dispense dell'insegnamento di

Programmazione Logica e Funzionale

Marco Bernardo Alessandro Aldini Luca Padovani

Università degli Studi di Urbino Carlo Bo
Dipartimento di Scienze Pure e Applicate
Scuola di Scienze, Tecnologie e Filosofia dell'Informazione
Corso di Laurea in Informatica Applicata (classe L-31)

versione 04/04/2024

Queste dispense sono state preparate con LaTeX e sono reperibili sulla piattaforma Moodle `blended.uniurb.it`
Esse costituiscono soltanto un ausilio per il docente, quindi non sono in nessun modo sostitutive dei testi consigliati.

Si richiede di portare le dispense alle esercitazioni in laboratorio, ma non alle lezioni teoriche in aula.
Se non si capisce un argomento, fare domande a lezione o sul forum di Moodle oppure usufruire del ricevimento.

In aula e in laboratorio, seguire le attività in silenzio per non disturbare e alzare la mano per chiedere la parola.
Quando si arriva tardi a lezione o si prevede di andare via in anticipo, sedersi nei posti vicini all'uscita.

Si consiglia di studiare durante tutto il periodo didattico, evitando di ridursi agli ultimi giorni prima dell'esame.
In ogni caso, è opportuno approfondire sui testi consigliati prima di svolgere il progetto d'esame, non dopo.

Indice

1 Introduzione a Paradigmi e Linguaggi di Programmazione **1**
- 1.1 Definizioni di Base per Paradigmi e Linguaggi . . . 1
- 1.2 Programmazione Imperativa: Procedurale e a Oggetti . . . 2
- 1.3 Programmazione Dichiarativa: Funzionale e Logica . . . 4
- 1.4 Linguaggi di Programmazione Sequenziali e Concorrenti . . . 6
- 1.5 Linguaggi di Script, Interrogazione, Markup, Modellazione . . . 6

2 Richiami di Matematica Discreta **7**
- 2.1 Elementi di Teoria degli Insiemi . . . 7
- 2.2 Relazioni, Funzioni, Operazioni . . . 9
- 2.3 Principio di Induzione . . . 11

3 Lambda Calcolo **15**
- 3.1 Sintassi del Lambda Calcolo . . . 15
- 3.2 Semantica del Lambda Calcolo e Logica Combinatoria . . . 17
- 3.3 Ricorsione via Punti Fissi e Calcolabilità . . . 19
- 3.4 Terminazione e Confluenza nel Lambda Calcolo . . . 22
- 3.5 Lambda Calcolo con Tipi . . . 23

4 Programmazione Funzionale: Il Linguaggio Haskell **27**
- 4.1 Dal Lambda Calcolo alla Programmazione Funzionale . . . 27
- 4.2 Haskell: Assemblaggio di Caratteristiche Funzionali . . . 28
- 4.3 Haskell: Espressioni, Tipi di Dati, Classi di Tipi . . . 31
- 4.4 Haskell: Funzioni, Guardie, Pattern Matching . . . 34
- 4.5 Haskell: Funzioni Polimorfe, di Ordine Superiore, Anonime . . . 38
- 4.6 Haskell: Valutazione Pigra, Input/Output, Moduli . . . 40

5 Logica Proposizionale **45**
- 5.1 Sintassi della Logica Proposizionale . . . 45
- 5.2 Semantica e Intrattabilità della Logica Proposizionale . . . 47
- 5.3 Conseguenza ed Equivalenza nella Logica Proposizionale . . . 51
- 5.4 Proprietà Algebriche dei Connettivi Logici . . . 53
- 5.5 Sistemi Deduttivi per la Logica Proposizionale . . . 55

6 Logica dei Predicati **59**
- 6.1 Sintassi della Logica dei Predicati . . . 59
- 6.2 Semantica e Indecidibilità della Logica dei Predicati . . . 63
- 6.3 Conseguenza ed Equivalenza nella Logica dei Predicati . . . 65
- 6.4 Proprietà Algebriche dei Quantificatori . . . 68
- 6.5 Sistemi Deduttivi per la Logica dei Predicati . . . 69

7 Refutazione di Formule Logiche **71**
7.1 Forme Normali per la Logica Proposizionale e dei Predicati 71
7.2 Unificazione di Formule di Logica dei Predicati 74
7.3 Teoria di Herbrand e Algoritmo di Refutazione 77
7.4 Risoluzione di Robinson e Algoritmo di Refutazione 81

8 Programmazione Logica: Il Linguaggio Prolog **87**
8.1 Dalla Logica alla Programmazione Logica 87
8.2 Prolog: Clausole di Horn e Strategia di Risoluzione SLD 90
8.3 Prolog: Sintassi dei Termini e Predicati Predefiniti 93
8.4 Prolog: Negazione, Taglio, Input/Output, Predicati Avanzati 98

9 Attività di Laboratorio in Linux **103**
9.1 Il Compilatore/Interprete `ghci` 103
9.2 Implementazione e Modifica di Programmi Haskell 104
9.3 Il Compilatore/Interprete `gprolog` 104
9.4 Implementazione e Modifica di Programmi Prolog 105

Capitolo 1

Introduzione a Paradigmi e Linguaggi di Programmazione

1.1 Definizioni di Base per Paradigmi e Linguaggi

- Un linguaggio di programmazione è uno strumento linguistico attraverso il quale è possibile formalizzare un algoritmo in modo tale che esso sia eseguibile da un computer in quanto macchina programmabile. Il risultato di questa formalizzazione prende il nome di programma informatico o applicazione informatica ed è costituito da una collezione di istruzioni memorizzate in un file detto file sorgente.

- Ogni linguaggio di programmazione ha le proprie regole lessicali e grammaticali. Le prime specificano come combinare i simboli per ottenere le parole o lessemi del linguaggio, mentre le seconde descrivono come combinare le parole per formare prima i sintagmi e poi le frasi sintatticamente ben formate. I linguaggi di programmazione hanno lessico e sintassi molto più semplici di quelli dei linguaggi naturali e sono sostanzialmente classificabili come linguaggi liberi dal contesto nella gerarchia di Chomsky.

- Ogni linguaggio di programmazione è inoltre dotato di una semantica, che attribuisce un significato a ogni costrutto linguistico, e di conseguenza a ogni programma, in modo tale da stabilirne l'effetto a tempo d'esecuzione. Non tutte le frasi sintatticamente ben formate del linguaggio hanno un significato.

- Esempio: la frase "il gatto guida una motocicletta" è costituita dai lessemi "il", "gatto", "guida", "una", "motocicletta" e dai sintagmi "il gatto" (soggetto), "guida" (predicato), "una motocicletta" (complemento), quindi è sintatticamente ben formata in linguaggio naturale, ma non ha senso nella realtà.

- Ciascun computer è dotato di due linguaggi di programmazione che sono specifici del proprio processore: il linguaggio macchina e il linguaggio assemblativo. Ogni istruzione di questi linguaggi è costituita dal codice operativo dell'istruzione (addizione, sottrazione, moltiplicazione, divisione, trasferimento dati da/verso memoria o salto condizionato/incondizionato di istruzioni) e dai valori o dagli indirizzi di memoria degli operandi.

- Nel linguaggio macchina codici e indirizzi sono espressi in binario, mentre nel linguaggio assemblativo possono essere espressi in formato mnemonico tramite identificatori. Di conseguenza le istruzioni in linguaggio macchina sono direttamente eseguibili dal computer, mentre quelle in linguaggio assemblativo – che sono più intelligibili – necessitano di essere preventivamente trasformate nelle corrispondenti istruzioni in linguaggio macchina tramite un traduttore detto assemblatore.

- I programmi scritti in linguaggio macchina o assemblativo non sono portabili, cioè non possono essere eseguiti su computer aventi un processore diverso. Ancor peggio, questi linguaggi non consentono di scrivere programmi agevolmente perché si collocano a un livello di astrazione troppo basso per la mente umana.

- Dalla metà degli anni 1950 sono stati sviluppati linguaggi di programmazione di alto livello di astrazione in cui è più facile scrivere programmi perché, rispetto ai linguaggi macchina e assemblativi, essi sono più vicini al linguaggio naturale delle persone e alla notazione matematica di uso comune. Un'importante conseguenza è la naturale portabilità dei programmi su computer diversi, conducendo pertanto a compimento una completa separazione tra hardware e software iniziata con l'idea di computer a programma memorizzato di Von Neumann e teorizzata ancor prima da Turing.
- Un programma scritto in un linguaggio di alto livello di astrazione non è però direttamente eseguibile da un computer, in quanto necessita di essere preventivamente tradotto in un programma equivalente scritto nel linguaggio macchina o assemblativo di quel computer. Dato il diverso livello di astrazione, ogni istruzione del programma originario verrà tradotta in una sequenza di più istruzioni macchina o assemblative. Il traduttore è detto compilatore o interprete a seconda che la traduzione venga effettuata una volta per tutte prima dell'esecuzione del programma (nel qual caso il risultato della traduzione viene memorizzato in un file detto file eseguibile) o contestualmente all'esecuzione istruzione per istruzione.
- I numerosi linguaggi di programmazione di alto livello di astrazione che sono stati ideati possono essere molto diversi tra loro, talché hanno dato luogo a più paradigmi di programmazione. Un paradigma di programmazione è un insieme di concetti, costrutti e astrazioni che determinano il processo di stesura di un programma in certi linguaggi. Ad un estremo abbiamo i linguaggi di programmazione che si concentrano solo sulla definizione di cosa calcolare senza specificarne il come, esprimendo soltanto la conoscenza relativa al dominio del problema da risolvere. All'estremo opposto abbiamo i linguaggi di programmazione in cui occorre indicare pure tutti i dettagli su come calcolare la soluzione del problema.
- Esempio: il fattoriale di $n \geq 1$ è n moltiplicato per il fattoriale di $n-1$ con fattoriale di 1 uguale a 1 vs. per calcolare il fattoriale di n bisogna moltiplicare 1 per 2 poi il risultato va moltiplicato per 3 poi ancora il risultato va moltiplicato per 4 e così via fino ad arrivare alla moltiplicazione per n.

