce come segue: viene iniziata una computazione *ausiliaria* per il goal $\leftarrow r(b)$ (si veda Fig. 1). La derivazione ausiliaria fornisce come risposta *no*: il risultato al goal generale originario è *yes*.

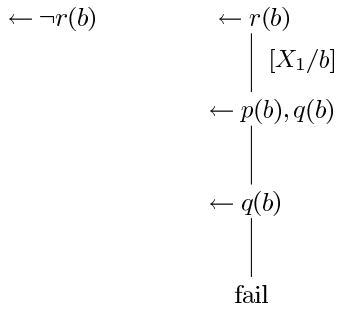


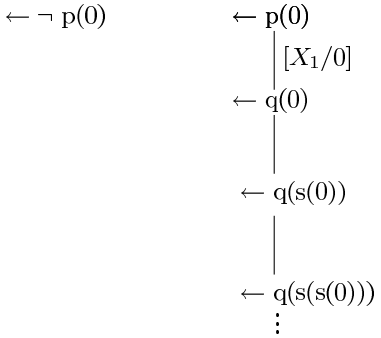
FIGURE 1. Derivazione usando NaF

ESEMPIO 6.3. Consideriamo ora il programma:

$$\begin{array}{lcl} q(X) & \leftarrow & q(s(X)). \\ p(X) & \leftarrow & q(0). \end{array}$$

ed analizziamo il goal $\leftarrow \neg p(0)$.

- CWA risponde *yes*, visto che $M_P = \emptyset$.
- NaF verifica il goal ausiliario $\leftarrow p(0)$. La computazione ausiliaria entra però in loop.



Pertanto NaF non risponde nulla in questo caso.

La formalizzazione della metodologia risolutiva impiegata per implementare la NaF è chiamata *SLDNF*, e funziona nel seguente modo:

DEFINIZIONE 6.4. Sia $G = \leftarrow L_1, \ldots, L_n$ un goal generale. Sia L_i il letterale selezionato.

(1) Se il letterale selezionato L_i è positivo, allora si esegue un passo standard di *SLD*-risoluzione (Def. 4.4);
(2) se il letterale selezionato L_i è della forma $\neg A$ ovvero è negativo e l'atomo A è *ground*, allora si inizi una *SLDNF* derivazione ausiliaria per A:
   - Se tale derivazione termina e fornisce yes, allora la derivazione del goal G è di fallimento.
   - Se la derivazione termina e fornisce no, allora la risolvente di G è: $G = \leftarrow L_1, \ldots, L_{i-1}, L_{i+1}, \ldots, L_n$.

Vediamo come derivare usando SLDNF faccia perdere la semantica puramente logica delle clausole.

ESEMPIO 6.5. Dato il programma P

$$p(a) \leftarrow \neg q(a)$$

equivalente logicamente a $p(a) \vee q(a)$, consideriamo i seguenti goals:

(1) $\leftarrow p(a)$. Si usa l'unica clausola e ci si riconduce al goal $\leftarrow \neg q(a)$. Viene lanciata la compuatzione ausiliaria $\leftarrow q(a)$ la quale è di fallimento finito. Dunque la risposta finale è *yes*.
(2) $\leftarrow \neg p(a)$. Viene lanciata la derivazione per il goal $\leftarrow p(a)$ la quale, come visto, fornisce *yes*: il risultato finale è *no*;
(3) $\leftarrow q(a)$. La risposta è *no*.
(4) $\leftarrow \neg q(a)$: si lancia la derivazione del goal $\leftarrow q(a)$. Abbiamo appena visto che a questa si risponde *no*, pertanto la risposta finale è *yes*.

L'insieme $Oss(P)$ risulta pertanto $\{p(a)\}$ oppure, usando una nozione più estesa, $\{p(a), \neg q(a)\}$. In ogni caso si noti una completa asimmetria rispetto al significato logico della clausola. $Oss(P)$ è in qualche modo associato al modello minimale (non minimo!) $\{p(a)\}$.

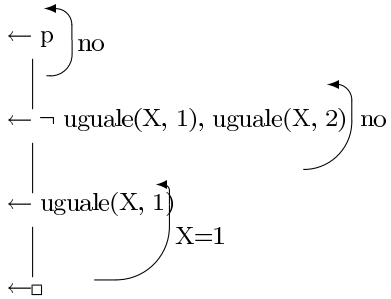
Cerchiamo ora di intuire perchè nella definizione di *SLDNF* è stato imposto che l'atomo A sia *ground*.

Esempio 6.6. Si considerino le clausole

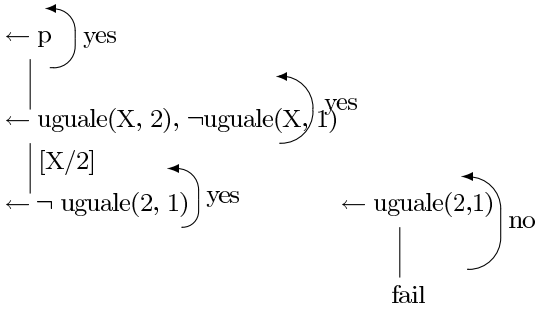
(1) $\mathsf{uguale}(X, X)$.
(2) $p \leftarrow \neg\mathsf{uguale}(X, 1), \mathsf{uguale}(X, 2)$.
(3) $p \leftarrow \mathsf{uguale}(X, 2), \neg\mathsf{uguale}(X, 1)$.
(4) $q \leftarrow \neg p$

E consideriamo i programmi $P_1 = \{(1), (2), (4)\}$ e $P_2 = \{(1), (3), (4)\}$ logicamente equivalenti. Analizzando il programma uno si aspetta che p sia vero e q sia falso.

$P_1 = \{(1), (2), (4)\}$**:** Osserviamo la derivazione di $\leftarrow p$.



$P_1 = \{(1), (3), (4)\}$**:** Osserviamo ora la derivazione di $\leftarrow p$.



Si osservi cha aver applicato la procedura di SLDNF su un goal non ground ci ha fatto perdere una soluzione: in tale caso allora si ha che la SLDNF è una procedura *incompleta*.

Se consideriamo ora il goal $\leftarrow q$ abbiamo che, sempre nel caso in cui il passo SLDNF sia applicato anche a letterali non ground, viene fornita una risposta non attesa. Pertanto si ha che la SLDNF è una procedura *incorretta*.

Vi sono in letteratura molti studi sulle condizioni che garantiscono che la SLDNF sia corretta e completa. Ne forniamo qui una sufficiente ma non necessaria: *il letterale negato deve essere ground e nell'albero ausiliario non si devono incontrare altre negazioni.*

Come regola generale, pertanto, si deve cercare di scrivere programmi tali che quando si arriva al letterale negato, tutte le variabili presenti in esso siano già ground.

2.3. Regola di Herbrand. Tale regola computazionale dice di dedurre il goal $\leftarrow \neg A$ da P quando A è falso in tutti i modelli di Herbrand del completamento del programma P, $Comp(P)$. Definiamo ora $Comp(P)$ su un esempio:

Sia P il programma:

$$\begin{array}{rcl} r(a,c). & & \\ p(a). & & \\ p(b). & & \\ q(X) & \leftarrow & p(X), r(X). \\ q(X) & \leftarrow & s(X). \end{array}$$

Passiamo innanzitutto ad una forma normalizzata di P, detta $norm(P)$: gli argomenti dell'atomo della testa devono tutti essere variabili (meglio usarle in modo uniforme $X_1, X_2, \dots$). Per far ciò introduco delle uguaglianze.

$$\begin{array}{rcl} r(X_1,X_2) & \leftarrow & X_1 = a, X_2 = c. \\ p(X_1) & \leftarrow & X_1 = a. \\ p(X_1) & \leftarrow & X_1 = b. \\ q(X_1) & \leftarrow & p(X_1), r(X_1,Y_1). \\ q(X_1) & \leftarrow & s(X_1). \end{array}$$

L'aver messo variabili in modo uniforme ci permette di raccogliere le teste uguali: Da $norm(P)$ si costruisce la teoria $\mathsf{iff}(P)$ nel modo seguente:

$$\begin{array}{rcl} r(X_1,X_2) & \leftrightarrow & (X_1 = a \wedge X_2 = c) \\ p(X_1) & \leftrightarrow & (X_1 = a) \vee (X_1 = b) \\ q(X_1) & \leftrightarrow & \exists Y_1(p(X_1), r(X_1,Y_1)) \vee \\ & & s(X_1) \\ s(X_1) & \leftrightarrow & \texttt{false} \end{array}$$

Il predicato s è usato nella definizione di q ma mai dfinito. Pertanto lo si forza ad essere falso.

Il completamento di P, $Comp(P) = \mathsf{iff}(P) \wedge (F1)(F2)(F3)$, dove $(F1)(F2)(F3)$ sono gli assiomi della *Clark's Equality Theory* (si veda Def. 3.22).

3. Confronti tra le tre regole

La regola CWA è non ricorsivamente enumerabile e pertanto non applicabile. Anche la regola di Herbrand basata sui modelli del completamento non appare facilmente automatizzabile. Tuttavia il seguente teorema giustifica l'introduzione delle tre regole e le colloca precisamente.

TEOREMA 6.7. *Sia P un programma definito e A un atomo di $\mathcal{B}_P$. Allora:*

(1) $\leftarrow \neg A$ *è inferito per CWA se e solo se* $A \in \mathcal{B}_P \setminus T_P \uparrow \omega$
(2) $\leftarrow \neg A$ *è inferito dalla regola di Herbrand se e solo se* $A \in \mathcal{B}_P \setminus gfp(T_P)$.
(3) $\leftarrow \neg A$ *è inferito per 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 $\alpha \geq \omega$. Sappiamo inoltre che $lfp(T_P) = T_P \uparrow \omega \subseteq gfp(T_P)$. Pertanto i tre insiemi caratterizzati nel teorema sono inclusi l'uno nell'altro.

ESERCIZIO 6.8. Si verifichi il teorema 6.7 sul seguente programma definito, in cui i tre insiemi di cui si parla sono inclusi strettamente.