1.2 Programmazione Imperativa: Procedurale e a Oggetti

- Il paradigma imperativo trae la sua origine dai linguaggi macchina e assemblativi e ricomprende sia la programmazione procedurale che la programmazione a oggetti.
- Nel paradigma imperativo i programmi sono sequenze di istruzioni che prescrivono dettagliatamente quali operazioni effettuare e in quale ordine. L'istruzione fondamentale è l'istruzione di assegnamento, la quale stabilisce come modificare il contenuto di una locazione di memoria individuata da un identificatore di variabile. Per il controllo del flusso di esecuzione sono disponibili istruzioni di selezione e di ripetizione che, grazie al teorema fondamentale della programmazione strutturata di Böhm e Jacopini, evitano il ricorso a istruzioni di salto condizionato/incondizionato. È altresì supportata la ricorsione. Sono presenti tipi di dati scalari per valori numerici, logici, enumerati, caratteri e indirizzi, tipi di dati strutturati omogenei quali array, stringhe e insiemi ed eterogenei quali record e file, nonché tipi di dati definiti dal programmatore. Le rispettive variabili sono allocabili staticamente o dinamicamente.
- Nella programmazione procedurale le istruzioni di un programma sono raggruppate in blocchi detti procedure per evitare ridondanze di codice. Ogni procedura è definita una sola volta nel programma e può essere poi invocata da più istruzioni del programma, presenti in altre procedure o nella procedura stessa. La definizione della procedura è basata su dati di valore ignoto chiamati parametri formali che vengono inizializzati al momento dell'invocazione della procedura attraverso i corrispondenti parametri effettivi, i quali possono essere passati per valore o per indirizzo. La procedura invocata può restituire un risultato da utilizzare nell'istruzione contenente l'invocazione.
- Esempi di linguaggi di programmazione procedurali:
  - Fortran: 1957, John Backus presso IBM, applicazioni scientifiche, inizialmente privo di articolazione in sottoprogrammi e supporto a ricorsione e allocazione dinamica.
  - Cobol: 1960, comitato industriale e governativo americano tra cui Jean Sammet, applicazioni gestionali, sintassi molto verbosa, limitazioni iniziali simili a quelle del Fortran.

  - Algol: 1958, comitato accademico europeo e americano, descrizione chiara degli algoritmi, sintassi in formato Backus-Naur, blocchi di istruzioni delimitati da inizio e fine, privo di istruzioni di input/output predefinite.
  - PL/I: 1966, IBM, misto di Fortran e Cobol influenzato da Algol da usare come unico linguaggio per applicazioni scientifiche e gestionali oltre che sistemistiche su IBM 360 quale unica architettura che soppiantava tutte le precedenti, privo di parole riservate.
  - Basic: 1964, Kemeny e Kurtz presso Dartmouth College, programmazione per principianti, interpretato anziché compilato, inizialmente privo di supporto per la programmazione strutturata (famosa critica di Dijkstra), evolutosi in Visual Basic nel 1991 presso Microsoft.
  - Pascal: 1970, Niklaus Wirth presso ETH Zurigo, insegnamento della programmazione strutturata, evoluzione di Algol in cui i blocchi vengono specializzati in procedure e funzioni eventualmente annidate, tipi ricorsivi e dinamici (per liste, alberi, grafi) aggiunti a tipi scalari e array.
  - C: 1972, Dennis Ritchie presso Bell Labs, evoluzione di B – derivato da Ken Thompson come semplificazione di BCPL, progettato nel 1967 da Martin Richards presso l'Università di Cambridge – messa a punto per consentire agli sviluppatori del sistema operativo Unix di implementare funzionalità, no annidamento di funzioni, portabilità coniugata con vicinanza alla macchina.
  - Modula: 1975, Niklaus Wirth presso ETH Zurigo, evoluzione di Pascal basata su moduli come unità di programmazione compilabili separatamente che sono comprensive di dati e procedure – estensione di unità Pascal e librerie C – e su processi e segnali per la programmazione concorrente.
  - Ada: 1980, Jean David Ichbiah presso Honeywell Bull, safety-critical embedded software tipico dell'industria aerospaziale, sistema di tipi molto evoluto, supporto a modularità basato su package e a programmazione concorrente basato su task e scambio sincrono di messaggi.

- La programmazione a oggetti è un'evoluzione della programmazione procedurale finalizzata ad aumentare l'efficienza del processo di produzione e manutenzione del software, attraverso un innalzamento del livello di astrazione ottenuto separando i dati dalle procedure pur collocandoli vicino a queste ultime.
- Nella programmazione a oggetti le istruzioni vengono raggruppate assieme ai loro dati, dando luogo alle cosiddette classi. Una classe è una collezione di attributi (dati) e metodi (procedure), collettivamente chiamati membri, la cui visibilità può essere pubblica, protetta o privata a seconda che siano accessibili da tutte le altre classi, dalla classe e relative estensioni oppure solo dalla classe stessa. Ispirandosi al concetto di tipo di dato astratto, che comprende sia un insieme di valori che le operazioni su di esso, ciascuna classe esibisce un'interfaccia costituita dalla dichiarazione dei soli membri pubblici e incapsula la dichiarazione di tutti gli attributi e la definizione di tutti i metodi. Grazie al meccanismo dell'ereditarietà è possibile definire gerarchie di classi, promuovendo il riutilizzo del codice attraverso l'estensione di classi già esistenti, nonché supportare il polimorfismo, mediante classi astratte che lasciano alle classi derivate il compito di completare la definizione dei suoi metodi.
- Un programma a oggetti è una collezione di oggetti che interagiscono tra loro invocando reciprocamente i relativi metodi per consultare o modificare i rispettivi dati e sono dotati di meccanismi per gestire eccezioni quando si verificano determinate situazioni. Un oggetto è un'istanza di una classe avente valori specifici ed eventualmente immodificabili per gli attributi della classe, quindi è assimilabile all'allocazione dinamica con relativa inizializzazione di una variabile avente come tipo quella classe. Gli oggetti di una classe hanno gli stessi metodi, ma possono differire per i valori degli attributi.
- Esempi di linguaggi di programmazione a oggetti:
  - Simula: 1965, Dahl e Nygaard presso Norwegian Computing Center, programmi di simulazione al computer, coroutine come variante di subroutine (cioè procedura) la cui esecuzione può essere sospesa e poi ripresa con istruzioni apposite, classi e sottoclassi di oggetti.
  - Smalltalk: 1972, Alan Kay presso Learning Research Group di Xerox, apprendimento costruzionista, programmi compatti, tutti i valori sono oggetti (anche numeri/booleani/caratteri), scambio di messaggi (anche per il controllo del flusso), gestione di eccezioni, compilato in bytecode che viene poi interpretato da una macchina virtuale, software design pattern, interfacce grafiche, ambiente di sviluppo integrato, riflessione come capacità di un programma di ispezionare e modificare struttura e comportamento a tempo d'esecuzione.

  - C++: 1985, Bjarne Stroustrup presso Bell Labs, evoluzione di C orientata agli oggetti, template per consentire a funzioni e classi di operare su tipi generici specificati come parametri, ammesso il riuso di operatori esistenti in nuovi tipi di dati definiti dal programmatore, ereditarietà anche da molteplici classi base, distruttori di oggetti, try-catch-throw per la gestione di eccezioni, spazi di nomi per evitare conflitti tra nomi, poi esteso con thread per la programmazione concorrente.
  - Eiffel: 1986, Bertrand Meyer, affidabilità del software, design by contract in cui asserzioni come precondizioni e postcondizioni oppure invarianti di classe o di ciclo assicurano la correttezza di un programma, gestione delle eccezioni basata su tali asserzioni, allocazione dinamica della memoria comprensiva di garbage collection automatica.
  - Java: 1996, James Gosling presso Sun Microsystems, verso applicazioni web tramite applet e servlet, compilato in bytecode che viene poi interpretato o compilato a tempo d'esecuzione dalla Java Virtual Machine, edizioni diverse per ambiti di applicazione diversi, classi organizzate in package, tutti i valori sono oggetti eccetto numeri/booleani/caratteri (per motivi prestazionali), non supportati il riuso di operatori e l'ereditarietà multipla (per chiarezza), privo degli operatori di basso livello di C/C++ e del tipo puntatore, diverse forme di garbage collection automatica, supporto alla programmazione concorrente basato su thread.
  - C#: 2000, Anders Hejlsberg presso Microsoft, evoluzione di C++ che ne rafforza il sistema di tipi e viene compilato nel Common Intermediate Language della Common Language Infrastructure, supportato il riuso di operatori ma non l'ereditarietà multipla, poi esteso con costrutti della programmazione funzionale.
- Esistono varianti a oggetti di linguaggi procedurali come Visual Basic per il Basic, Delphi per il Pascal e Objective-C per il C. Fortran, Cobol, Modula e Ada sono stati estesi in modo orientato agli oggetti.

1.3 Programmazione Dichiarativa: Funzionale e Logica

- Nel paradigma dichiarativo i programmi sono collezioni di espressioni matematico-logiche che descrivono cosa deve essere calcolato senza specificare come (anziché istruzioni di assegnamento da eseguire nell'ordine stabilito dalle istruzioni di controllo del flusso) e le loro esecuzioni sono assimilabili a deduzioni in un sistema formale (anziché sequenze di passi che modificano il contenuto della memoria). Questo comporta un notevole cambio di mentalità nell'attività di programmazione, come evidenziato da John Backus nel suo famoso articolo del 1978 intitolato "Can Programming Be Liberated from the Von Neumann Style? A Functional Style and Its Algebra of Programs".
- Principali differenze tra il paradigma dichiarativo e il paradigma imperativo:
  - I programmi dichiarativi sono espressi a un livello concettuale più alto dei programmi imperativi, così come i loro traduttori ed esecutori operano a un livello concettuale più alto rispetto a quelli convenzionali. Questo favorisce la verifica di correttezza dei programmi dichiarativi.
  - La nozione di stato della computazione intesa come il contenuto della memoria non esiste più nella programmazione dichiarativa perché i dati sono immutabili. In particolare, il valore di una variabile non cambia più una volta che è stato attribuito (legame anziché assegnamento).
  - In programmazione dichiarativa non ci sono effetti collaterali in quanto la valutazione di un'espressione non può modificare variabili al di fuori dell'ambiente locale. Questo è invece possibile in programmazione imperativa in presenza di variabili globali o parametri passati per indirizzo.
  - La conseguente trasparenza referenziale fa sì che venga restituito lo stesso risultato ogni volta che un'espressione viene valutata sugli stessi argomenti. Il risultato di una procedura che usa variabili globali o passa parametri per indirizzo potrebbe invece dipendere dal momento in cui è invocata.
  - Dal punto di vista prestazionale, il valore di un'espressione complessa su determinati argomenti può essere memorizzato in modo da non doverlo ricalcolare. Analogamente, due espressioni tra cui non ci sono dipendenze di dati possono essere valutate in qualsiasi ordine e persino in parallelo.
  - La ricorsione è praticamente l'unico strumento linguistico per rappresentare l'iterazione e il tipo di dato ricorsivo lista è praticamente l'unico tipo di dato strutturato disponibile. La gestione dinamica della memoria per le liste è completamente automatizzata (allocazioni e disallocazioni sono implicite e quindi non devono essere programmate come nel caso dei puntatori).