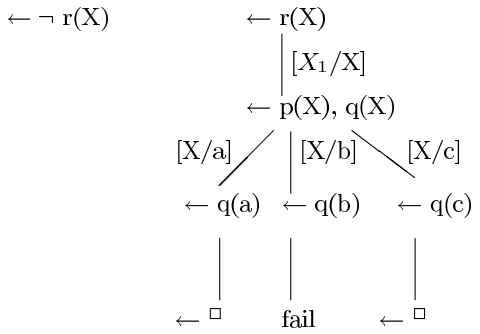
$$\begin{array}{rcl} p(f(X)) & \leftarrow & p(X) \\ q(a) & \leftarrow & p(X) \\ q(f(X)) & \leftarrow & q(f(X)) \\ r(a) & & \end{array}$$

4. Negazione Costruttiva

Abbiamo visto che goals negati non ground non possono essere calcolati dalla NaF. Vediamo se è possibile rimuovere questo vincolo. Sia dato il programma

$$\begin{array}{rcl} p(a). & & \\ p(b). & & \\ p(c). & & \\ q(a). & & \\ q(c). & & \\ r(X) & \leftarrow & p(X), q(X). \end{array}$$

e costruiamo l'albero SLD per il goal $\leftarrow r(X)$. Si ha che:



$$r(X) \leftrightarrow (X = a) \vee (X = c)$$

Pertanto potremmo immaginare una regola che complementa tale risposta e fornisce a $\leftarrow \neg r(X)$ il risultato:

$$yes, (X \neq a) \wedge (X \neq c)$$

Sembra un metodo applicabile, solo che nella risposta necessitiamo non solo di sostituzioni ma anche di disugualianze. Dovremmo pertanto cambiare le definizioni di risposta calcolata. Consideriamole clausole:

$$p(X, Y) \leftarrow X = f(Z), Y = g(Z)$$

Si avrebbe che:

$$\forall X, Y(p(X,Y) \leftrightarrow \exists Z\,(X = f(Z) \wedge Y = g(Z))$$

La complementazione di tale formula sarebbe:

$$\neg p(X,Y) \leftrightarrow \forall Z\,(X \neq f(Z) \vee Y \neq g(Z))$$

che, in questo caso, può essere riscritto in

$$\begin{array}{l}\forall X(X \neq f(Z)) \vee \\ \exists Z\,(X = f(Z) \wedge \forall Z\,(Y \neq g(Z)))\end{array}$$

Le risposte ad un goal negato di questo tipo dovrebbero pertanto contenere anche formule universalmente quantificate.

La negazione costruttiva può essere implementata. Tuttavia vi sono anche qui delle assunzioni (ad esempio la finitezza degli alberi) e le risposte che si ottengono sono spesso troppo implicite da essere di aiuto all'utente.

5. Implementazione della NaF

In *Prolog*, la negazione viene implementata nel seguente modo:

```
not(P) :- P, !, fail.
not(P).
```

`Osservazione:` La definizione non è delle migliori, visto che `P` passa da termine ad atomo ed inoltre si utilizza un `cut rosso`.

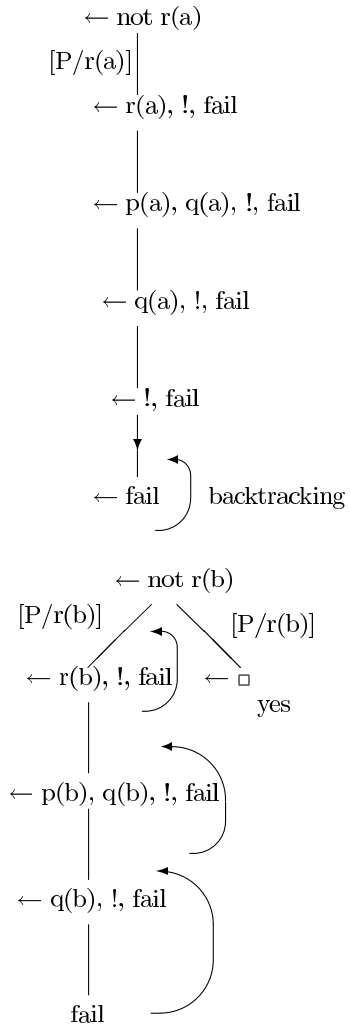
Nonostante questo, vediamo con due esempi definiti sul seguente programma, come possa funzionare:

```
p(a).
p(b).
q(a).
r(X) ← p(X), q(X).
```

- esempio 1:

  L'idea di base è che appena trova una soluzione risponde *no*.
- esempio 2:

Esercizio 6.9. Verificare gli esempi di *flounders* (quelli con i goals non ground) visti prima, con la soluzione sopra.


← not r(a)
[P/r(a)]
← r(a), !, fail
← p(a), q(a), !, fail
← q(a), !, fail
← !, fail
← fail
backtracking
← not r(b)
[P/r(b)]
[P/r(b)]
← r(b), !, fail
← □
yes
← p(b), q(b), !, fail
← q(b), !, fail
fail

CHAPTER 7

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. Il linguaggio Prolog permette di programmare in modo dichiarativo e meno. In questo capitolo si forniranno alcuni principi di base e si mostreranno alcuni di programmi il più possibile dichiarativi.

1. Principi di base

Programmare in modo dichiarativo permette di aumentare

- chiarezza e
- leggibilità

dei programmi. Nel caso di programmazione in Prolog, la particolare concisione possibile permette di poter scrivere un programma in unico file. Questo fa sì che alcuni problemi legati alla non composizionalità della semantica denotazionale di Prolog non emergano.[1]

E' fondamentale scegliere dei buoni *nomi* per i

- Simboli predicativi,
- Simboli di funzione,
- Variabili

Per quanto riguarda il primo punto, è buona norma cercare di assegnare dei nomi che ricordino il significato di tale predicato, non lo scopo per il problema che si vuole risolvere.

Malgrado un buon programma Prolog sia di facile lettura, commenti sono sempre benvenuti. In particolare, sarebbe auspicabile l'inserimento, come commenti, delle seguenti specifiche:

- procedure p(T1,...,Tn)
- type T1: (ad esempio: lista di variabili o interi)
- ...
- type Tn: ...
- Significato della procedura in corso di definizione
- Modi: Ad esmpio si vuole dire che T1 e T2 sono input, Tn è output. Per far ciò vi è una sintassi tipica.
  - + sta per termine ground
  - - sta per non ground
  - ? sta per entrambi.

[1] Si pensi ai due programmi $P_1 = \{p \leftarrow q\}, P_2 = \{q\}$. Si ha che $M_{P_1} = \emptyset, M_{P_2} = \{q\}$, mentre $M_{P_1 \cup P_2} = \{p, q\}$. L'introduzione di p nella semantica del programma complessivo è del tutto inattesa guardando le semantiche separate.

Un modo del tipo:

$$(+,+,?),(+,?,+)$$

significa che il predicato può essere usato sostanzialmente con T1 e T2 come input oppure con T1 e T3 come input.

- Molteplicità: per ogni modo è bene dire se ci si aspetta un comportamento deterministico (al massimo una soluzione) o non-deterministico.

Il debugging di un programma può essere fatto per mezzo di tecniche tradizionali (tracer dell'interprete, comandi di write messi in vari punti) o automatiche (abstract debugging [**?**]). Nella Sezione **??** sarà mostrato come costruirsi un tracer configurabile.

E' buona norma usare una giusta commistione tra dichiaratività ed efficienza. Cercare l'efficienza snaturando la leggibilità del programma non fa parte dei principi della programmazione dichiarativa (se si è portati a programmare così è meglio usare altri paradigmi). Vediamo un esempio in tale direzione. Consideriamo i due programmi equivalenti (ed equidichiarativi):

```
sumlist([], 0).                    sumlist([X|Xs], Sum) ←
sumlist([X|Xs], Sum) ←                 sumlist(Xs, S),
    sumlist(Xs, S),                    Sum is S + X.
    Sum is S + X.                  sumlist([], 0).
```

Mediamente ci si aspetta di chiamare dei goals con lista non vuota $\leftarrow \mathsf{sumlist}([\mathsf{t}_1, \ldots, \mathsf{t}_n], \mathsf{L})$. Per tali goals il secondo programma risulta più efficiente in quanto nello sviluppare la derivazione non viene tentata l'unificazione dall'atomo selezionato alla prima clausola. Vi sono pertanto n unificazioni in meno.

2. Generate and test

La metodologia *generate and test* è una metodologia ampiamente utilizzata in programmazione dichiarativa per cercare soluzioni ad un dato problema. La struttura di base è la seguente:

```
find(X̄) ←
    genera(X̄),
    testa(X̄).
```

Il predicato genera è tipicamente non-deterministico. Fornirà una ad una varie possibili soluzioni da uno spazio che, tipicamente, ma non necessariamente, ha dimensioni esponenziali. testa è invece un predicato deterministico, solitamente efficiente, che verifica se il valore in oggetto soddisfa i requisiti.

Il seguente esempio mostra una possibile implementazione della verifica di intersezione non vuota fra insiemi con metodologia generate and test.

```
interseca(X, Y) ←
    member(Z, X),   %%generazione di una Z
    member(Z, Y).   %%verifica se Z appartiene a Y.
```

Il seguente esempio mostra come risolvere (inefficientemente!) 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 definiti nel seguito:

```
ordinata([X, Y|R]) ←
    X < Y,
    ordinata([Y|R].
ordinata([]).
ordinata([X]).
%%%
permutazione(Xs, [Z|Zs]) ←
    seleziona(Z, Xs, Ys), %%%Ys = Xs \ [Z]
    permutazione(Ys, Zs).
permutazione([], []).
%%%
seleziona(X, [Y|R], [Y|S]) ←
    seleziona(X, R, S).
seleziona(X, [X|R], R).
```

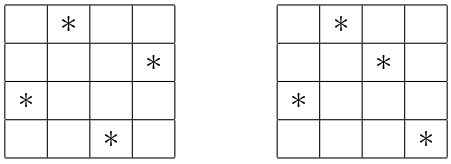
3. N-regine

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 pezzo nella stessa riga, colonna o diagonale).

Vediamo la formulazione generate and test del problema:

```
regine(N, Allocazione) ←
    genera(N, Allocazione),
    safe(Allocazione).
```

Una allocazione si può rappresentare mediante una lista di interi. Ad esempio le due configurazioni seguenti



saranno rappresentate, rispettivamente, dalle liste $[3, 1, 4, 2]$ e $[3, 1, 2, 4]$. La prima è una configurazione sicura, la seconda no.

Definiamo il predicato *genera*:

```
genera(N, Allocazione) ←
    lista_vars(N, Allocazione),
    valori_ammessi(N, ListaValori),
    funzione(Allocazione, ListaValori).
```

Dove lista_vars genera una lista di N variabili. valori_ammessi una lista che contiene gli elementi dall'1 all'N (compresi). funzione invece prova un qualunque assegnamento dalle variabili all'insieme dei valori ammessi. Sono definite:

```
lista_vars(N, [X|R]) ←
    N > 0,
    M is N − 1,
    lista_vars(M, R).
lista_vars(0, []).
%%%
valori_ammessi(N, [M|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).
```

Rimane da implementare il predicato safe che realizza la fase di testing, ed i suoi predicati ausiliari:

```
safe([]).
safe([Q|Qs]) ←
    safe(Qs),
    not attacca(Q, Qs).
%%%
attacca(X, Xs) ←
    att(X, 1, Xs).
%%%
att(X, N, [Y|R]) ←
    X is Y + N.
att(X, N, [Y|R]) ←          (*)
    X = Y.
att(X, N, [Y|R]) ←
    X is Y − N.
att(X, N, [Y|R]) ←
    N1 is N + 1,
    att(X, N1, R).
```

Si osservi che l'uso della negazione è giustificato dal fatto che in quel punto i termini coinvolti saranno ground.

Analizziamo il problema così formulato dal punto di vista computazionale. La fase genera(N, Allocazione) genera N^N configurazioni possibili: le verifichiamo tutte fin quando non troviamo una soluzione.

La fase safe(Allocazione) è invece deterministica ed ha un ordine quadratico di complessità.

Per velocizzare la soluzione si deve cercare di ridurre lo spazio di ricerca.

Una prima ottimizzazione possibile è quella di non generare tutte le N^N soluzioni ma solo le permutazioni delle liste $[1, \ldots, N]$. In tal modo si passa ad una complessità di $N!$. Il fatto $(*)$ usato nella fase di verifica può ora essere omesso.

```
genera(N, Allocazione) ←
    lista_vars(N, ListaValori),
    permutazione(ListaValori, Allocazione).
```

Un'ulteriore ottimizzazione si ottiene da una diversa formulazione del problema: si usa infatti un approccio *incrementale* che comporta la collocazione delle regine una ad una (con questo metodo si mescola in qualche modo la fase di generate and test).

```
regine(N, Allocazione) ←
    lista_vars(N, ListaValori),
    aggiungi_regina(ListaValori, [], Allocazione).
%%%
aggiungi_regina(NonMesse, Sicure, Allocazione) ←
    seleziona(Q, NonMesse, Rimanenti),
    notattacca(Q, Sicure),
    aggiungi_regina(Rimanenti, [Q|Sicure], Allocazione).
aggiungi_regina([], Allocazione, Allocazione).
```

ESERCIZIO 7.1. Il predicato builtin statistics, usato come statistics(walltime, [T, _]) restituisce in T un tempo assoluto (compresi i garbage collection. Per avere solo tempi effettivi va usato runtime invece di walltime) da quando la sessione di Prolog è iniziata. Si testino i tre approcci al problema delle N regine.

ESERCIZIO 7.2. Si risolva il problema analogo di posizionare M cavalli in una scacchiera $N \times N$.

4. Map Coloring

Si deve colorare una cartina geografica usando un dato insieme di colori in modo che non si abbiano mai due regioni confinanti con lo stesso colore.

Si può rappresentare tale situazione con una struttura a grafo composta da N nodi (che costituiscono le N regioni) e da archi che rappresentano la relazione di confinanza (ad esempio, se nell'insieme degli archi vi è la coppia $[X_1, X_2]$ significa che la regione X_1 confina con la regione X_2).

```
coloring(Nodi, Archi, Colori) ←
    funzione(Nodi, Colori),
    verifica(Archi).
```

Definiamo quindi il predicato verifica:

```
verifica([]).
    verifica([[X,Y]|R]) ←
    X\ = Y,
    verifica(R).
```

Il problema computazionale è dato dalla fase (funzione(Nodi, Colori) che genera tutte le possibili configurazioni che sono: $|\text{Colori}|^{|\text{Nodi}|}$.

Possiamo sostituire alla chiamata a funzione la procedura assegna(Archi, Colori) così implementata:

```
assegna([], _).
assegna([[X,Y]|R], Colori) ←
    member(X, Colori),
    member(Y, Colori),
    X\ == Y,
    assegna(R, Colori).
```

Il test di verifica è ora banalmente vero. Pertanto può essere omesso. Questa seconda soluzione, più efficiente, non assegna colore ad eventuali nodi isolati. Ma per questi qualunque colore va bene.

Esercizio 7.3. Verificare i diversi tempi di esecuzione dei due metodi su grafi tratti da cartine reali.

5. Commesso Viaggiatore

Il problema del commesso viaggiatore (*TSP*, dall'inglese travelling salesperson problem) è il problaema di visitare un insieme di città esattamente una volta ritornando al punto di partenza, il tutto con un costo inferiore ad una certa costante.

Rappresentiamo i dati mediante un grafo diretto con archi pesati, rappresentati da fatti del tipo:

```
arco(N1, Costo, N2).
```

Nodi sarà l'insieme dei nodi da attraversare. I rappresenta il nodo di partenza (e di arrivo!), Path sarà l'itinerario calcolato, mentre K è il costo del cammino massimo accettabile. Descriviamo la soluzione generate & test:

```
tsp(I, Nodi, [I|Path], K) ←
    permutazione(Nodi, [I|Path]),
    verifica_lunghezza(I, [I|Path], K).
```

Ove verifica_lunghezza viene definito come:

```
verifica_lunghezza(I, Path, C) ←
    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, aggiungiamo gli archi uno alla volta, tagliando la ricerca quando si supera il valore K.

```
tsp(I, Nodi, [I|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.
```

6. Predicati di secondo ordine

6.1. Collezionare soluzioni. Nel Capitolo precedente abbiamo visto come sia possibile, usando il fail e il write stampare l'elenco degli X per cui vale il predicato $p(X)$. Per far ciò tuttavia abbiamo sfruttato del side-effect e non abbiamo di fatto collezionato le risposte in una struttura dati tradizionale (per esempio una lista).

In Prolog sono presenti tre primitive meta-logiche (o di logica del secondo ordine) che ci consentono di fare ciò: la findall, bagof e la setof. Cerchiamo di caratterizzare la loro semantica mediante due esempi.

Consideriamo il seguente programma:

```
p(a).
p(a).
p(b).
p(a).
```

e analizziamo gli outputs dei seguenti goals:

$$\leftarrow \mathsf{findall}(\mathsf{X}, \mathsf{p}(\mathsf{X}), \mathsf{L}).$$
$$\mathsf{L} = [\mathsf{a}, \mathsf{a}, \mathsf{b}, \mathsf{a}]$$
$$\leftarrow \mathsf{bagof}(\mathsf{X}, \mathsf{p}(\mathsf{X}), \mathsf{L}).$$
$$\mathsf{L} = [\mathsf{a}, \mathsf{a}, \mathsf{b}, \mathsf{a}]$$
$$\leftarrow \mathsf{setof}(\mathsf{X}, \mathsf{p}(\mathsf{X}), \mathsf{L}).$$
$$\mathsf{L} = [\mathsf{a}, \mathsf{b}]$$

findall trova tutte le risposte di p(X); le soluzioni sono fornite (con eventuali ripetizioni) nell'ordine in cui l'SLD risoluzione di Prolog le restituirebbe. La differenza con il predicato bagof viene chiarita nell'esempio seguente. Infine il predicato setof fornisce una lista ordinata senza ripetizioni delle soluzioni.

Consideriamo ora il seguente programma:

$$\mathsf{q}(\mathsf{a}, \mathsf{b}).$$
$$\mathsf{q}(\mathsf{a}, \mathsf{c}).$$
$$\mathsf{q}(\mathsf{a}, \mathsf{c}).$$
$$\mathsf{q}(\mathsf{b}, \mathsf{d}).$$
$$\mathsf{q}(\mathsf{b}, \mathsf{e}).$$

Analizziamo il risultato fornito dai goals:

$$\leftarrow \mathsf{findall}(\mathsf{Y}, \mathsf{q}(\mathsf{X}, \mathsf{Y}), \mathsf{L}).$$
$$\mathsf{L} = [\mathsf{b}, \mathsf{c}, \mathsf{c}, \mathsf{d}, \mathsf{e}]$$

In questo caso in qualche modo perdo la relazione fra le variabili X e Y (maschero la variabile X). In un certo senso è come l'insieme (a meno di ordine e ripetizioni) $\{Y : \exists X\, q(X, Y)\}$.

$$\leftarrow \mathsf{bagof}(\mathsf{Y}, \mathsf{q}(\mathsf{X}, \mathsf{Y}), \mathsf{L}).$$
$$\mathsf{X} = \mathsf{a}, \mathsf{L} = [\mathsf{b}, \mathsf{c}, \mathsf{c}];$$
$$\mathsf{X} = \mathsf{b}, \mathsf{L} = [\mathsf{d}, \mathsf{e}]$$

In questo caso le relazioni tra le variabili si mantiene. E' come trovare, dato X, l'insieme $\{Y : q(X, Y)\}$.

$$\leftarrow \mathsf{setof}(\mathsf{Y}, \mathsf{q}(\mathsf{X}, \mathsf{Y}), \mathsf{L}).$$
$$\mathsf{X} = \mathsf{a}, \mathsf{L} = [\mathsf{b}, \mathsf{c}];$$
$$\mathsf{X} = \mathsf{b}, \mathsf{L} = [\mathsf{d}, \mathsf{e}].$$

In quest'ultimo caso si hanno i medesimi risultati ottenuti con il predicato bagof con l'eliminazione però di eventuali ripetizioni.

Si vuol comunque far notare come setof non sia un un vero operatore insiemistico. Infatti alla richiesta:

$$\leftarrow \mathsf{setof}(\mathsf{Y}, \mathsf{q}(\mathsf{a}, \mathsf{Y}), \mathsf{L}).$$

viene fornito come output L=[b,c] (e ciò corrisponde al risultato atteso!), ma alla richiesta :

$$\leftarrow \mathsf{setof}(\mathsf{Y}, \mathsf{q}(\mathsf{a}, \mathsf{Y}), [\mathsf{c}, \mathsf{b}]).$$

il risolutore fallisce fornendo *no* in output, anche se dal punto di vista insiemistico i due insiemi $\{b, c\}$ e $\{c, b\}$ sono equivalenti.

ESERCIZIO 7.4. Definire un predicato di ordinamento con eliminazione di ripetizioni di una lista utilizzante il setof.

6.2. Alterazione del Programma: assert e retract. Mediante i comandi

- assert (asserta, assertz)
- retract

è possibile aggiungere (assert) e rimuovere (retract) fatti e regole dal programma dinamicamente. In particolare asserta(C) aggiunge la clausola C in testa al programma. assertz(C) in fondo al programma. retract(C) toglie, una alla volta, tutte le clausole che unificano con C dal programma.

Un esempio di uso è la definizione di un contatore:

```
inizializza_contatore ←
    assert(time(0)).
incrementa_contatore ←
    retract(time(T)),
    T1 is T + 1,
    assert(time(T1)).
val_contatore(T) ←
    time(T).
```

7. Metainterpreti

Il comando clause(H, B) ha successo se esiste una clausola ridenominata $A \leftarrow C$ da P tale che unify(H, A) ha successo, con mgu θ. La risposta calcolata è la sostituzione θ (in altri termini, se H è un atomo da considerarsi come input, viene restituito $B\theta$).

Il comando suddetto ci servirà per definire degli interpreti di Prolog in Prolog (altresì detti metainterpreti).

Un metainterprete è un programma basato su un predicato risolutore (chiamiamolo solve) tale che

$$\leftarrow \text{solve}(G)$$

deve simulare il comportamento dell'SLD risoluzione sul goal $\leftarrow G$.

Il più semplice metainterpete per programmi definiti è il seguente:

```
solve(G) ← G
```

Il programma è un metainterprete e funziona perfettamente (e banalmente). Si osservi come G sia a sinistra visto come un termine (un dato) e a destra invece come un goal.

Il seguente invece entra un po' più in dettaglio sul significato della SLD-risoluzione. E' anche noto come *vanilla metainterpreter*.