- La programmazione dichiarativa nasce in intelligenza artificiale per supportare il ragionamento automatico, la rappresentazione della conoscenza e l'elaborazione del linguaggio naturale. I programmi dichiarativi agevolano l'interazione con l'utente e il debugging da parte del programmatore perché la loro esecuzione consiste nell'applicazione di un ciclo leggi-valuta-stampa alle espressioni matematico-logiche presenti nei programmi. I linguaggi dichiarativi sono dunque interpretati, ma per motivi di efficienza vengono solitamente accompagnati anche da compilatori da utilizzare quando il codice diventa stabile.
- Il primo filone ad affermarsi è stato quello della programmazione funzionale. Esso si basa sul λ-calcolo sviluppato da Alonzo Church negli anni 1930, un sistema formale dello stesso potere computazionale della macchina di Turing universale per l'astrazione e l'applicazione di funzioni matematiche calcolabili. Un programma funzionale è pertanto una collezione di definizioni di funzioni e relative applicazioni e composizioni, che sfrutta la naturale trasparenza referenziale delle funzioni matematiche.
- Esempi di linguaggi funzionali (molti successivamente estesi con costrutti procedurali o a oggetti):
  - Lisp: 1958, John McCarthy presso MIT, lista come unica struttura dati i cui elementi sono racchiusi tra tonde e separati da spazi, funzioni espresse come liste in cui il primo elemento è il nome della funzione e i successivi sono gli argomenti, ricorsione e funzioni di ordine superiore, metaprogrammazione come capacità di trattare programmi come dati nonché di generare e modificare programmi, gestione dinamica della memoria comprensiva di garbage collection automatica, moltissimi dialetti.
  - Scheme: 1973, Steele e Sussman presso MIT, dialetto minimalista di Lisp, chiamate ricorsive ammesse solo in coda alla definizione di una funzione (maggiore efficienza perché non è necessario allocare un nuovo record sullo stack), strutturazione a blocchi per dichiarazioni di variabili, visibilità di una variabile statica solo all'interno della funzione in cui è dichiarata come in Algol e nei suoi successori anziché dinamica anche nelle funzioni invocate da quella funzione come in Lisp (maggiori chiarezza ed efficienza perché non è necessario scendere nello stack).
  - ML: 1973, Robin Milner presso Università di Edimburgo, linguaggio per il dimostratore automatico e interattivo di teoremi LCF, sintassi più simile a quella della matematica rispetto a Lisp, visibilità statica delle variabili, sistema di tipi comprensivo di inferenza automatica del tipo delle espressioni, pattern matching per gli argomenti delle funzioni, strutturazione a blocchi per la gestione delle eccezioni, moduli dotati di interfacce che dichiarano tipi e funzioni messi a disposizione (dialetti: SML presso Università di Edimburgo, Caml presso INRIA, F# presso Microsoft).
  - Haskell: 1990, comitato accademico europeo e americano, valutazione dei parametri effettuata solo nel momento in cui diventa necessaria (come in Miranda, progettato nel 1985 da David Turner in Inghilterra), visibilità statica delle variabili, sistema di tipi comprensivo di inferenza automatica del tipo delle espressioni, classi di tipi e polimorfismo, pattern matching per gli argomenti delle funzioni, principale implementazione Glasgow Haskell Compiler di Peyton-Jones e Marlow.
- Il secondo filone ad affermarsi è stato quello della programmazione logica. Esso si basa sulla logica dei predicati limitandosi a formule espresse come clausole di Horn per motivi di calcolabilità. Tali formule consentono di formalizzare sia fatti che rappresentano la conoscenza su un certo dominio, sia regole per ricavare ulteriore conoscenza applicando a tempo d'esecuzione il sistema di deduzione basato sulla risoluzione di Robinson guidata dalla strategia SLD di Kowalski. Un programma logico è pertanto una collezione di definizioni di predicati, considerati come relazioni matematiche che godono naturalmente di trasparenza referenziale, e relative applicazioni e composizioni che danno luogo a fatti e regole.
- Il principale linguaggio logico è il Prolog. Esso venne sviluppato nel 1972 da Colmerauer e Roussel presso l'Università di Marsiglia per l'elaborazione del linguaggio naturale, in collaborazione presso l'Università di Edimburgo con Kowalski, che fornì tra l'altro l'interpretazione procedurale delle clausole di Horn, e Warren, che poi implementò il primo compilatore. Le clausole di Horn che costituiscono fatti e regole in un programma Prolog seguono la sintassi dei predicati estesa col tipo di dato strutturato lista. La loro esecuzione è attivata tramite un'interrogazione espressa essa pure come clausola di Horn. Il risultato è un valore di verità accompagnato dai legami creati per le variabili eventualmente presenti nell'interrogazione. Prolog supporta sia la metaprogrammazione che il pattern matching per gli argomenti dei predicati e ha lo stesso potere computazionale della macchina di Turing universale.

1.4 Linguaggi di Programmazione Sequenziali e Concorrenti

- Nel paradigma imperativo un programma sequenziale è un programma in cui viene eseguita una sola istruzione alla volta sulla base del controllo del flusso stabilito dal programma stesso. Intendendo per stato della computazione il contenuto della memoria a un certo punto dell'esecuzione, il comportamento di un programma sequenziale viene formalizzato mediante una funzione matematica che descrive quale sia lo stato finale della computazione a fronte dello stato iniziale della computazione indotto da una generica istanza dei dati di ingresso, ignorando tutti gli stati intermedi della computazione.
- Un programma concorrente è invece composto da varie parti che possono essere eseguite contemporaneamente su un computer multiprocessore oppure su più computer collegati in rete. La sincronizzazione e la comunicazione tra le parti del programma avvengono mediante memoria condivisa nel primo caso e scambio di messaggi nel secondo caso. Il comportamento di un programma concorrente non può essere formalizzato tramite una funzione matematica che descrive l'effetto ingresso/uscita, in quanto il risultato calcolato dal programma può dipendere anche dalle velocità relative con cui vengono eseguite le parti del programma. Un modello più appropriato è un grafo stati-transizioni che riporta tutti i possibili stati finali a fronte di quello iniziale, dove ogni sequenza di transizioni è ottenuta interfogliando le istruzioni man mano eseguite dalle varie parti e mostra tutti gli stati intermedi attraversati.
- Tra i sistemi formali alla base della programmazione concorrente menzioniamo le algebre di processi, linguaggi algebrici aventi lo stesso potere computazionale della macchina di Turing universale. Esse sono dotate di operatori per rappresentare le composizioni sequenziale, alternativa e parallela di processi intesi come comportamenti. La loro semantica è basata su grafi stati-transizioni non deterministici dove la concorrenza è ridotta all'interfogliazione di computazioni locali. Come esempi citiamo CCS (1980, Milner), CSP (1984, Hoare), ACP (1984, Bergstra e Klop) e π-calcolo (1992, Milner). In altri formalismi come le reti di Petri (1962), il modello ad attori (1973, Hewitt) e le strutture di eventi (1980, Winskel), la concorrenza tra computazioni locali è invece rappresentata esplicitamente.
- Esempi di linguaggi concorrenti sono Occam (1983, David May presso Inmos, microprocessori, procedurale, CSP), Erlang (1986, Joe Armstrong presso Ericsson, telefonia, funzionale, CSP rivisitato con scambio di messaggi asincrono), Scala (2004, Martin Odersky presso EPF Losanna, a oggetti e funzionale, compatibilità con Java ma sintassi più compatta, reti di Petri). Anche linguaggi precedenti come C, Modula, Ada, C++, Java, C# e le varianti concorrenti di Pascal, Lisp, ML, Haskell, Prolog supportano in varia misura la programmazione concorrente.

1.5 Linguaggi di Script, Interrogazione, Markup, Modellazione

- Oltre ai linguaggi di programmazione e ai linguaggi visuali come Logo (1967, BBN) e Scratch (2003, MIT) pensati per avvicinare i più giovani, in informatica vengono impiegati pure altri linguaggi.
- I linguaggi di script vengono utilizzati per automatizzare l'esecuzione di compiti in contesti come sistemi operativi, server web e interfacce grafiche. Sono linguaggi interpretati dotati di una sintassi snella poi arricchitasi nel tempo, tra cui citiamo Unix shell (anni 1970), Perl/Raku (1987, Larry Wall, Unisys), Tcl/Tk (1988, John Ousterhout, UC Berkeley), Python (1991, Guido van Rossum, CWI), PHP (1995, Rasmus Lerdorf), JavaScript (1995, Brendan Eich, Netscape), Ruby (1995, Yukihiro Matsumoto).
- I linguaggi di interrogazione vengono utilizzati per consultare banche dati e modificarne i contenuti. Sono linguaggi interpretati dichiarativi il cui esempio più famoso è SQL (1974, Chamberlin e Boyce presso IBM, modello relazionale di Edgar Codd), seguito dai linguaggi per banche dati non relazionali.
- I linguaggi di markup vengono utilizzati per annotare documenti digitali con etichette sintatticamente distinguibili dal contenuto testuale che sortiscono il loro effetto nel momento in cui i documenti vengono visualizzati. Ne sono esempi HTML (1993, Tim Berners-Lee presso CERN, condivisione di documenti ipertestuali tramite Internet, pagine web multimediali interpretate dai browser), XML (1996, Jon Bosak presso W3C, formato comprensibile sia dalle persone che dalle macchine, estensibilità dell'insieme delle etichette), LaTeX(1984, Leslie Lamport presso SRI, preparazione di documenti scientifici, compilato).
- I linguaggi di modellazione vengono utilizzati per la progettazione software basata su modelli e abilitano la generazione e la verifica automatiche del software. Oltre ai linguaggi per la descrizione di architetture software, l'esempio più famoso è UML (1994, Rational Software, modello di sviluppo software a oggetti di Grady Booch, diagrammi del comportamento e della struttura dei sistemi software). ■fltplf_1