```
solve(true).
solve((A, B)) ←
    solve(A),
    solve(B).
solve(A) ←
    clause(A, B),
    solve(B).
```

Se vogliamo estenderlo per trattare correttamente anche alcuni predicati built-in, allora si deve aggiungere:

```
builtin(A is B).
builtin(A < B).
builtin(write(A)).
builtin(A = B).
solve(A) ←
    builtin(A),
    A.
```

Similmente, possiamo generare un metainterprete che possa compiere delle funzioni di tracing per coaudiuvare la fase di debugging:

```
solvetrace(true).
solvetrace((A, B)) ←
    solvetrace(A),
    solvetrace(B).
solvetrace(A) ←
    write('selezionato atomo '), write(A), nl,
    clause(A, B),
    write('selezionata clausola '), write(A), write('←'), write(B), nl,
    solve(B).
solvetrace(A) ←
    builtin(A),
    write('built-in '), write(A), nl,
    A.
```

ESERCIZIO 7.5. Si scrivano un metainterprete per ciascuno dei seguenti requisiti:

(1) effettua SLD con regola di selezione destra anziché sinistra
(2) effettua la visita dell'albero SLD in ampiezza anziché in profondità
(3) date clausole asserite con probabilità, per esempio:

```
clausolap(padre(a, b, 0.9)).
clausolap((padre(X, Y, P) :− genitore(X, Y, P1), PisP1 ∗ 0.5)).
```

il metainterprete deve cercare di inferire la probabilità di un goal di essere dedotto.

8. Quiz

In questa sezione si cerca di fornire un metainterprete in grado di risolvere dei rompicapo logici.

Ragioniamo su un esempio, ma tutto viene realizzato in modo da essere generalizzabile.

In una gara tre amici arrivano primo, secondo e terzo.
Ciscuno di loro pratica uno sport diverso.
Michele gioca a pallacanestro ed è arrivato meglio dell'americano.
Simone, l'inglese è arrivato meglio del tennista.
Il calciatore è arrivato primo.
Si chiede:

(1) Chi è l'australiano?
(2) A cosa gioca Riccardo?

Iniziamo col scegliere una rappresentazione per il gioco. Diamo il nome test1 a tale gioco. La struttura del problema è una lista di tre amici.

```
structure(test1, [amico(Nome1, Nazione1, Sport1),
                  amico(Nome2, Nazione2, Sport2),
                  amico(Nome3, Nazione3, Sport3)]).
```

Vi sono poi gli indizi (clues), in questo caso 3, per il quiz:

```
clues(test1, Amici, [ (maggiore(U1C1, U2C2, Amici),
                       nome(U1C1, michele),
                       sport(U1C1, pallacanestro),
                       nazione(U2C1, america)),          %%%fine clue1
                      (maggiore(U1C2, U2C2, Amici),
                       nome(U1C2, simone),
                       sport(U2C2, tennis),
                       nazione(U1C2, inghilterra)),      %%%fine clue2
                      (primo(UC3, Amici),
                       sport(UC3, calcio))]).            %%%fine clue3
```

E per finire le domande (due) al quiz.

```
queries(test1, Amici, [ member(Q1, Amici),
                        nome(Q1, Nome),
                        nazione(Q1, australia), %%%fine query1
                        member(Q2, riccardo),
                        sport(Q2, Sport)], %%%fine query2
                        [[Nome,'australiano'], ['Riccardo gioca a', Sport]]).%%%Output atteso
```

Il metainterprete generale per risolvere la famiglia di quiz è il seguente:

```
test_puzzle(Name, Solution) ←
    structure(Name, Structure),
    clues(Name, Structure, clues),
    queries(Name, Structure, queries, Solution), %%%fin qui recuperati i dati del problema
    solve_puzzle(Clues, Queries, Solution).
%%%
solve_puzzle(Clues, Queries, Solution) ←
    solve(Clues),
    solve(Queries).
%%%
solve([]).
solve([A|R]) ←
    A,
    solve(R).
```

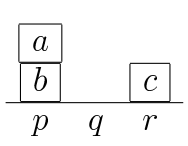
Predicati ausiliari per il problema:

```
maggiore(A, B, [A, B, C]).
maggiore(A, C, [A, B, C]).
maggiore(B, C, [A, B, C]).
primo(A, [A, B, C]).
nome(amico(Nome, Nazione, Sport), Nome).
nazione(amico(Nome, Nazione, Sport), nazione).
sport(amico(Nome, Nazione, Sport), sport).
```

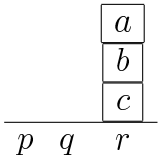
9. Mondo dei blocchi

Il problema del planning su un mondo di blocchi può essere cos formulato: da una situazione iniziale in cui vi è una serie di blocchi (anche sovrapposti) e una serie di posti dove i blocchi possono essere messi si deve arrivare ad una specificata configurazione finale posti-blocchi. Il vincolo è che si può spostare solo un blocco alla volta e si può scegliere solo un blocco "clear" ovvero con nessun altro blocco sopra.

Anche in questo caso, ragioniamo su un esempio e ci convinciamo della facilità di generalizzazione.



Stato iniziale



Stato finale

Innanzitutto rappresentiamo le entità in gioco mediante dei fatti:

```
block(a).
block(b).
block(c).
place(p).
place(q).
place(r).
```

La rappresentazione delle due configurazioni invece:

$$[\mathsf{on(a,b)}, \mathsf{on(b,p)}, \mathsf{on(c,r)}], [\mathsf{on(a,b)}, \mathsf{on(b,c)}, \mathsf{on(c,r)}]$$

L'idea della risoluzione potrebbe essere quella di fermarsi se si è già nella configurazione finale. Altrimenti scegliere una mossa e reiterare. Si deve poi agire sul criterio di scelta della mossa. Bisogna innanzitutto stare attenti a non tornare mai su uno stato già ragiunto. Vedremo poi come inserire qualche altra strategia.

NOTA 7.6. E' fondamentale affinché la seguente soluzione funzioni, che l'ordine dei blocchi nelle liste che contengono gli stati sia coerente. Altrimenti potremmo non accorgerci di essere già giunti allo stato finale.

```
transform(Siniziale, Sfinale, Plan) ←
    tranform(Siniziale, Sfinale, [Siniziale], Plan).
%%%
transform(Sfinale, Sfinale, []).
transform(S, Sfinale, Visitati, [Azione|Plan]) ←
    choose_action(Azione, S, Sfinale).
    update(Azione, S, Sint).
    nonmember(Sint, Visitati).
    tranform(Sint, Sfinale, [S|Visitati], Plan).
%%%
choose_action(Azione, S, Sfinale) ←                    (*)
    legal_action(Azione, S).
%%%
legal_action(to_place(Block, Y, Place), S) ←
    block(Block),
    place(Place),
    on(Block, Y, S),
    clear(Block, State),
    clear(Place, State).
legal_action(to_block(Block, Y, Block2), S) ←
    block(Block),
    block(Block2),
    Block\ = Block2,
    on(Block, Y, S),
    clear(Block, State),
    clear(Block2, State).
```

La definizione dei predicati on e clear è lasciata per esercizio.

Un'ottimizzazione di tale codice che permette una migliore strategia risolutiva si può ottenere sostituendo la regola (∗) con le due clausole (e le due ausiliarie):

```
choose_action(Azione, S, Sfinale) ←
    suggest(Action, Sfinale),
    legal_action(Azione, S).
choose_action(Azione, S, Sfinale) ←
    legal_action(Azione, S).
suggest(to_place(X, Y, Z), Sfinale) ←
    member(on(X, Z), Sfinale),
    place(Z).
suggest(to_block(X, Y, Z), Sfinale) ←
    member(on(X, Z), Sfinale),
    block(Z).
```

Si dà priorità alle azioni che producono una situazione presente nello stato finale.

ESERCIZIO 7.7. Si scriva un programma di planning per il seguente problema dei cinque mariti gelosi:

In seguito ad un'alluvione 5 coppie si trovano su di un'isola temporanea. Possiedono una barca che può portare 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.

ESERCIZIO 7.8. Si scriva un programma di planning per il noto problema della capra e cavolo.

10. Giochi

Lo schema generale di un gioco può essere descritto dal seguente programma:

```
play(Game) ←
    inizializza(Game, Position, Player),
    display_game(Position, Player),
    gioca(Position, Player, Result).
%%%
gioca(Position, Player, Result) ←
    gameover(Position, Player, Solution),
    !,
    announce(Result).
gioca(Position, Player, Result) ←
    choose_move(Position, Player, Move),
    move(Move, Position, Positon1),
    display_game(Position1, Player),
    next_player(Player, Player1),
    !,
    gioca(Position1, Player1, Result).
```

Ogni singolo predicato dipende necessariamente dal gioco in esame. La variabile Gioco permette di selezionare il gioco in esame (per esempio scacchi, briscola, ecc.). inizializza negli scacchi sarà semplicemente la collocazione iniziale standard dei pezzi. Nei giochi di carte sarà simulato il mescolare delle carte. Verrà poi scelto chi inizia (ad esempio user vs. computer) e assegnato alla variabile Player. display_game stamperà in qualche modo lo scenario di gioco. gameover assicura che la partita è finita. C'è un cut che sta a significare che ormai non si torna più indietro. Nella seconda clausola di gioca, una volta scelta la mossa con choose_move, si muove, si sceglie il prossimo giocatore e non si torna più indietro. Le definizioni dei vari predicati possono essere lunghe per le complesse regole del gioco, ma, salvo per choose_move, tutto è deterministico e privo di difficoltà.

Diversa la situazione per choose_move, che rappresenta la parte intelligente dell'intero algoritmo. Il suo schema può essere il seguente:

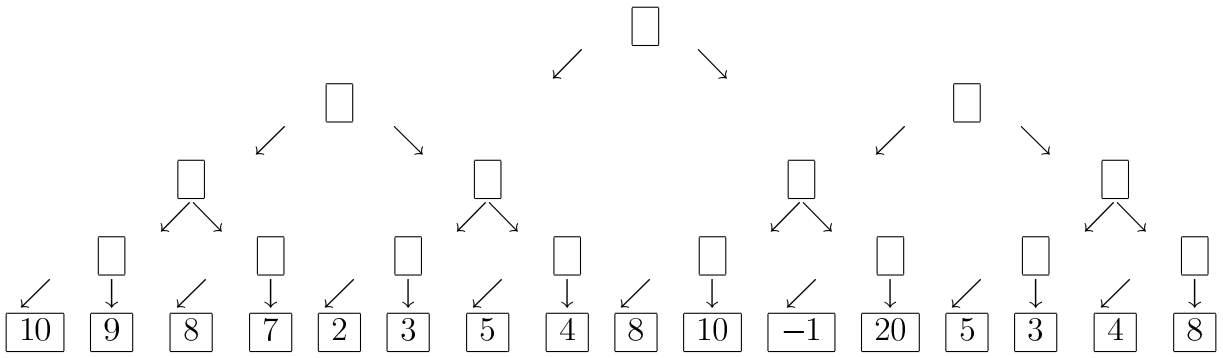
```
choose_move(Position, Player, Move) ←
    user_player(Player),
    accetta_da_utente(Player, Move).
choose_move(Position, Player, Move) ←
    computer_player(Player),
    setof(M, move(position, M), Moves),
    evaluate_and_choose(Position, Moves, Move).
```

Dato dunque l'insieme di mosse successive possibili (per esempio tre nella briscola, molte di più per gli scacchi) bisogna saper sceglierne una. Bisogna innanzitutto saper assegnare un valore ad ogni posizione.

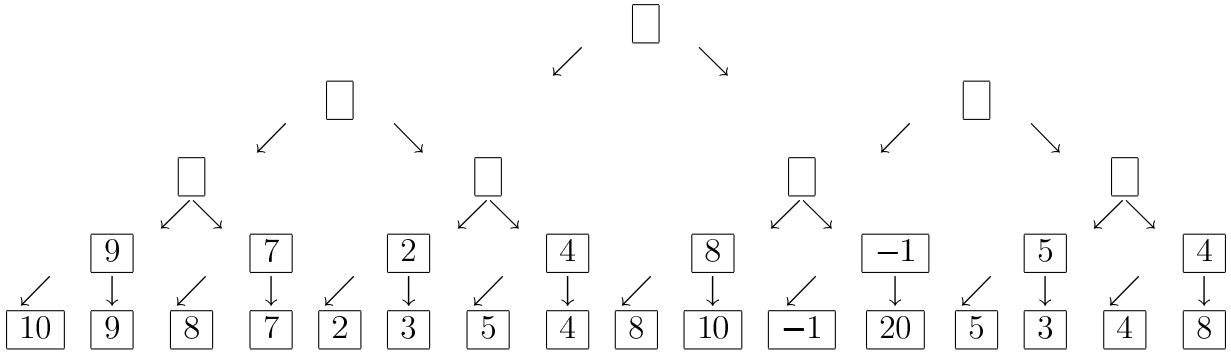
A questo punto un primo modo per definire la evaluate_and_choose è quello di valutare le nuove posizioni associate ad ognuna delle mosse presenti in Moves e scegliere quella che massimizza il guadagno.

Un modo un po' più intelligente è quello di guardare avanti di un certo numero D di mosse. Poi di applicare un algoritmo detto mini-max.

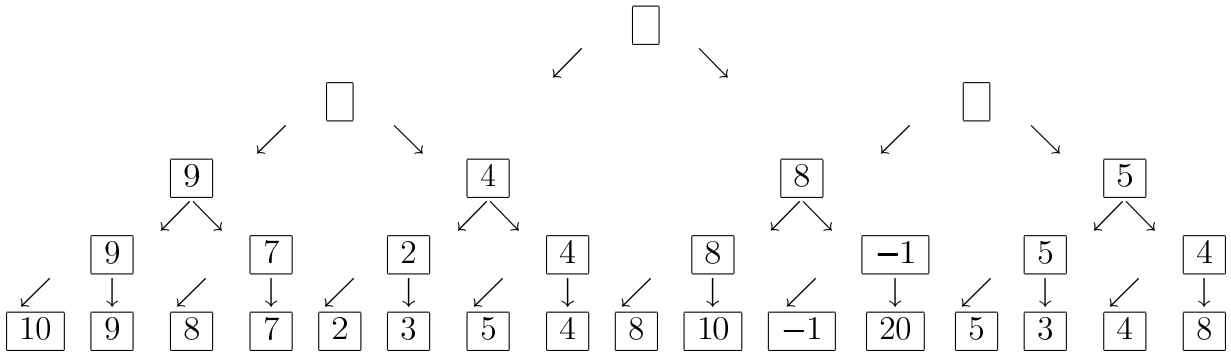
Consideriamo un gioco a due giocatori e $D = 2$. Creiamo l'albero ottenuto dalle mosse (calcolatore, utente, calcolatore, utente) e associamo il costo ad ogni posizione finale raggiunta.



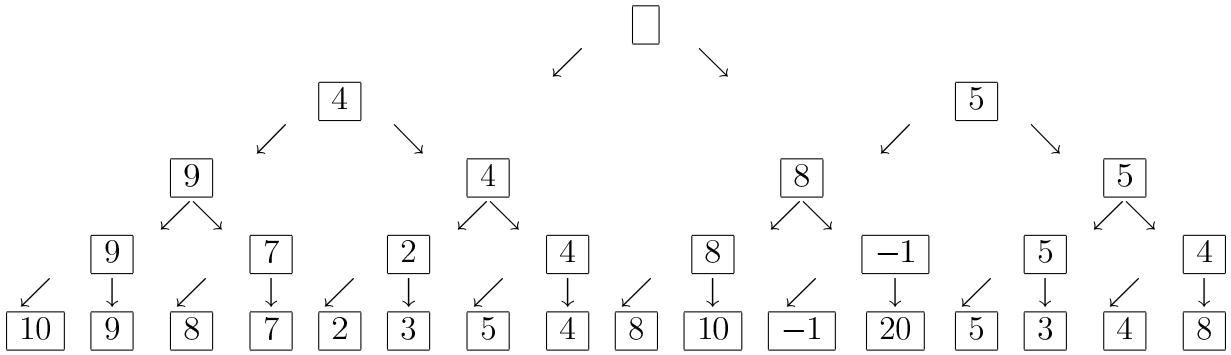
Risaliamo dunque dalle foglie verso la radice. L'ultima mossa è dell'uomo. Il programma assume che faccia la più intelligente, ovvero quella che resituisce una situaizone meno vantaggiosa per lui. Pertanto associa ai nodi vicini alle foglie un valore che indica quale sia il punteggio peggiore per lui in seguito al raggiungimento di tale nodo dell'albero:



Il passo precedente era del calcolatore e dunque lui cerca di raggiungere un nodo in cui il valore sia il massimo possibile. I nuovi valori sono pertanto:



A questo punto si ragione di nuovo in termini della mossa dell'utente. L'utente astuto cercherà di minimizzare tali valori. si giungerà a:



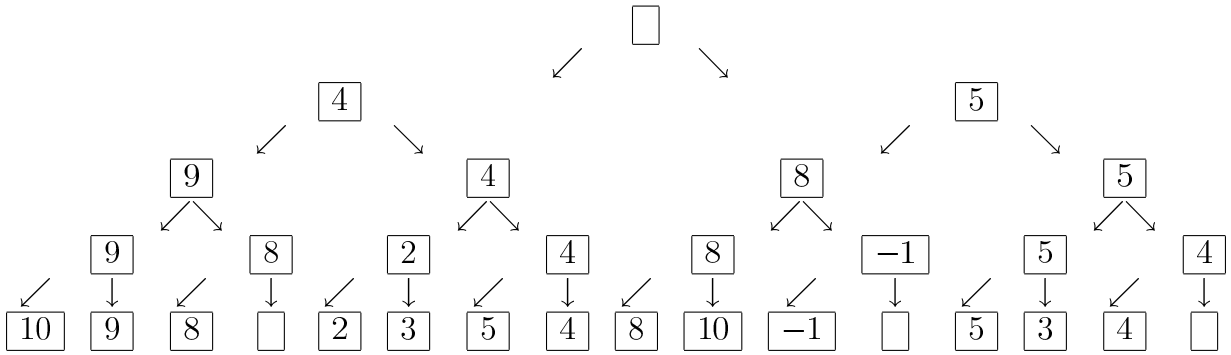
Ora il calcolatore è in grado di scegliere: eseguirà la mossa di destra che gli garantisce (guardando avanti di sole due mosse) di massimizzare il suo valore, assumendo che sia lui che l'uomo continuino poi a giocare in modo massimale.

Risalendo il livello dell'albero, una volta viene calcolato il massimo, un'altra il minimo. L'algoritmo è detto minimax. Entrambi i passaggi si possono implementare con la stessa routine (ad esempio quella del massimo) avendo cura di richiamare ricorsivamente la procedura con i valori con segno invertito. Per la procedura in Prolog si veda [**7**].

NOTA 7.9. Con un buon programma in un normale computer, si può effettuare una ricerca di circa 1000 posizioni al secondo. Nei tornei di scacchi, si dispone di circa 150 secondi per mossa, perciò possiamo guardare 150000 posizioni; il fattore di ramificazione è circa 35, perciò il nostro programma sarà in grado di considerare solo tre o quattro strati e giocherà al livello di un perfetto principiante.

Una semplice ottimizzazione del metodo sopra è una tecnica detta *alfa-beta pruning* che taglia l'albero di ricerca quando possibie. Supponiamo di rimandare il più possibile la valutazione dei nodi foglia raggiunti. Se pensiamo all'etichettatura precedente e pensiamo di farla in profondità, dopo aver valutato i due nodi più a sinistra (10 e 9) e calcolato che il minimo è 9, scendiamo verso il terzo trovando 8. A questo punto non serve più andare a scendere sul quarto nodo dal valore 7 in quanto sappiamo già che dall'altra parte sapremmo arrivare a 8 e pertanto sarà quello il cammino che seguiremo.

Nel seguente albero le caselle in bianco sono quelle che non serve calcolare. Anche in questo caso il codice Prolog si può trovare in dettagio su [**7**].



CHAPTER 8

Programmi logici con vincoli

1. Sintassi e semantica operazionale

L'insieme Π dei simboli di predicato viene partizionato come $\Pi = \Pi_C \cup \Pi_P$ con $\Pi_C \cap \Pi_P = \emptyset$. Π_P sono i simboli predicativi definiti nel programma e Π_C è un insieme di simboli di predicato di costraint. Π_C contiene il predicato di uguaglianza "=".

DEFINIZIONE 8.1. Un *constraint primitivo* è un letterale sull'alfabeto $\langle \Pi_C, \mathcal{F}, \mathcal{V} \rangle$. Assumiamo che true e false siano anche constraint primitivi.[1] Un *constraint* è una congiunzione di constraint primitivi.

Se $p \in \Pi_P$, e $t_1, \ldots, t_n$ sono termini, $p(t_1, \ldots, t_n)$ è detto *atomo di programma*.

ESEMPIO 8.2. I seguenti sono constraint primitivi

$$X = Z \quad X \neq Z \quad X \neq Y \quad 0 < X \quad 0 \geq X \quad X \in Y \quad A \subseteq B$$

Riformuliamo ora le definizioni di programmi, goal, etc.

DEFINIZIONE 8.3. $\leftarrow \bar{B}$ è un *goal CLP* se $\bar{B}$ è una congiunzione di atomi di programma e di constraint primitivi.

ESEMPIO 8.4. $X \neq Y, p(X_1, X_2), X_1 \leq X, Y_2 > Y$ è un goal CLP. $p(X, Y), \neg q(Y, Z)$ non è un CLP goal.

DEFINIZIONE 8.5. Un *fatto CLP* è $p(t_1, \ldots, t_n)$, dove $p \in \Pi_P$, t_i termini. Una *regola CLP* è $p(t_1, \ldots, t_n) \leftarrow \bar{B}$ dove $\leftarrow \bar{B}$ è un CLP goal, $p \in \Pi_P$, t_i termini.

Si osservi che una regola CLP non è necessariamente una clausola di Horn. Ad esempio

$$p(X, Y) \leftarrow X \neq Y, X < Z, q(X, Z)$$

equivale a

$$p(X, Y) \vee (X = Y) \vee \neg(X < Z) \vee \neg q(X, Z)$$

dove ci sono due letterali positivi.

Definiamo il concetto di stato.

DEFINIZIONE 8.6. Uno *stato* è una coppia $\langle G \,|\, C \rangle$ dove G è un CLP goal e C è un constraint (anche detto *constraint store*).

Uno *stato* è di *successo* se ha la forma $\langle \leftarrow \Box | C \rangle$ e $solv(C) \neq \texttt{false}$.

Uno *stato* è di *fallimento* se ha la forma $\langle G | C \rangle$ e $solv(C) =$ false oppure se G è una congiunzione di atomi di programma e non vi è nessuna clausola la cui testa abbia simboli di predicato di programma presenti nel goal.

[1] Potremmo anche definirli come $a = a$ e $a \neq b$, rispettivamente.

DEFINIZIONE 8.7. Un *passo di derivazione* CLP $\langle G_1 \,|\, C_1\rangle \Rightarrow \langle G_2 \,|\, C_2\rangle$ 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:

(1) se L_1 è un constraint primitivo, allora $C_2 = L_1 \wedge C_1$, inoltre se $solv(C_2) = \mathsf{false}$, allora $G_2 = \leftarrow \square$, altrimenti $G_2 = \leftarrow L_2, \ldots, L_n$.
(2) se L_1 è un atomo di programma $p(t_1, \ldots, t_n)$ e $p(s_1, \ldots, s_n) \leftarrow \bar{B}$ è una rinomina di una regola di P, allora $G_2 = \leftarrow t_1 = s_1, \ldots, t_n = s_n, \bar{B}, L_2, \ldots, L_n$ mentre $C_2 = C_1$.

Una *derivazione per uno stato* S_0 è una sequenza massimale di passi di derivazione

$$S_0 \Rightarrow S_1 \Rightarrow \cdots$$

Una *derivazione per un CLP goal* G è una derivazione per lo stato $\langle G \,|\, \mathsf{true}\rangle$.

Una *derivazione* $S_0 \Rightarrow \cdots \Rightarrow S_n$ è di *successo* se S_n è di successo. La *risposta calcolata* è $simpl(C_n, vars(S_0))$.

Una *derivazione* $S_0 \Rightarrow \cdots \Rightarrow S_n$ è di *fallimento* se S_n è di fallimento.

Tutta la procedura si basa su *solv* e *simpl*. Per esse non viene data una definizione rigida. Cerchiamo di capire il loro significato su degli esempi. Si consideri il dominio $\mathbb{R}$:

$$\begin{gathered} solv(X < 5, 4 < X, X \neq 0) \neq \mathsf{false} \\ \Downarrow simpl \\ 4 < X, X < 5 \end{gathered}$$

Mentre nel dominio $\mathbb{N}$:

$$solv(X < 5, 4 < X, X \neq 0) = \mathsf{false}$$

Lo stesso vincolo dà risultati diversi a seconda del modello: si parla di $CLP(\mathbb{N}), CLP(\mathbb{R})$, $CLP(FD)$ (su domini finiti), $CLP(\mathcal{SET})$ (con vincoli di tipo insiemistico), eccetera.

Vediamo invece un esempio più concreto dell'uso di *simpl*: Sia $CLP(\mathbb{N})$:

$$\begin{gathered} \langle \leftarrow p(X) | \mathsf{true}\rangle \\ \downarrow \\ \vdots \\ \downarrow \\ \langle \leftarrow \square \,|\, X = f(X_1), X_2 = f(X_3), X_2 \neq X_3, Y > W, X_1 = f(X_4), X_4 < 3, X_4 \neq 4\rangle \\ \downarrow simpl \\ X = f(f(X_4)), X_4 < 3 \end{gathered}$$

Poiché in Π c'è sempre l'uguaglianza, in *simpl* vi dev'essere la capacità di effettuare l'unificazione sintattica (in certi casi quella semantica—E unificazione).

ESEMPIO 8.8. Sia il programma P:

$$\begin{gathered} \mathsf{num(0).} \\ \mathsf{num(s(X))} \leftarrow \mathsf{num(X).} \end{gathered}$$

E si consideri la derivazione:

$$
\begin{array}{cc}
\langle \leftarrow num(s(s(0)))|\mathsf{true}\rangle & \\
| & num(s(X_1)) \leftarrow num(X_1) \\
\langle \leftarrow s(X_1) = s(s(0)), num(X_1)|\mathsf{true}\rangle & \\
| & \\
\langle \leftarrow num(X_1)|s(X_1 = s(s(0))) & solv(s(X_1) = s(s(0))) \neq \mathsf{false} \\
| & \ulcorner\neg s(X_2)) \leftarrow num(X_2) \\
\langle \leftarrow X_1 = s(X_2), num(X_2)|s(X_1) = s(s(0))\rangle & \\
| & \\
\langle \leftarrow num(X_2)|\underbrace{X_1 = s(X_2), s(X_1) = s(s(0))}_{a}\rangle & solv(a) \neq \mathsf{false} \\
| & \\
\langle \leftarrow X_2 = 0|X_1 = s(X_1), s(X_1) = s(s(0))\rangle & \\
| & \\
\langle \leftarrow \Box|\underbrace{X_2 = 0, X_1 = s(X_2), s(X_1) = s(s(0))}_{b}\rangle & solv(b) \neq \mathsf{false}
\end{array}
$$

Si osservi come si sia simulata (ma con granularità più fine) una SLD derivazione. *solv* si occuperà di effettuare l'unificazione.

simpl dovrà poi fornire in qualche modo un output. Poichè $vars(G_0) = \emptyset$ non si fornirà nulla. In altri termini, *simpl* agisce su

$$X_2 = 0, X_1 = s(X_2), s(X_1) = s(s(0))$$

che semplifica in:

$$\begin{array}{c} X_2 = 0 \\ X_1 = s(0) \end{array}$$

ma restringendosi a $vars(G_0) = \emptyset$ non fornisce nulla.

Esempio 8.9. 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:

$$\langle \leftarrow P(X,Y) | \text{true} \rangle$$
$$\Big| \qquad P(X_1,Y_1) \leftarrow X_1 = Y_1 + 2, Y_1 \geq 0, q(X_1,Y_1)$$
$$\langle \leftarrow X = X_1, Y = Y_1, X_1 = Y_1 + 2, Y_1 \geq 0, q(X_1,Y_1) | \text{true} \rangle$$
$$\Big| \qquad \text{in 4 passi}$$
$$\langle q(X_1,Y_1) | \underbrace{Y_1 \geq 0, X_1 = Y_1 + 2, Y = Y_1, X = X_1}_{(a)} \rangle \qquad solv(a) \neq \text{false}$$
$$\Big| \qquad q(X_2,Y_2) \leftarrow X_2 \leq 1, r(X_2,Y_2)$$
$$\langle \leftarrow X_1 = X_2, Y_1 = Y_2, X_2 \leq 1, r(X_2,Y_2) | (a) \rangle$$
$$\Big|$$
$$\langle \leftarrow r(X_2,Y_2) | \underbrace{X_2 \leq 1, X_1 = X_2, Y_1 = Y_2, (a)}_{(b)} \rangle \qquad solv(b) = \text{false}$$

Quindi viene fatto backtracking fino a $\langle q(X_1,Y_1) | Y_1 \geq 0, X_1 = Y_1 + 2, Y = Y_1, X = X_1 \rangle$ e si ricomincia ...

$$\langle q(X_1,Y_1) | \underbrace{Y_1 \geq 0, X_1 = Y_1 + 2, Y = Y_1, X = X_1}_{(a)} \rangle \qquad solv(a) \neq \text{false}$$
$$\Big| \qquad q(X_2,Y_2) \leftarrow X_2 \leq 3, r(X_2,Y_2)$$
$$\langle \leftarrow X_2 = X_1, Y_2 = Y_1, X_2 \leq 3, r(X_2,Y_2) | (a) \rangle$$
$$\Big| \qquad \text{in 3 passi}$$
$$\langle \leftarrow r(X_2,Y_2) | \underbrace{X_2 = X_1, Y_2 = Y_1, X_2 \leq 3, (a)}_{b} \rangle \qquad solv(b) \neq \text{false}$$
$$\Big| \qquad r(X_3,1)$$
$$\langle \leftarrow X_2 = X_3, Y_2 = 1 | (b) \rangle$$
$$\Big| \qquad \text{in 2 passi}$$
$$\langle \leftarrow \Box | X = Y + 2, Y > 0, X \leq 3, Y = 1 \rangle$$
$$\downarrow$$
$$X = 3, 1 > 0, 3 \leq 3, Y = 1$$

Che con *simpl* ottengo $X = 3, Y = 1$

2. $CLP(\mathbb{R})$ in SICStus

Per usare il pacchetto dei risolutori di vincoli sui reali (similmente sui razionali) bisogna richiamare:[2]