Capitolo 2

Richiami di Matematica Discreta

2.1 Elementi di Teoria degli Insiemi

- La teoria degli insiemi svolge un ruolo fondazionale nella matematica moderna, in quanto permette di interpretare all'interno di una singola teoria affermazioni relative a oggetti – quali ad esempio numeri, relazioni e funzioni – tratti da tutte le principali aree della matematica.
- Assumiamo come primitivi (cioè non riconducibili ad altri più semplici) i concetti di elemento, insieme e appartenenza. Scriviamo $a \in A$ per indicare che a è un elemento appartenente all'insieme A e $a \notin A$ per indicare che a non è un elemento appartenente all'insieme A. Denotiamo inoltre con U l'universo del discorso, cioè un insieme a cui appartengono tutti gli elementi di cui si sta parlando.
- Ogni insieme può essere definito in al più due modi:
  - Se l'insieme è formato da un numero finito di elementi, esso può essere definito elencandone gli elementi, i quali saranno separati da virgole e racchiusi tra parentesi graffe: $\{a_1, a_2, \ldots, a_n\}$. L'ordine in cui gli elementi sono elencati non è rilevante (lo sarebbe in una tupla ordinata); inoltre gli elementi sono diversi tra loro (la molteplicità degli elementi conta solo nei multinsiemi).
  - L'insieme può sempre essere definito attraverso una proprietà $\mathcal{Q}$ soddisfatta da tutti e soli gli elementi dell'insieme: $\{u \in U \mid u \text{ soddisfa } \mathcal{Q}\}$. L'insieme è detto essere l'insieme di verità per $\mathcal{Q}$.
- L'insieme vuoto, cioè l'insieme privo di elementi, viene denotato con $\emptyset$.
- Siano A e B due insiemi:
  - A è un sottoinsieme di B, scritto $A \subseteq B$, sse $u \in A$ implica $u \in B$ (altrimenti scriviamo $A \not\subseteq B$).
  - A è uguale a B, scritto $A = B$, sse $A \subseteq B$ e $B \subseteq A$ (altrimenti scriviamo $A \neq B$).
  - A è un sottoinsieme proprio di B, scritto $A \subset B$ o $A \subsetneq B$ o $A \subsetneqq B$, sse $A \subseteq B$ e $A \neq B$.
- Sia A un insieme:
  - La cardinalità di A, denotata con $|A|$ o $card(A)$ o $\#A$, è la numerosità degli elementi di A.
  - L'insieme delle parti di A, denotato con $\mathcal{P}(A)$ o 2^A, è l'insieme di tutti i sottoinsiemi di A.
- Proprietà dell'inclusione e dell'uguaglianza insiemistiche rispetto alla cardinalità:
  - Se $A \subseteq B$, allora $|A| \leq |B|$.
  - Se $A \subset B$ e $|A| \in \mathbb{N}$, allora $|A| < |B|$.
  - Se $A = B$, allora $|A| = |B|$.
  - $|A| < |\mathcal{P}(A)|$.
  - Se $|A| = n \in \mathbb{N}$, allora $|\mathcal{P}(A)| = 2^n$.

- Dato un insieme A, si definiscono le seguenti operazioni unarie su di esso:
  - Complementazione: $\overline{A} = A^{\mathrm{c}} = \mathcal{C}(A) = \complement(A) = \{u \in U \mid u \notin A\}$.
- Dati due insiemi A e B, si definiscono le seguenti operazioni binarie su di essi:
  - Unione: $A \cup B = \{u \in U \mid u \in A \text{ o } u \in B\}$.
  - Intersezione: $A \cap B = \{u \in U \mid u \in A \text{ e } u \in B\}$.
  - Differenza: $A \setminus B = \{u \in U \mid u \in A \text{ e } u \notin B\}$.
  - Differenza simmetrica: $A \,\Delta\, B = \{u \in U \mid u \in A \text{ e } u \notin B, \text{ oppure } u \in B \text{ e } u \notin A\}$.
- Proprietà delle operazioni insiemistiche:
  - $\overline{A} = U \setminus A$, $\overline{\emptyset} = U$, $\overline{U} = \emptyset$.
  - $A \cup B$ è il più piccolo insieme che include sia A che B e quindi:
    - $A \subseteq B$ sse $A \cup B = B$.
    - $A \subseteq B$ sse $\overline{A} \cup B = U$.
  - $A \cap B$ è il più grande insieme incluso sia in A che in B e quindi:
    - $A \subseteq B$ sse $A \cap B = A$.
    - $A \subseteq B$ sse $A \cap \overline{B} = \emptyset$.
  - Le due operazioni di differenza possono essere derivate dalle altre tre operazioni:
    - $A \setminus B = A \cap \overline{B}$.
    - $A \,\Delta\, B = (A \setminus B) \cup (B \setminus A) = (A \cup B) \setminus (A \cap B)$.
- La struttura algebrica $(\mathcal{P}(U), \cup, \cap, \bar{}, \emptyset, U)$ è un reticolo booleano, cioè soddisfa le seguenti leggi:
  - Commutatività:
    - $A \cup B = B \cup A$.
    - $A \cap B = B \cap A$.
  - Associatività:
    - $(A \cup B) \cup C = A \cup (B \cup C)$.
    - $(A \cap B) \cap C = A \cap (B \cap C)$.
  - Assorbimento:
    - $A \cup (A \cap B) = A$.
    - $A \cap (A \cup B) = A$.
  - Distributività:
    - $A \cup (B \cap C) = (A \cup B) \cap (A \cup C)$.
    - $A \cap (B \cup C) = (A \cap B) \cup (A \cap C)$.
  - Elemento neutro:
    - $A \cup \emptyset = A$.
    - $A \cap U = A$.
  - Elemento inverso:
    - $A \cup \overline{A} = U$.
    - $A \cap \overline{A} = \emptyset$.

- La struttura algebrica $(\mathcal{P}(U), \cup, \cap, \overline{\ }, \emptyset, U)$ possiede inoltre le seguenti proprietà derivate:
  - Elemento assorbente:
    - $A \cup U = U$.
    - $A \cap \emptyset = \emptyset$.
  - Idempotenza:
    - $A \cup A = A$.
    - $A \cap A = A$.
  - Doppia inversione:
    - $\overline{\overline{A}} = A$.
  - Leggi di De Morgan:
    - $\overline{A \cup B} = \overline{A} \cap \overline{B}$.
    - $\overline{A \cap B} = \overline{A} \cup \overline{B}$.

2.2 Relazioni, Funzioni, Operazioni

- Molto spesso in matematica si cerca di stabilire delle relazioni tra gli elementi di vari insiemi. Queste relazioni possono essere di diversa natura e tra le più importanti citiamo le relazioni d'ordine, le relazioni d'equivalenza e le relazioni funzionali (che includono le operazioni). Esse sono definite sulla base del concetto di coppia ordinata e di un'ulteriore operazione insiemistica detta prodotto cartesiano.
- Dati due elementi a e b, la coppia ordinata (a, b) è definita come l'insieme $\{\{a, b\}, \{a\}\}$. Si noti che i due insiemi $\{a, b\}$ e $\{b, a\}$ sono uguali perché l'ordine in cui gli elementi di un insieme sono elencati è irrilevante, mentre le due coppie ordinate (a, b) e (b, a) sono diverse quando $a \neq b$.
- Dati due insiemi A e B, il loro prodotto cartesiano è l'insieme di tutte le coppie ordinate formate dai loro elementi, cioè $A \times B = \{(a, b) \mid a \in A \text{ e } b \in B\}$. Questa operazione non è commutativa in generale.
- Una relazione $\mathcal{R}$ tra l'insieme A e l'insieme B è un sottoinsieme del prodotto cartesiano dei due insiemi, cioè $\mathcal{R} \subseteq A \times B$. Quando $A = B$, diciamo che $\mathcal{R}$ è una relazione binaria su A. Poiché le relazioni sono insiemi, ad esse sono applicabili tutte le operazioni sugli insiemi.
- Date due relazioni $\mathcal{R}_1 \subseteq A \times B$ ed $\mathcal{R}_2 \subseteq B \times C$, la loro composizione è definita ponendo $\mathcal{R}_1 \circ \mathcal{R}_2 = \{(a, c) \in A \times C \mid \text{esiste } b \in B \text{ tale che } (a, b) \in \mathcal{R}_1 \text{ e } (b, c) \in \mathcal{R}_2\}$. Nemmeno questa operazione è commutativa in generale.
- Sia $\mathcal{R} \subseteq A \times B$:
  - Il dominio di $\mathcal{R}$ è l'insieme $dom(\mathcal{R}) = \{a \in A \mid \text{esiste } b \in B \text{ tale che } (a, b) \in \mathcal{R}\}$, incluso in A.
  - Il codominio di $\mathcal{R}$ è l'insieme $cod(\mathcal{R}) = \{b \in B \mid \text{esiste } a \in A \text{ tale che } (a, b) \in \mathcal{R}\}$, incluso in B.
  - Il campo di $\mathcal{R}$ è l'insieme $campo(\mathcal{R}) = dom(\mathcal{R}) \cup cod(\mathcal{R})$, incluso in $A \cup B$.
- Data una relazione $\mathcal{R} \subseteq A \times A$, diciamo che essa è:
  - Riflessiva sse $(a, a) \in \mathcal{R}$ per ogni $a \in campo(\mathcal{R})$.
  - Antiriflessiva sse $(a, a) \notin \mathcal{R}$ per ogni $a \in campo(\mathcal{R})$.
  - Simmetrica sse $(a_1, a_2) \in \mathcal{R}$ implica $(a_2, a_1) \in \mathcal{R}$ per ogni $a_1, a_2 \in campo(\mathcal{R})$.
  - Antisimmetrica sse $(a_1, a_2) \in \mathcal{R}$ implica $(a_2, a_1) \notin \mathcal{R}$ per ogni $a_1, a_2 \in campo(\mathcal{R})$ con $a_1 \neq a_2$.
  - Transitiva sse $(a_1, a_2) \in \mathcal{R}$ e $(a_2, a_3) \in \mathcal{R}$ implicano $(a_1, a_3) \in \mathcal{R}$ per ogni $a_1, a_2, a_3 \in campo(\mathcal{R})$.

- Diciamo che $\mathcal{R} \subseteq A \times A$ è una relazione d'ordine parziale sse $\mathcal{R}$ è riflessiva, antisimmetrica e transitiva. Diciamo che $\mathcal{R}$ è una relazione d'ordine totale o lineare sse $\mathcal{R}$ soddisfa anche la seguente proprietà chiamata dicotomia: $(a_1, a_2) \in \mathcal{R}$ o $(a_2, a_1) \in \mathcal{R}$ per ogni $a_1, a_2 \in \mathit{campo}(\mathcal{R})$.
- Esempi:
  - $\leq$ è una relazione d'ordine totale su $\mathbb{N}$.
  - $\subseteq$ è una relazione d'ordine parziale su $\mathcal{P}(U)$. Essa non è totale: se $U = \{u_1, u_2\}$ abbiamo che $\mathcal{P}(U) = \{\emptyset, \{u_1\}, \{u_2\}, U\}$ dove $\{u_1\} \not\subseteq \{u_2\}$ e $\{u_2\} \not\subseteq \{u_1\}$.
- Diciamo che $\mathcal{R} \subseteq A \times A$ è una relazione d'equivalenza sse $\mathcal{R}$ è riflessiva, simmetrica e transitiva. In tal caso, chiamiamo insieme quoziente l'insieme $\mathit{campo}(\mathcal{R})/\mathcal{R}$ di tutte le classi d'equivalenza $[a]_\mathcal{R} = \{a' \in A \mid (a, a') \in \mathcal{R}\}$ per $a \in \mathit{campo}(\mathcal{R})$, il quale risulta essere una partizione di $\mathit{campo}(\mathcal{R})$, cioè:
  - $[a]_\mathcal{R} \neq \emptyset$ per ogni $a \in \mathit{campo}(\mathcal{R})$.
  - $[a_1]_\mathcal{R} \cap [a_2]_\mathcal{R} = \emptyset$ per ogni $a_1, a_2 \in \mathit{campo}(\mathcal{R})$ tali che $(a_1, a_2) \notin \mathcal{R}$.
  - $\bigcup_{a \in \mathit{campo}(\mathcal{R})} [a]_\mathcal{R} = \mathit{campo}(\mathcal{R})$.
- Esempi:
  - $=$ è una relazione d'equivalenza su $\mathbb{N}$ e su $\mathcal{P}(U)$.
  - $\mathcal{R}_5 = \{(n_1, n_2) \in \mathbb{N} \times \mathbb{N} \mid n_1 \text{ ed } n_2 \text{ danno lo stesso resto nella divisione per } 5\}$ è una relazione d'equivalenza le cui classi sono $[0]_{\mathcal{R}_5} = \{5 \cdot n \mid n \in \mathbb{N}\}$, $[1]_{\mathcal{R}_5} = \{5 \cdot n + 1 \mid n \in \mathbb{N}\}$, $[2]_{\mathcal{R}_5} = \{5 \cdot n + 2 \mid n \in \mathbb{N}\}$, $[3]_{\mathcal{R}_5} = \{5 \cdot n + 3 \mid n \in \mathbb{N}\}$, $[4]_{\mathcal{R}_5} = \{5 \cdot n + 4 \mid n \in \mathbb{N}\}$.
- Sia $\mathcal{R} \subseteq A \times A$:
  - La chiusura riflessiva di $\mathcal{R}$ è data da $\mathcal{R} \cup \mathcal{I}_{\mathit{campo}(\mathcal{R})}$, dove $\mathcal{I}_{A'} = \{(a, a) \mid a \in A'\}$ è la relazione identità su $A' \subseteq A$ e vale che $\mathcal{I}_A$ è l'elemento neutro della composizione di relazioni binarie su A.
  - La chiusura simmetrica di $\mathcal{R}$ è data da $\mathcal{R} \cup \mathcal{R}^{-1}$, dove $\mathcal{R}^{-1} = \{(a_2, a_1) \mid (a_1, a_2) \in \mathcal{R}\}$ è la relazione inversa di $\mathcal{R}$.
- Diciamo che $\mathcal{R} \subseteq A \times B$ è una relazione funzionale tra A e B, o equivalentemente che $\mathcal{R}$ è una funzione da A a B, sse per ogni $a \in \mathit{dom}(\mathcal{R})$ esiste esattamente un $b \in B$ tale che $(a, b) \in \mathcal{R}$. In tal caso scriviamo $\mathcal{R} : A \to B$ e poniamo $\mathcal{R}(a) = b$ per indicare che $(a, b) \in \mathcal{R}$.
- Data una funzione $f : A \to B$, diciamo che essa è:
  - Parziale sse $\mathit{dom}(f) \subset A$.
  - Totale sse $\mathit{dom}(f) = A$.
  - Iniettiva sse per ogni $b \in B$ esiste al più un $a \in \mathit{dom}(f)$ tale che $f(a) = b$.
  - Suriettiva sse per ogni $b \in B$ esiste almeno un $a \in \mathit{dom}(f)$ tale che $f(a) = b$.
  - Biiettiva sse per ogni $b \in B$ esiste esattamente un $a \in \mathit{dom}(f)$ tale che $f(a) = b$.
- Le funzioni possono essere usate per caratterizzare gli insiemi e la loro cardinalità:
  - Ogni insieme A può essere individuato attraverso la sua funzione caratteristica $\chi_A : U \to \{0, 1\}$ così definita: $\chi_A(u) = 1$ se $u \in A$, $\chi_A(u) = 0$ se $u \notin A$.
  - Due insiemi A e B sono equipotenti o equinumerosi, cioè $|A| = |B|$, sse esiste una funzione totale e biiettiva da A a B.
  - Un insieme non vuoto A è finito sse esiste $n \in \mathbb{N} \setminus \{0\}$ tale che A è equipotente a $\{1, \ldots, n\}$, nel qual caso $|A| = n$, mentre $|\emptyset| = 0$.
  - Un insieme A è infinito sse è equipotente a un suo sottoinsieme proprio, cioè diverso da $\emptyset$ e da A.
  - Un insieme infinito A è numerabile sse è equipotente a $\mathbb{N}$, nel qual caso $|A| = \aleph_0$, mentre $|\mathbb{R}| > \aleph_0$.

- Tutte le operazioni (come quelle aritmetiche e quelle insiemistiche) sono formalizzate tramite funzioni e vengono classificate come unarie o binarie a seconda che il dominio delle corrispondenti funzioni sia costituito da un solo insieme (inversione di segno; complementazione) o dal prodotto cartesiano di due insiemi (addizione, sottrazione, moltiplicazione, divisione; unione, intersezione, differenza, differenza simmetrica, prodotto cartesiano).
- Le operazioni non vengono espresse con la stessa notazione delle funzioni. Le operazioni unarie sono rappresentate in notazione prefissa — cioè il simbolo dell'operazione viene scritto prima dell'operando senza racchiudere quest'ultimo tra parentesi, ad esempio $-n$ invece di $-(n)$ — mentre le operazioni binarie sono rappresentate in notazione infissa — cioè il simbolo dell'operazione viene scritto in mezzo ai due operandi, ad esempio $n_1 + n_2$ invece di $+(n_1, n_2)$.
- Un insieme è detto essere chiuso rispetto a un'operazione sse l'operazione fornisce un risultato appartenente a quell'insieme quando viene applicata a operandi di quell'insieme. Ad esempio, $\mathbb{N}$ non è chiuso rispetto alla sottrazione, $\mathbb{Z}$ non è chiuso rispetto alla divisione, $\mathbb{Q}$ non è chiuso rispetto all'estrazione di radice di numeri positivi, $\mathbb{R}$ non è chiuso rispetto all'estrazione di radice di numeri negativi e $\mathcal{P}(U)$ non è chiuso rispetto al prodotto cartesiano.
- Una relazione d'equivalenza su un insieme chiuso rispetto a un'operazione è detta essere una congruenza rispetto a quell'operazione sse, ogni volta che si sostituisce un operando con un altro operando equivalente al primo, si ottiene un risultato equivalente a quello originario. Formalmente, date una relazione d'equivalenza $\mathcal{R} \subseteq A \times A$ e un'operazione n-aria $\odot : A \times A \times \cdots \times A \to A$, la relazione $\mathcal{R}$ è una congruenza rispetto all'operazione $\odot$ sse per ogni $a_1, a_2, \ldots, a_n, a'_1, a'_2, \ldots, a'_n \in A$ risulta che, se $(a_i, a'_i) \in \mathcal{R}$ per ogni $1 \leq i \leq n$, allora $(\odot(a_1, a_2, \ldots, a_n), \odot(a'_1, a'_2, \ldots, a'_n)) \in \mathcal{R}$.
- L'importanza di essere una congruenza è dovuta al fatto che questa proprietà supporta il ragionamento composizionale. Ad esempio, le manipolazioni che si possono apportare a singole parti di un'espressione aritmetica o insiemistica garantiscono la preservazione del valore dell'espressione originaria proprio grazie al fatto che la relazione $=$ è una congruenza rispetto alle operazioni aritmetiche e insiemistiche. Se consideriamo $e_1 + 4 \cdot 3 + e_2$ dove e_1 ed e_2 sono espressioni aritmetiche arbitrarie, poiché $4 \cdot 3 = 12$ e $=$ è una congruenza rispetto all'addizione, possiamo sostituire $4 \cdot 3$ con 12 nell'espressione originaria senza modificarne il valore, cioè $e_1 + 4 \cdot 3 + e_2 = e_1 + 12 + e_2$. In altri termini, $4 \cdot 3$ fa 12 non solo in isolamento, ma anche in ogni possible contesto aritmetico.

2.3 Principio di Induzione

- Il principio di induzione è il quinto dei postulati introdotti alla fine del 1800 da Giuseppe Peano per dare una definizione assiomatica di $\mathbb{N}$ (e più in generale dell'aritmetica, nota come aritmetica di Peano) come il più piccolo insieme che contiene 0 ed è chiuso rispetto all'operazione unaria di successore:

  1. Esiste un elemento $0 \in \mathbb{N}$.
  2. Esiste una funzione totale $succ : \mathbb{N} \to \mathbb{N}$.
  3. Per ogni $n \in \mathbb{N}$, $succ(n) \neq 0$.
  4. Per ogni $n, n' \in \mathbb{N}$, se $n \neq n'$ allora $succ(n) \neq succ(n')$.
  5. Se M è un sottoinsieme di $\mathbb{N}$ tale che:
     (a) $0 \in M$;
     (b) per ogni $n \in \mathbb{N}$, $n \in M$ implica $succ(n) \in M$;
     allora $M = \mathbb{N}$.

- Gli elementi di $\mathbb{N}$ sono quindi $0, succ(0), succ(succ(0)), \ldots$ dove $succ(0)$ viene denotato col simbolo 1, $succ(succ(0))$ viene denotato col simbolo 2, e così via. In altri termini, il numero di unità che compongono un numero naturale è pari al numero di volte che bisogna applicare la funzione successore allo 0 per ottenere il numero naturale considerato.