```
:−use_module(library(clpr)).
```

Per far capire che un vincolo riguarda variabili reali, va messo tra parentesi grafa. Ad esempio:

```
:−{V = I ∗ R, V = 2, R = 5}.
```

restituisce:

```
I = 0.4, R = 5.0, V = 2.0
```

[2]The full $CLP(\mathbb{R})$ distribution, including a stand-alone manual and an examples directory that is possibly more up to date than the version in the SICStus Prolog distribution, is available from: http://www.ai.univie.ac.at/clpqr/.

Alcuni simboli predicativi di constraint ammessi sono:

$$=, <, >, =<, >=, = \backslash =$$

Alcuni tra i simboli di funzione interpretati ammessi sono:

$$+, -, *, /, \mathsf{abs}, \mathsf{sin}, \mathsf{cos}, \mathsf{tan}, \mathsf{pow} \text{ (oppure } \hat{}), \mathsf{min}, \mathsf{max}$$

Esempi di goals:

$$:-\{\mathsf{X} = \mathsf{pow}(2,3)\}.$$

restituisce $\mathsf{X} = 8.0$.

$$:-\{\mathsf{X} = \mathsf{min}(2*3, \mathsf{pow}(1,7))\}.$$

restituisce $\mathsf{X} = 1.0$.

Si noti che non c'è la funzione logaritmo.[3] Essa si può comunque definire come predicato:

$$\mathsf{log}(\mathsf{B},\mathsf{X},\mathsf{Y}) :- \{\mathsf{pow}(\mathsf{B},\mathsf{Y}) = \mathsf{X}\}.$$

Vi sono poi delle primitive più caratteristiche, che invitiamo a leggere a fondo nel manuale. Discutiamo solo qui della maximize. Al goal:

$$:-\{2*\mathsf{X}+\mathsf{Y} =< 16, \mathsf{X}+2*\mathsf{Y} =< 11, \mathsf{X}+3*\mathsf{Y} =< 15, \mathsf{Z} = 30*\mathsf{X}+50*\mathsf{Y}\}, \mathsf{maximize}(\mathsf{Z}).$$

Viene risposto:

$$\mathsf{X} = 7.0, \mathsf{Y} = 2.0, \mathsf{Z} = 310.0$$

ovvero viene massimizzata la funzione associata a Z relativamente ai vincoli di dominio per le variabili in gioco.

3. $CLP(\mathcal{FD})$ in SICStus

Il constraint solver sui domini finiti viene chiamato con il comando:

$$:-\mathsf{use_module}(\mathsf{library}(\mathsf{clpfd})).$$

3.1. Fissare/restringere domini a variabili. Per una singola variabile:

$$\mathsf{A\ in\ 1..20}$$

Per una lista $\mathsf{L} = [\mathsf{A}_1, \ldots, \mathsf{A}_n]$ di variabili:

$$\mathsf{domain}(\mathsf{L}, -5, 12)$$

Per usare la comoda notazione ammessa in ECLIPSE:

$$\mathsf{L} :: 1..20$$

basta definire l'operatore :: e aggiungere una clausola definitoria:

$$:-\mathsf{op}(100, \mathsf{xfx}, ::).$$
$$\mathsf{L} :: \mathsf{A..B} :- \mathsf{domain}(\mathsf{L}, \mathsf{A}, \mathsf{B}).$$

[3] Almeno ad oggi, 12 Marzo 2001.

3.2. Operatori matematici. In clpfd si possono usare i soliti operatori arimetici e relativi simboli relazionali. Non vanno messi tra parentesi grafa. L'unica accortezza è di anteporre il simbolo # ai simboli predicativi. Alcuni simboli predicativi di constraint ammessi sono:

$$\#=, \#<, \#>, \#=<, \#>=, \#\backslash =$$

I simboli funzionali ammessi sono:

$$+, -, *, /, \mathsf{mod}, \mathsf{min}, \mathsf{max}, \mathsf{abs}$$

ESEMPIO 8.10. Al goal

$$\mathsf{domain}([\mathsf{A},\mathsf{B},\mathsf{C}],1,4), \mathsf{A}\#<\mathsf{B}, \mathsf{B}\#<\mathsf{C}$$

viene risposto:

$$\begin{array}{l}\mathsf{A\ in\ 1..2},\\ \mathsf{B\ in\ 2..3},\\ \mathsf{C\ in\ 3..4}\end{array}$$

Si noti come la forma di risposta non sia molto espressiva (i vincoli vengono tenuti nascosti). Viene detto che ci sono soluzioni e vengono ristretti i domini delle variabili coinvolte. E' una scelta implementativa di SICSstus. Tuttavia anche la soluzione (sbagliata) $\mathsf{A}=2, \mathsf{B}=2, \mathsf{C}=3$ soddisfa i vincoli forniti ai domini. Si può comunque forzare tale scelta con il comando di asserzione:

$$:-\mathsf{clpfd}:\mathsf{assert}(\mathsf{clpfd}:\mathsf{full_answer}).$$

La risposta ora diventa:

$$\begin{array}{l}\mathsf{clpfd}:'\ \mathsf{t}>=\mathsf{u}+\mathsf{c}'(\mathsf{B},\mathsf{A},1),\\ \mathsf{clpfd}:'\ \mathsf{t}>=\mathsf{u}+\mathsf{c}'(\mathsf{C},\mathsf{B},1),\\ \mathsf{A\ in\ 1..2},\\ \mathsf{B\ in\ 2..3},\\ \mathsf{C\ in\ 3..4}\end{array}$$

Se si fosse ristretto il dominio a 1..3 si sarebbe invece ottenuta la risposta A = 1,B=2,C=3.

3.3. Comandi per istanziare variabili (soddisfacendo i vincoli). 3 Per una singola variabile V:

$$\mathsf{indomain}(\mathsf{V})$$

Alcuni valori del dominio si possono ottenere con

- fd_max usato come $\mathsf{fd_max}(\mathsf{Var},\mathsf{Max})$ che restituisce il massimo valore del dominio ammesso in quel momento.
- fd_min usato come $\mathsf{fd_min}(\mathsf{Var},\mathsf{Max})$ che restituisce il minimo valore del dominio ammesso in quel momento.

Per una lista:

$$\mathsf{labeling}([\,],\mathsf{L})$$

In [] si possono inserire delle opzioni: The Options argument of labeling/2 controls

(1) the order in which variables are selected for assignment (variable choice heuristic),
(2) the way in which choices are made for the selected variable (value choice heuristic), and

(3) whether all solutions or a single, optimal solution should be found.

The options are divided into four groups. One option may be selected per group. Finally, the number of assumptions (choices) made during the search can be collected.

The following options control the order in which the next variable is selected for assignment.

leftmost: The leftmost variable is selected. This is the default.
min: The leftmost variable with the smallest lower bound is selected.
max: The leftmost variable with the greatest upper bound is selected.
ff: The first-fail principle is used: the leftmost variable with the smallest domain is selected.
ffc: The most constrained heuristic is used: a variable with the smallest domain is selected, breaking ties by
(1) selecting the variable that has the most constraints suspended on it and
(2) selecting the leftmost one.
variable(Sel): Sel is a predicate to select the next variable. Given Vars, the variables that remain to label, it will be called as Sel(Vars,Selected,Rest).

Sel is expected to succeed deterministically, unifying Selected and Rest with the selected variable and the remaining list, respectively.

Sel should be a callable term, optionally with a module prefix, and the arguments Vars,Selected,Rest will be appended to it. For example, if Sel is mod:sel(Param), it will be called as mod:sel(Param,Vars,Selected,Rest).

The following options control the way in which choices are made for the selected variable X:

step: Makes a binary choice between X# = B and X#\ = B, where B is the lower or upper bound of X. This is the default.
enum: Makes a multiple choice for X corresponding to the values in its domain.
bisect: Makes a binary choice between X# =< M and X# > M, where M is the midpoint of the domain of X. This strategy is also known as domain splitting.
value(Enum): Enum is a predicate which should narrow the domain of X, possibly but not necessarily to a singleton. It will be called as Enum(X,Rest,BB) where Rest is the list of variables that need labeling except X, and BB is a parameter described below.

Enum is expected to succeed non-deterministically, narrowing the domain of X, and to backtrack one or more times, providing alternative narrowings. In all solutions except the first one, it must call the auxiliary predicate apply_bound(BB) to ensure that branch-and-bound search works correctly.

Enum should be a callable term, optionally with a module prefix, and the arguments X,Rest,BB will be appended to it. For example, if Enum is mod:enum(Param), it will be called as mod:enum(Param,X,Rest,BB).

The following options control the order in which the choices are made for the selected variable X. Not useful with the value(Enum) option:

up: The domain is explored in ascending order. This is the default.
down: The domain is explored in descending order.

The following options control whether all solutions should be enumerated by backtracking or whether a single solution that minimizes (maximizes) X is returned, if one exists.

all: All solutions are enumerated. This is the default.

minimize(X), maximize(X): Uses a branch-and-bound algorithm to find an assignment that minimizes (maximizes) the domain variable X. The labeling should constrain X to become assigned for all assignments of Variables.

Finally, the following option counts the number of assumptions (choices) made during the search:

assumptions(K):

When a solution is found, K is unified with the number of choices made.

For example, to enumerate solutions using a static variable ordering, use:

$$:-\mathsf{constraints}(\mathsf{Variables}),\mathsf{labeling}([],\mathsf{Variables}).\%\text{same as [leftmost,step,up,all]}$$

To minimize a cost function using branch-and-bound search, a dynamic variable ordering using the first-fail principle, and domain splitting exploring the upper part of domains first, use:

$$:-\mathsf{constraints}(\mathsf{Variables},\mathsf{Cost}),\mathsf{labeling}([\mathsf{ff},\mathsf{bisect},\mathsf{down},\mathsf{minimize}(\mathsf{Cost})],\mathsf{Variables}).$$

The file library('clpfd/examples/tsp.pl') contains an example of user-defined variable and value choice heuristics.

3.4. Altri constraints primitivi.

$$\mathsf{element}(?\mathsf{X},+\mathsf{List},?\mathsf{Y})$$

where X and Y are integers or domain variables and List is a list of integers or domain variables. True if the X:th element of List is Y. Operationally, the domains of X and Y are constrained so that for every element in the domain of X, there is a compatible element in the domain of Y, and vice versa.

This constraint uses an optimized algorithm for the special case where List is ground.

element/3 maintains domain-consistency in X and interval-consistency in List and Y.

Inoltre è deterministico:

$$\mathsf{element}(3,[1,1,2,2,3,3,1,1],\mathsf{Y})$$

restituisce $\mathsf{Y}=2$.

$$\mathsf{element}(\mathsf{X},[1,1,2,2,3,3,1,1],1)$$

restituisce $\mathsf{X}\ \mathsf{in}\ (1..2)\vee(7..8)$.

$$\mathsf{element}(\mathsf{X},[1,1,2,2,3,3,1,1],\mathsf{Y})$$

restituisce $\mathsf{X}\ \mathsf{in}\ 1..8,\mathsf{Y}\ \mathsf{in}\ 1..3$, tenendosi implicitamente i vari constraints coinvolti.

$$\mathsf{all_different}(+\mathsf{Variables})$$

where Variables is a list of domain variables or integers. Each variable is constrained to take a value that is unique among the variables. Declaratively, this is equivalent to an inequality constraint for each pair of variables. (consultare anche le opzioni sul manuale).

serialized/3 can model a set of tasks to be serialized with sequence-dependent setup times. For example, the following constraint models three tasks, all with duration 5, where task 1 must precede task 2 and task 3 must either complete before task 2 or start at least 10 time units after task 2 started:

```
?- domain([S1,S2,S3], 0, 20),
   serialized([S1,S2,S3], [5,5,5], [precedences([d(2,1,sup),d(2,3,10)])]).

S1 in 0..15,
S2 in 5..20,
S3 in 0..20 ?
```

The bounds of S1 and S2 changed because of the precedence constraint. Setting S2 to 5 will propagate S1=0 and S3 in 15..20.

The following constraint can be thought of as constraining n tasks, each with a start time Sj, a duration Dj, and a resource amount Rj, so that the total resource consumption does not exceed Limit at any time:

```
cumulative(+Starts,+Durations,+Resources,?Limit)
cumulative(+Starts,+Durations,+Resources,?Limit,+Options)
```

where Starts = [S1,...,Sn], Durations = [D1,...,Dn], Resource = [R1,...,Rn] are lists of domain variables with finite bounds or integers, and Limit is a domain variable with finite bounds or an integer. Durations, Resources and Limit must be non-negative. Let:

```
a = min(S1,...,Sn),
b = max(S1+D1,...,Sn+Dn)
Rij = Rj, if Sj =< i < Sj+Dj
Rij = 0 otherwise
```

The constraint holds if:

```
Ri1+...+Rin =< Limit, for all a =< i < b
```

If given, Options should be of the same form as in serialized/3, except the resource(R) option is not useful in cumulative/5.

3.5. Knapsack. Si consideri il problema di *Knapsack del contrabbandiere*. Lo zaino ha dimensione 9, il whisky occupa 4 e vale 15, il profumo occupa 3 e vale 10, le sigarette oocupano 2 e valgono 7. Si ricerca un'allocazione dello zaino che assicuri un profitto di almeno 30. Al goal

$$[W, P, S] :: 0..9,$$
$$4 * W + 3 * P + 2 * S\# <= 9,$$
$$15 * W + 10 * P + 7 * S\# >= 30.$$

Viene risposto:

$$W \text{ in } 0..2, P \text{ in } 0..3, S \text{ in } 0..4$$

Che sta a significare che esiste una soluzione all'interno di quei vincoli. Se volessimo averne una, dovremmo dare il goal:

$$[\mathsf{W},\mathsf{P},\mathsf{S}] :: 0..9,$$
$$4 * \mathsf{W} + 3 * \mathsf{P} + 2 * \mathsf{S}\# < 9,$$
$$15 * \mathsf{W} + 10 * \mathsf{P} + 7 * \mathsf{S}\# >= 30,$$
$$\mathsf{labeling}([],[\mathsf{W},\mathsf{P},\mathsf{S}]).$$

Viene restituita la soluzione ammissibile:

$$\mathsf{P} = 3, \mathsf{S} = 0, \mathsf{W} = 0$$

In $CLP(\mathcal{F}D)$ vi è però anche la possibilità di massimizzare una soluzione soddisfacente i vincoli inseriti.

$$\mathsf{L}\# = 15 * \mathsf{W} + 10 * \mathsf{P} + 7 * \mathsf{S},$$
$$\mathsf{maximize}((([\mathsf{W},\mathsf{P},\mathsf{S}] :: 0..9, 4 * \mathsf{W} + 3 * \mathsf{P} + 2 * \mathsf{S}\# =< 9, \mathsf{labeling}([],[\mathsf{W},\mathsf{P},\mathsf{S}])), \mathsf{L}).$$

restituisce $\mathsf{L} = 32, \mathsf{P} = 1, \mathsf{S} = 1, \mathsf{W} = 1$.

4. Metodologia Constrain & generate

La metodologia *Constrain & generate* è una metodologia di programmazione dichiarativa, alternativa alla generate & test che permette di rimandare il più possibile la fase di istanziazione delle variabili. I seguenti esempi illustreranno la tecnica.

5. N-Regine

```
queens( N , Queens) :-
  length(Queens, N),
  Queens :: 1 .. N,
  constrain( Queens ),
  labeling( [ff] , Queens ).