- Oltre che $\mathbb{N}$ stesso, il principio di induzione consente di definire *in modo finito* le quattro operazioni aritmetiche su $\mathbb{N}$ avendo a disposizione la funzione $succ : \mathbb{N} \to \mathbb{N}$ e introducendo la funzione totale $pred : \mathbb{N}_{\neq 0} \to \mathbb{N}$ tale che $pred(succ(n)) = n$ per ogni $n \in \mathbb{N}$ e $succ(pred(n)) = n$ per ogni $n \in \mathbb{N}_{\neq 0}$. Le seguenti operazioni binarie, che hanno due parametri denotati con m ed n rispettivamente, sono tutte definite procedendo per induzione sul parametro n:
  - Addizione: $m \oplus n = \begin{cases} m & \text{se } n = 0 \\ succ(m) \oplus pred(n) & \text{se } n \neq 0 \end{cases}$

    sulla cui base si definisce $m \leq n$ sse esiste $m' \in \mathbb{N}$ tale che $m \oplus m' = n$ e conseguentemente $m < n$ sse $m \leq n$ con $m \neq n$, nonché $m \geq n$ sse $n \leq m$ ed $m > n$ sse $n < m$.
  - Sottrazione: $m \ominus n = \begin{cases} m & \text{se } n = 0 \\ pred(m) \ominus pred(n) & \text{se } n > 0 \end{cases}$ dove $m \geq n$.
  - Moltiplicazione: $m \otimes n = \begin{cases} 0 & \text{se } n = 0 \\ m \oplus (m \otimes pred(n)) & \text{se } n > 0 \end{cases}$.
  - Divisione: $m \oslash n = \begin{cases} 0 & \text{se } m < n \\ succ((m \ominus n) \oslash n) & \text{se } m \geq n \end{cases}$ dove $n \neq 0$.
- Esempi di applicazione delle precedenti definizioni:
  - $5 \oplus 2 = 6 \oplus 1 = 7 \oplus 0 = 7$.
  - $5 \ominus 2 = 4 \ominus 1 = 3 \ominus 0 = 3$.
  - $5 \otimes 2 = 5 \oplus (5 \otimes 1) = 5 \oplus (5 \oplus (5 \otimes 0)) = 5 \oplus (5 \oplus 0) = 5 \oplus 5 = \ldots = 10$.
  - $5 \oslash 2 = succ((5 \ominus 2) \oslash 2) = \ldots = succ(3 \oslash 2) = succ(succ((3 \ominus 2) \oslash 2)) = \ldots = succ(succ(1 \oslash 2)) = succ(succ(0)) = succ(1) = 2$.
- Il principio di induzione fornisce in generale un meccanismo per descrivere *in modo finito* un insieme *infinito numerabile*. Esso prende il nome di definizione ricorsiva e comprende uno o più casi base – in ciascuno dei quali la definizione è espressa in modo diretto – e uno o più casi induttivi – in ciascuno dei quali la definizione è espressa ricorrendo a una definizione della stessa natura che però è più vicina a uno dei casi base. Nella definizione ricorsiva bisogna specificare su quale entità si procede per induzione.
- Esempi:
  - Il fattoriale di un numero $n \in \mathbb{N}$ tale che $n \geq 1$ viene normalmente definito come $n! = \prod_{i=1}^{n} i$. È tuttavia possibile definirlo anche nel seguente modo ricorsivo:
    $$n! = \begin{cases} 1 & \text{se } n = 1 \\ n \cdot (n-1)! & \text{se } n > 1 \end{cases}$$
  - La potenza cartesiana n-esima di un insieme A, con $n \in \mathbb{N}$ tale che $n \geq 1$, viene normalmente definita come $A^n = \underbrace{A \times \cdots \times A}_{n \text{ volte}}$. È tuttavia possibile definirla anche nel seguente modo ricorsivo:
    $$A^n = \begin{cases} A & \text{se } n = 1 \\ A^{n-1} \times A & \text{se } n > 1 \end{cases}$$
  - La chiusura transitiva di una relazione $\mathcal{R}$ su un insieme A è la relazione $\mathcal{R}^+$ ottenuta da $\mathcal{R}$ rendendola transitiva. Usando l'operazione di composizione di relazioni, poniamo $\mathcal{R}^+ = \bigcup_{n \in \mathbb{N}_{\neq 0}} \mathcal{R}^n$ dove $\mathcal{R}^n = \underbrace{\mathcal{R} \circ \cdots \circ \mathcal{R}}_{n \text{ volte}}$ può essere definita ricorsivamente come segue:
    $$\mathcal{R}^n = \begin{cases} \mathcal{R} & \text{se } n = 1 \\ \mathcal{R}^{n-1} \circ \mathcal{R} & \text{se } n > 1 \end{cases}$$
    Poiché $\mathcal{R}$ è interpretabile come un grafo i cui vertici corrispondono agli elementi di A e i cui archi corrispondono agli elementi di $\mathcal{R}$, la relazione $\mathcal{R}^+$ stabilisce se esiste un percorso tra ciascuna coppia di vertici del grafo, con $\mathcal{R}^n$ rappresentante l'esistenza di un percorso composto da $n \in \mathbb{N}_{\neq 0}$ archi. La chiusura riflessiva e transitiva di $\mathcal{R}$ è data da $\mathcal{R}^* = \mathcal{R}^+ \cup \mathcal{I}_{campo(\mathcal{R})}$.

  - Dato un insieme non vuoto di simboli Σ detto alfabeto, l'insieme Σ^* di tutte le sequenze (o stringhe o parole o frasi) di lunghezza finita basate su Σ viene ottenuto attraverso una definizione ricorsiva analoga a quella della chiusura riflessiva e transitiva di una relazione, che prende il nome di chiusura (o stella) di Kleene. Precisamente $\Sigma^* = \bigcup_{n\in\mathbb{N}} \Sigma^n$ dove:
$$\Sigma^n = \begin{cases} \{\varepsilon\} & \text{se } n = 0 \\ \{\sigma a \mid \sigma \in \Sigma^{n-1} \text{ e } a \in \Sigma\} & \text{se } n > 0 \end{cases}$$
utilizzando ε per denotare la sequenza vuota e σa per denotare la concatenazione di σ e a. Dunque Σ^n rappresenta l'insieme delle sequenze composte da $n \in \mathbb{N}$ simboli di Σ. Si dice linguaggio su Σ un qualsiasi sottoinsieme L di Σ^*. Il motivo per cui un linguaggio L su Σ non coincide necessariamente con l'intero Σ^* è che L contiene tutte e sole le sequenze di Σ^* che soddisfano le regole grammaticali del linguaggio stesso, cioè le cosiddette sequenze sintatticamente ben formate.

- Ci sono due formulazioni equivalenti del principio di induzione:
  - Siano $n_0 \in \mathbb{N}$ e $\mathcal{Q}$ una proprietà definita su ogni $n \in \mathbb{N}$ tale che $n \geq n_0$. Se:
    1. $\mathcal{Q}$ è soddisfatta da n_0;
    2. per ogni $n \in \mathbb{N}$ tale che $n \geq n_0$, $\mathcal{Q}$ soddisfatta da n implica $\mathcal{Q}$ soddisfatta da $succ(n)$;

    allora $\mathcal{Q}$ è soddisfatta da ogni $n \in \mathbb{N}$ tale che $n \geq n_0$.
  - Siano $n_0 \in \mathbb{N}$ e $\mathcal{Q}$ una proprietà definita su ogni $n \in \mathbb{N}$ tale che $n \geq n_0$. Se:
    1. $\mathcal{Q}$ è soddisfatta da n_0;
    2. per ogni $n \in \mathbb{N}$ tale che $n \geq n_0$, $\mathcal{Q}$ soddisfatta da ogni $m \in \mathbb{N}$ tale che $n_0 \leq m \leq n$ implica $\mathcal{Q}$ soddisfatta da $succ(n)$;

    allora $\mathcal{Q}$ è soddisfatta da ogni $n \in \mathbb{N}$ tale che $n \geq n_0$.

- Il principio di induzione fornisce dunque anche una condizione sufficiente per verificare se una proprietà è soddisfatta da *tutti* gli elementi di un insieme *infinito numerabile* dotato di una relazione d'ordine totale che ammette l'elemento minimo. La dimostrazione si suddivide in due parti. Nella prima parte (caso base), si verifica direttamente se la proprietà è soddisfatta dall'elemento minimo dell'insieme. Nella seconda parte (caso induttivo), si verifica se la proprietà è soddisfatta da un generico elemento dell'insieme che è maggiore del minimo, assumendo che la proprietà sia soddisfatta dall'elemento che lo precede o da tutti gli elementi che lo precedono (ipotesi induttiva).

- Esempi:
  - Dimostriamo che $\sum_{i=1}^{n} i = \frac{n\cdot(n+1)}{2}$ per ogni $n \in \mathbb{N}$ tale che $n \geq 1$ procedendo per induzione su n:
    * Sia $n = 1$. Risulta $\sum_{i=1}^{1} i = 1 = \frac{1\cdot(1+1)}{2}$ e quindi la proprietà è vera per $n = 1$.
    * Dato un arbitrario $n \in \mathbb{N}$ tale che $n \geq 1$, supponiamo che $\sum_{i=1}^{n} i = \frac{n\cdot(n+1)}{2}$. Risulta $\sum_{i=1}^{(n+1)} i = (n+1) + \sum_{i=1}^{n} i = (n+1) + \frac{n\cdot(n+1)}{2}$ per ipotesi induttiva. Poiché $(n+1) + \frac{n\cdot(n+1)}{2} = \frac{2\cdot(n+1)+n\cdot(n+1)}{2} = \frac{(n+1)\cdot(n+2)}{2}$, abbiamo $\sum_{i=1}^{(n+1)} i = \frac{(n+1)\cdot((n+1)+1)}{2}$ e quindi la proprietà è vera per $n+1$.

– Dimostriamo che $|\mathcal{P}(A)| = 2^n$ per ogni insieme finito A tale che $|A| = n$ procedendo per induzione su $n \in \mathbb{N}$:

  * Sia $n = 0$. In questo caso $A = \emptyset$ e risulta $|\mathcal{P}(\emptyset)| = |\{\emptyset\}| = 1 = 2^0$, quindi la proprietà è vera per l'insieme finito di cardinalità $n = 0$.
  * Dato un arbitrario $n \in \mathbb{N}$, supponiamo che per ogni insieme A tale che $|A| = n$ risulti $|\mathcal{P}(A)| = 2^n$ e consideriamo un qualsiasi insieme A' di cardinalità $n+1$, che quindi possiamo scrivere come $A' = A'' \cup \{a\}$ con $a \notin A''$. Da $\mathcal{P}(A') = \{B \mid B \subseteq A''\} \cup \{B \cup \{a\} \mid B \subseteq A''\}$ con $\{B \mid B \subseteq A''\} \cap \{B \cup \{a\} \mid B \subseteq A''\} = \emptyset$ e $|\{B \mid B \subseteq A''\}| = |\{B \cup \{a\} \mid B \subseteq A''\}| = |\mathcal{P}(A'')|$, segue che $|\mathcal{P}(A')| = 2 \cdot |\mathcal{P}(A'')| = 2 \cdot 2^n$ per ipotesi induttiva. Poiché $2 \cdot 2^n = 2^{(n+1)}$, abbiamo $|\mathcal{P}(A')| = 2^{(n+1)}$ e quindi la proprietà è vera per ogni insieme finito di cardinalità $n+1$. ■fltplf_2

Capitolo 3

Lambda Calcolo

3.1 Sintassi del Lambda Calcolo

- In matematica una funzione è un caso particolare di relazione. Precisamente, una funzione $f : A \to B$ è un sottoinsieme di $A \times B$ che a ogni elemento di A associa al più un elemento di B, ossia è un insieme di coppie ordinate, ciascuna formata da un elemento di A e un elemento di B, che gode di una sorta di proprietà di univocità da A verso B. Questa è una visione *estensionale* del concetto di funzione, in quanto si focalizza sull'insieme di coppie ordinate senza considerare il procedimento computazionale tramite il quale si ottiene il risultato $b = f(a) \in B$ partendo dall'argomento $a \in A$.
- Nel 1932 Alonzo Church introdusse il λ-calcolo come parte di una teoria generale delle funzioni e della logica, da usare quale fondamento della matematica. Church era interessato a creare un calcolo che catturasse gli aspetti computazionali delle funzioni, dove per calcolo si intende un sistema formale dotato di una sintassi per generare termini in un certo formato accompagnata da un insieme di regole di riscrittura per trasformare i termini in altri termini. Ciò originò una visione *intensionale* del concetto di funzione, cioè caratterizzata dalle regole computazionali che conducono dall'argomento al risultato.
- Il λ-calcolo descrive le funzioni nella loro piena generalità – incluse funzioni ricorsive e funzioni di ordine superiore (cioè aventi funzioni come argomenti) – attraverso una sintassi estremamente semplice che distingue tra definizione di funzione, detta λ-astrazione, e applicazione di funzione. Questa distinzione consente di superare l'ambiguità che talvolta viene a crearsi in merito alla notazione $f(x)$, usata a volte per denotare la definizione di f e altre volte per denotare il valore che f assume in x.
- Dato un insieme numerabile Var di simboli di variabile, i quali verranno indicati con le lettere x, y, z, l'insieme Λ dei λ-termini è il più piccolo linguaggio su $\mathit{Var} \cup \{\lambda, ., (,)\}$ tale che:
  - $\mathit{Var} \subseteq \Lambda$.
  - $(\lambda x \,.\, E) \in \Lambda$ per ogni $x \in \mathit{Var}$ ed $E \in \Lambda$, detta λ-astrazione, dove λ è detto legatore per x in E.
  - $(E_1 \, E_2) \in \Lambda$ per ogni $E_1, E_2 \in \Lambda$, detta applicazione di E_1 a E_2.
- Se vediamo la λ-astrazione come un operatore unario prefisso e l'applicazione come un operatore binario infisso, possiamo alternativamente definire Λ come il più piccolo insieme che include Var ed è chiuso rispetto a λ-astrazione e applicazione.
- Per evitare l'uso sistematico delle parentesi assumiamo che l'applicazione abbia precedenza sulla λ-astrazione – così $\lambda x \,.\, E_1 \, E_2$ sta per $(\lambda x \,.\, (E_1 \, E_2))$ – e che la λ-astrazione sia associativa da destra – $\lambda x \,.\, \lambda y \,.\, E$ sta per $(\lambda x \,.\, (\lambda y \,.\, E))$ – mentre l'applicazione da sinistra – $E_1 \, E_2 \, E_3$ sta per $((E_1 \, E_2) \, E_3)$ – da cui la sintassi semplificata $E ::= x \mid \lambda x \,.\, E \mid E \, E$.
- Esempi:
  - $\lambda x \,.\, x$ è la funzione identità mentre $\lambda x \,.\, \lambda y \,.\, x$ e $\lambda x \,.\, \lambda y \,.\, y$ selezionano uno dei loro due argomenti.
  - $\lambda x \,.\, x + 1$ è la funzione successore mentre $\lambda x \,.\, x - 1$ è la funzione predecessore.
  - $(\lambda x \,.\, x + 1) \, 2$ è l'applicazione della funzione successore al valore 2.

- Nei precedenti esempi abbiamo usato i simboli $+$, $-$, 1, 2 sebbene essi non facciano parte della sintassi del λ-calcolo. Ci siamo presi questa libertà perché i numeri naturali e le loro operazioni sono rappresentabili nel λ-calcolo. Ispirandosi a Peano ogni $n \in \mathbb{N}$ è espresso da una funzione $\underline{n}$ detta numerale di Church che, dati una funzione s e un argomento z, applica n volte s partendo da z. Precisamente:
  - $\underline{0} = \lambda s \,.\, \lambda z \,.\, z$ così la rappresentazione di 0 restituisce z.
  - $\underline{1} = \lambda s \,.\, \lambda z \,.\, s\, z$ così la rappresentazione di 1 restituisce una singola applicazione di s a z.
  - $\underline{2} = \lambda s \,.\, \lambda z \,.\, s\,(s\, z)$ così la rappresentazione di 2 restituisce una doppia applicazione di s a z.
  - $\underline{n} = \lambda s \,.\, \lambda z \,.\, s\,(\ldots(s\, z)\ldots)$ dove l'applicazione di s è ripetuta n volte come se $\underline{n}$ fosse un iteratore.
- Diciamo che $E' \in \Lambda$ è un sottotermine di $E \in \Lambda$ sse E' compare in E. Più precisamente, l'insieme $st(E)$ dei sottotermini di $E \in \Lambda$ è definito per induzione sulla struttura sintattica di E come segue:
$$st(E) \;=\; \{E\} \cup \begin{cases} \emptyset & \text{se } E \in \mathit{Var} \\ st(E') & \text{se } E \text{ è della forma } \lambda x \,.\, E' \\ st(E'_1) \cup st(E'_2) & \text{se } E \text{ è della forma } E'_1\, E'_2 \end{cases}$$
- Siano $E \in \Lambda$ e $x \in \mathit{Var}$:
  - x occorre in E sse x compare in un sottotermine di E privo di legatori (cioè privo di λ).
  - Un'occorrenza di x in E è legata (a un legatore) se compare in un sottotermine di E della forma $\lambda x \,.\, E'$ – nel qual caso E' è il campo d'azione di quel legatore – altrimenti è libera.
  - E è chiuso se ogni occorrenza di ogni variabile che compare in E è legata, altrimenti è aperto.
- Esempi:
  - $st(\lambda x \,.\, \lambda y \,.\, x\, y) \;=\; \{\lambda x \,.\, \lambda y \,.\, x\, y,\ \lambda y \,.\, x\, y,\ x\, y,\ x,\ y\}$.
  - La variabile x non occorre nel termine aperto $\lambda x \,.\, y$ perché il solo sottotermine senza legatori è y.
  - Il termine chiuso $(\lambda x \,.\, x)\, \lambda x \,.\, x$ ha due occorrenze della stessa variabile che ricadono nel campo d'azione di due legatori che compaiono in due sottotermini indipendenti, mentre in $\lambda x \,.\, x\,(\lambda x \,.\, x)$ il campo d'azione del legatore che sta a destra è contenuto in quello del legatore che sta a sinistra.
- Sia $E \in \Lambda$:
  - L'insieme $var(E)$ delle variabili che occorrono in E è definito per induzione sulla struttura sintattica di E come segue:
$$var(E) \;=\; \begin{cases} \{E\} & \text{se } E \in \mathit{Var} \\ var(E') & \text{se } E \text{ è della forma } \lambda x \,.\, E' \\ var(E'_1) \cup var(E'_2) & \text{se } E \text{ è della forma } E'_1\, E'_2 \end{cases}$$
  - L'insieme $varleg(E)$ delle variabili che occorrono legate in E è definito per induzione sulla struttura sintattica di E come segue:
$$varleg(E) \;=\; \begin{cases} \emptyset & \text{se } E \in \mathit{Var} \\ varleg(E') & \text{se } E \text{ è della forma } \lambda x \,.\, E' \text{ e } x \notin var(E') \\ varleg(E') \cup \{x\} & \text{se } E \text{ è della forma } \lambda x \,.\, E' \text{ e } x \in var(E') \\ varleg(E'_1) \cup varleg(E'_2) & \text{se } E \text{ è della forma } E'_1\, E'_2 \end{cases}$$
  - L'insieme $varlib(E)$ delle variabili che occorrono libere in E è definito per induzione sulla struttura sintattica di E come segue:
$$varlib(E) \;=\; \begin{cases} \{E\} & \text{se } E \in \mathit{Var} \\ varlib(E') \setminus \{x\} & \text{se } E \text{ è della forma } \lambda x \,.\, E' \\ varlib(E'_1) \cup varlib(E'_2) & \text{se } E \text{ è della forma } E'_1\, E'_2 \end{cases}$$
  - Osserviamo che $var(E) = varleg(E) \cup varlib(E)$, ma in generale $varleg(E) \cap varlib(E) \neq \emptyset$ come si vede in $(\lambda x \,.\, x)\, x$. Inoltre E è chiuso quando $varlib(E) = \emptyset$, aperto quando $varlib(E) \neq \emptyset$.
- Siano $E, F \in \Lambda$ e $y \in \mathit{Var}$. Il termine ottenuto da E sostituendo con F ogni occorrenza di y libera in E è definito per induzione sulla struttura sintattica di E come segue:
$$E[{}^{F}/_{y}] \;=\; \begin{cases} F & \text{se } E \in \mathit{Var} \text{ ed } E = y \\ E & \text{se } E \in \mathit{Var} \text{ ed } E \neq y \\ E & \text{se } E \text{ è della forma } \lambda y \,.\, E' \\ \lambda x \,.\, (E'[{}^{F}/_{y}]) & \text{se } E \text{ è della forma } \lambda x \,.\, E' \text{ e } x \neq y \text{ e } x \notin varlib(F) \\ \lambda z \,.\, (E'[{}^{z}/_{x}][{}^{F}/_{y}]) & \text{se } E \text{ è della forma } \lambda x \,.\, E' \text{ e } x \neq y \text{ e } x \in varlib(F) \text{ con } z \notin \{y\} \cup var(F) \cup var(E') \\ E'_1[{}^{F}/_{y}]\, E'_2[{}^{F}/_{y}] & \text{se } E \text{ è della forma } E'_1\, E'_2 \end{cases}$$

dove nella penultima clausola è necessaria una sostituzione preliminare della forma $[^z/_x]$, con z diversa da tutte le variabili presenti, altrimenti le occorrenze di x libere in F verrebbero erroneamente legate. Ad esempio, il termine aperto $\lambda x \,.\, x\, y$ verrebbe trasformato nel termine chiuso $\lambda x \,.\, x\, x$ se gli fosse direttamente applicata la sostituzione $[^x/_y]$, mentre la clausola in questione produce $\lambda z \,.\, ((x\, y)[^z/_x][^x/_y]) = \lambda z \,.\, ((z\, y)[^x/_y]) = \lambda z \,.\, z\, x$ che è ancora un termine aperto. ■fltplf_3