%%%

constrain( Queens ) :-
    all_different( Queens ),  % Spiega
    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.
```

ESERCIZIO 8.11. Si testi il programma delle regine, cercando di trovare il massimo N per cui il risolutore CLP riesce a risolverlo. Si provi poi con l'opzione [ff] (first fail) nella labeling.

6. Map Coloring

```
clpcoloring( Mapname , N , Edges ) :-
  map( Mapname , Nodes , Edges ),
  Nodes :: 1 .. N,
  constrain( Edges ),
  labeling( [] , Nodes ).

%%%

constrain( [[A,B]|R] ) :-
    A  #\= B,
    constrain( R ).
constrain( []  ).
```

ESERCIZIO 8.12. Si generi automaticamente una cartina di dimensioni ragguardevoli (si veda sotto un esempio) e si confrontino computazionalmente le soluzioni Prolog e CLP allo stesso problema.

```
map( march9 , Nodes , Edges ) :-
     Levels = 10000,
     N is 3 * Levels,
     length( Nodes, N ),
     add_edges( Levels, Nodes, Edges1 ).
add_edges( 1 , [A,B,C] , [ [A,B], [B,C] ] ).
add_edges( 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,
  add_edges( M , [A1,B1,C1|R]  , E ).
```

7. Marriage Problem

Si consideri la situazione tradizionale (rappresentata in Prolog):

$$\begin{array}{l}\mathsf{likes(andrea, diana).}\\ \mathsf{likes(andrea, federica).}\\ \mathsf{likes(bruno, diana).}\\ \mathsf{likes(bruno, elena).}\\ \mathsf{likes(bruno, federica).}\\ \mathsf{likes(carlo, elena).}\\ \mathsf{likes(carlo, federica).}\end{array}$$

Si vuole trovare un accoppiamento ammissibile. La soluzione in Prolog è:

$$\mathsf{likes(andrea, A), likes(bruno, B), likes(carlo, C), A \backslash = B, A \backslash = C, B \backslash = C}$$

La seguente è la formulazione in $CLP(\mathcal{F}D)$ (senza far uso di fatti):

$$\mathsf{Sposi = [Andrea, Bruno, Carlo], Sposi :: 1..3, all_different(Sposi), Andrea\#\backslash = 2, Carlo\#\backslash = 1.}$$

La risposta fornita è

$$\mathsf{Sposi = [Andrea, Bruno, Carlo], Andrea\ in\ 1 \vee 3, Bruno\ in\ 1..3, Carlo\ in\ 2..3}$$

A questo punto basta aggiungere un labeling, per ottenere:

$$\mathsf{Carlo = 3, Bruno = 2, Sposi = [1, 2, 3], Andrea = 1}$$

8. SEND + MORE = MONEY

As a simple CSP example, let us consider the Send More Money puzzle. In this problem, the variables are the letters S, E, N, D, M, O, R, and Y. Each letter represents a digit between 0 and 9. The problem is to assign a value to each digit, such that SEND + MORE equals MONEY.

A program which solves the puzzle is given below. The program contains the typical three steps of a clp(FD) program:

(1) declare the domains of the variables
(2) post the problem constraints
(3) look for a feasible solution via backtrack search, or look for an optimal solution via branch-and-bound search

Sometimes, an extra step precedes the search for a solution: the posting of surrogate constraints to break symmetries or to otherwise help prune the search space. No surrogate constraints are used in this example.

The domains of this puzzle are stated via the domain/3 goal and by requiring that S and M be greater than zero. The two problem constraint of this puzzle are the equation (sum/8) and the constraint that all letters take distinct values (all_different/1). Finally, the backtrack search is performed by labeling/2. Different search strategies can be encoded in the Type parameter. In the example query, the default search strategy is used (select the leftmost variable, try values in ascending order).

```
:- use_module(library(clpfd)).

mm( [S,E,N,D,M,O,R,Y] ) :-
     domain([S,E,N,D,M,O,R,Y], 0, 9),      % step 1
     S#>0, M#>0,
     all_different([S,E,N,D,M,O,R,Y]),     % step 2
     sum(S,E,N,D,M,O,R,Y),
     labeling(Type, [S,E,N,D,M,O,R,Y]).    % step 3

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.

:- mm( [S,E,N,D,M,O,R,Y] ).

D = 7,
E = 5,
M = 1,
N = 6,
O = 0,
R = 8,
S = 9,
Y = 2
```

9. Uso del cumulative

E' il costrutto di base per lo scheduling. Un tipico esempio: ci sono 4 lavori che utilizzano la stessa risorsa. Il primo lavoro dura 3 e usa 2 unità di risorsa. Il secondo dura 2 e usa 3 unità, il terzo dura 1 e ne dura 2. Il quarto dura 2 e usa 3 unità. Si vuole scoprire una allocazione dei lavori.

$$[\mathsf{S1},\mathsf{S2},\mathsf{S3},\mathsf{S4}] :: 1..20,$$
$$\mathsf{cumulative}([\mathsf{S1},\mathsf{S2},\mathsf{S3},\mathsf{S4}],[3,2,1,2],[2,3,2,3],5),$$
$$\mathsf{labeling}([],[\mathsf{S1},\mathsf{S2},\mathsf{S3},\mathsf{S4}]).$$

Otteniamo $\mathsf{S1}=1,\mathsf{S2}=1,\mathsf{S3}=3,\mathsf{S4}=4$.

Un secondo esempio può essere dato dal seguente problema di allocazione per una variante della battaglia navale.

Dimensioni delle navi:

- Portaerei: larga 2, lunga 4.
- Corazzata: larga 1, lunga 4.
- Incrociatore: largo 1, lungo 3.
- Cacciamine: largo 1, lungo 1.

Si desiderano sistemare una flotta che consta di una portaerei, due corazzate, 3 incrociatori e 2 cacciamine in modo tale da attraversare un canale largo 3.

$$\mathsf{domain}([\mathsf{P1},\mathsf{C1},\mathsf{C2},\mathsf{I1},\mathsf{I2},\mathsf{I3},\mathsf{D1},\mathsf{D2}],1,20),$$
$$\mathsf{cumulative}([\mathsf{P1},\mathsf{C1},\mathsf{C2},\mathsf{I1},\mathsf{I2},\mathsf{I3},\mathsf{D1},\mathsf{D2}],[4,4,4,3,3,3,1,1],[2,1,1,1,1,1,1,1],3),$$
$$\mathsf{labeling}([],[\mathsf{P1},\mathsf{C1},\mathsf{C2},\mathsf{I1},\mathsf{I2},\mathsf{I3},\mathsf{D1},\mathsf{D2}]).$$

Si ottiene:

$$\mathsf{C1}=1,\mathsf{C2}=5,\mathsf{D1}=8,\mathsf{D2}=9,\mathsf{I1}=5,\mathsf{I2}=5,\mathsf{I3}=8,\mathsf{P1}=1$$

Tale schieramento può essere ulteriormente compresso. Aggiungiamo i vincoli:

$$\mathsf{P1}+4\#<\mathsf{A},\mathsf{C1}+4\#<\mathsf{A},\mathsf{C2}+4\#<\mathsf{A},\mathsf{I1}+3\#<\mathsf{A},\mathsf{I2}+3\#<\mathsf{A},$$
$$\mathsf{I3}+3\#<\mathsf{A},\mathsf{D1}+1\#<\mathsf{A},\mathsf{D2}+1\#<\mathsf{A},\mathsf{A}=11$$

Si ottiene la nuova soluzione:

$$\mathsf{A}=11,\mathsf{C1}=5,\mathsf{C2}=5,\mathsf{D1}=9,\mathsf{D2}=9,\mathsf{I1}=1,\mathsf{I2}=4,\mathsf{I3}=7,\mathsf{P1}=1$$

10. Allocazione di compiti

In un'officina artigianale ci sono 4 dipendenti $w1, w2, w3, w4$ e sono fabbricati 4 prodotti. La seguente matrice indica il profitto associato alla 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

Cerchiamo di massimizzare il profitto totale.

Forniamo due soluzioni; la prima esplicita la matrice:

```
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).
```

Per evitare il goal così lungo si suggerisce di definire dei predicati ausiliari sommarighe, sommacolonne eccetera...

Seconda soluzione: con element.

```
[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 8.13. Usare l'opzione assumptions(K) del labeling per verificare quale dei due approcci giunge con meno tentativi alla soluzione. Analizzare la risposta.

11. Constraints reificati

Supponiamo di avere un certo numero di constraints, diciamo 5 e che si desideri che almeno 3 di questi siano verificati nella soluzione.

Un modo per farlo è il seguente:

```
c1 #<=> B1,
c2 #<=> B2,
c3 #<=> B3,
c4 #<=> B4,
c5 #<=> B5,
B1 + B2 + B3 + B4 + B5 #>= 3
```

I vari *Bi* sono visti come constraints booleani. Questo modo di operare viene detto *reificazione* dei constraints.

Esempio:

```
| ?- 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
```

Bibliography

[1] K. R. Apt. *From Logic Programming to Prolog*. International Series in Computer Science. Prentice Hall, 1997.

[2] 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.

[3] J. W. Lloyd. *Foundations of Logic Programming*. Springer-Verlag, 2nd edition, 1987.

[4] Kim Marriott and Peter Stuckey. *Programming with Constraints*. The MIT Press, 1998.

[5] A. Martelli and U. Montanari. An efficient unification algorithm. *ACM Transactions on Programming Languages and Systems*, 4:258–282, 1982.

[6] M. S. Paterson and M. N. Wegman. Linear unification. *Journal of Computer System Science*, 16(2):158–167, 1978.

[7] Leon Sterling and Ehud Shapiro. *The Art of Prolog*. The MIT Press, 2nd edition, 1997.

[8] David H. D. Warren. Logic programming and compiler writing. volume 10, pages 97–125, 1980.