3.2 Semantica del Lambda Calcolo e Logica Combinatoria

- La semantica del λ-calcolo è costituita dalle seguenti tre regole di riscrittura di termini che fanno largo uso dell'operazione di sostituzione sintattica di variabili libere:
  - α-conversione: $\lambda x \,.\, E =_\alpha \lambda y \,.\, (E[^y/_x])$ purché $y \notin \mathit{varlib}(E)$.
  - β-conversione: $(\lambda x \,.\, E)\, F =_\beta E[^F/_x]$.
  - η-conversione: $\lambda x \,.\, E\, x =_\eta E$ purché $x \notin \mathit{varlib}(E)$.
- Indichiamo con $\equiv_\alpha$, $\equiv_\beta$, $\equiv_\eta$ le corrispondenti relazioni d'equivalenza su Λ indotte dalle chiusure riflessive, simmetriche e transitive delle tre regole rispettivamente. Esse sono congruenze rispetto a λ-astrazione e applicazione, cioè per ogni $\equiv \in \{\equiv_\alpha, \equiv_\beta, \equiv_\eta\}$ da $E_1 \equiv E_2$ segue che:
  - $\lambda x \,.\, E_1 \equiv \lambda x \,.\, E_2$ per ogni $x \in \mathit{Var}$.
  - $E_1\, F \equiv E_2\, F$ per ogni $F \in \Lambda$.
  - $F\, E_1 \equiv F\, E_2$ per ogni $F \in \Lambda$.
- Mentre l'operazione di sostituzione sintattica agisce sulle variabili libere, la regola di α-conversione permette di cambiare il nome delle occorrenze legate di una variabile assieme al corrispondente legatore. Essa è utile per identificare funzioni che differiscono soltanto per il nome delle variabili legate – ad esempio $\lambda x \,.\, x =_\alpha \lambda y \,.\, y$ – e consente di adottare la convenzione secondo cui all'interno di ogni termine i nomi delle variabili legate a occorrenze diverse di legatori siano sempre diversi tra di loro e dai nomi delle variabili libere.
- La regola di β-conversione, dove $(\lambda x \,.\, E)\, F$ è detto radicale ed $E[^F/_x]$ è detto ridotto, è la regola fondamentale del λ-calcolo in quanto formalizza l'effetto dell'applicazione di una funzione $\lambda x \,.\, E$ a un argomento F sostituendo il parametro effettivo F a ogni occorrenza del parametro formale x nel corpo E della funzione. Nel λ-calcolo vale quanto segue:
  - Ogni funzione con più argomenti è descritta come una funzione di un solo argomento che restituisce una funzione in cui sono gestiti i rimanenti argomenti, un fenomeno che prende il nome di currying. Di conseguenza non è necessario passare subito a una funzione tutti gli argomenti di cui necessita. Ad esempio la funzione somma è descritta da $\lambda y \,.\, \lambda x \,.\, x{+}y$ e quando viene applicata all'argomento 1 restituisce la funzione successore perché $(\lambda y \,.\, \lambda x \,.\, x + y)\, 1 =_\beta (\lambda x \,.\, x + y)[^1/_y] = \lambda x \,.\, x + 1$.
  - D'altra parte una funzione può essere passata come argomento a un'altra funzione, ossia è possibile trattare funzioni di ordine superiore. Ad esempio $(\lambda y \,.\, y\, 2)\, (\lambda x \,.\, x + 1) =_\beta (y\, 2)[^{\lambda x \,.\, x+1}/_y] = (\lambda x \,.\, x + 1)\, 2 =_\beta (x + 1)[^2/_x] = 2 + 1 = 3$.
- La regola di η-conversione, aggiunta successivamente al λ-calcolo, codifica l'uguaglianza estensionale tra due funzioni, cioè il fatto che due funzioni sono uguali sse producono lo stesso risultato quando vengono applicate al medesimo argomento. Essa equivale ad avere una regola che stabilisce che, se per ogni $F \in \Lambda$ risulta $E_1\, F \equiv_\beta E_2\, F$, allora $E_1 = E_2$. Infatti:
  - Da $E_1\, F \equiv_\beta E_2\, F$ per ogni $F \in \Lambda$ segue in particolare $E_1\, z \equiv_\beta E_2\, z$ e quindi $\lambda z \,.\, E_1\, z \equiv_\beta \lambda z \,.\, E_2\, z$ per $z \notin \mathit{varlib}(E_1) \cup \mathit{varlib}(E_2)$, da cui $E_1 = E_2$ applicando la η-conversione ad ambo i membri.
  - Poiché $(\lambda x \,.\, E\, x)\, F \equiv_\beta (E\, x)[^F/_x] = E[^F/_x]\, x[^F/_x] = E\, F$ per ogni $F \in \Lambda$ purché $x \notin \mathit{varlib}(E)$, dalla regola di uguaglianza estensionale segue che $\lambda x \,.\, E\, x = E$ purché $x \notin \mathit{varlib}(E)$.
- Le regole di β-conversione ed η-conversione prendono rispettivamente il nome di β-riduzione ed η-riduzione quando vengono viste come regole di riscrittura di termini orientate da sinistra a destra:
  - β-riduzione: $(\lambda x \,.\, E)\, F \longrightarrow_\beta E[^F/_x]$.
  - η-riduzione: $\lambda x \,.\, E\, x \longrightarrow_\eta E$ purché $x \notin \mathit{varlib}(E)$.

  Indichiamo con $\longrightarrow_\beta^*$ e $\longrightarrow_\eta^*$ le rispettive chiusure riflessive e transitive a meno di α-conversione.

- Esempi:
  - Posto $\underline{succ} = \lambda n\,.\,\lambda x\,.\,\lambda y\,.\,x\,(n\,x\,y)$, vale $\underline{succ}\,\underline{n} \longrightarrow_\beta^* \underline{n+1}$. Ad esempio $\underline{succ}\,\underline{0} = (\lambda n\,.\,\lambda x\,.\,\lambda y\,.\,x\,(n\,x\,y))\,(\lambda s\,.\,\lambda z\,.\,z) \longrightarrow_\beta (\lambda x\,.\,\lambda y\,.\,x\,(n\,x\,y))[^{\lambda s\,.\,\lambda z\,.\,z}/_n] = \lambda x\,.\,\lambda y\,.\,x\,((\lambda s\,.\,\lambda z\,.\,z)\,x\,y) \longrightarrow_\beta \lambda x\,.\,\lambda y\,.\,x\,((\lambda z\,.\,z)[^x/_s]\,y) = \lambda x\,.\,\lambda y\,.\,x\,((\lambda z\,.\,z)\,y) \longrightarrow_\beta \lambda x\,.\,\lambda y\,.\,x\,(z[^y/_z]) = \lambda x\,.\,\lambda y\,.\,x\,y =_\alpha \underline{1}$.
  - Posto $\underline{add} = \lambda m\,.\,\lambda n\,.\,\lambda x\,.\,\lambda y\,.\,m\,x\,(n\,x\,y)$, vale $\underline{add}\,\underline{m}\,\underline{n} \longrightarrow_\beta^* \underline{m+n}$.
  - Posto $\underline{molt} = \lambda m\,.\,\lambda n\,.\,\lambda x\,.\,m\,(n\,x)$, vale $\underline{molt}\,\underline{m}\,\underline{n} \longrightarrow_\beta^* \underline{m \cdot n}$.
  - Posto $\underline{esp} = \lambda m\,.\,\lambda n\,.\,m\,n$, vale $\underline{esp}\,\underline{m}\,\underline{n} \longrightarrow_\beta^* \underline{n^m}$.
  - Posto $\underline{I} = \lambda x\,.\,x$, vale $\underline{I}\,\underline{I} =_\alpha (\lambda x\,.\,x)\,(\lambda x'.\,x') \longrightarrow_\beta x[^{\lambda x'.\,x'}/_x] = \lambda x'.\,x' =_\alpha \underline{I}$, cioè $\underline{I}\,\underline{I} \longrightarrow_\beta^* \underline{I}$.
  - Posto $\underline{K} = \lambda x\,.\,\lambda y\,.\,x$, vale $\underline{K}\,E = (\lambda x\,.\,\lambda y\,.\,x)\,E \longrightarrow_\beta (\lambda y\,.\,x)[^E/_x] = \lambda y\,.\,E$ e quindi risulta $\underline{K}\,E\,F \longrightarrow_\beta^* E$ perché $y \notin varlib(E)$ altrimenti avremmo ridenominato λy nella sostituzione.
  - Posto $\underline{S} = \lambda x\,.\,\lambda y\,.\,\lambda z\,.\,x\,z\,(y\,z)$, vale $\underline{S}\,\underline{K}\,\underline{K} =_\alpha (\lambda x\,.\,\lambda y\,.\,\lambda z\,.\,x\,z\,(y\,z))\,(\lambda x'.\,\lambda y'.\,x')\,(\lambda x''.\,\lambda y''.\,x'') \longrightarrow_\beta (\lambda y\,.\,\lambda z\,.\,x\,z\,(y\,z))[^{\lambda x'.\,\lambda y'.\,x'}/_x]\,(\lambda x''.\,\lambda y''.\,x'') = (\lambda y\,.\,\lambda z\,.\,(\lambda x'.\,\lambda y'.\,x')\,z\,(y\,z))\,(\lambda x''.\,\lambda y''.\,x'') \longrightarrow_\beta (\lambda y\,.\,\lambda z\,.\,(\lambda y'.\,x')[^z/_{x'}]\,(y\,z))\,(\lambda x''.\,\lambda y''.\,x'') = (\lambda y\,.\,\lambda z\,.\,(\lambda y'.\,z)\,(y\,z))\,(\lambda x''.\,\lambda y''.\,x'') \longrightarrow_\beta (\lambda y\,.\,\lambda z\,.\,z[^{y\,z}/_{y'}])\,(\lambda x''.\,\lambda y''.\,x'') = (\lambda y\,.\,\lambda z\,.\,z)\,(\lambda x''.\,\lambda y''.\,x'') \longrightarrow_\beta (\lambda z\,.\,z)[^{\lambda x''.\,\lambda y''.\,x''}/_y] = \lambda z\,.\,z =_\alpha \underline{I}$, cioè $\underline{S}\,\underline{K}\,\underline{K} \longrightarrow_\beta^* \underline{I}$.
- I due simboli K ed S sono alla base della logica combinatoria, che come il λ-calcolo aveva l'obiettivo di investigare i fondamenti della matematica usando il concetto base di operazione anziché di insieme. La logica combinatoria venne introdotta nel 1924 da Moses Schönfinkel e fu poi indipendentemente riformulata nel 1930 da Haskell Curry. Diversamente dal λ-calcolo, la logica combinatoria non ha definizioni di funzioni né legatori per le variabili, ma solo i combinatori K ed S, che possono essere visti come funzioni di ordine superiore, e le loro applicazioni. Tuttavia K ed S sono sufficienti per implementare la λ-astrazione e la β-riduzione (variabili e applicazioni sono invece comuni a entrambi i formalismi).
- L'insieme $\mathcal{L}$ dei termini della logica combinatoria è il più piccolo linguaggio su $Var \cup \{K, S, (,)\}$ tale che:
  - $Var \subseteq \mathcal{L}$.
  - $K, S \in \mathcal{L}$, detti combinatori.
  - $(P\,Q) \in \mathcal{L}$ per ogni $P, Q \in \mathcal{L}$, detta applicazione di P a Q.

  Assumendo l'applicazione associativa da sinistra, la sintassi semplificata è $P ::= x \mid K \mid S \mid P\,P$.
- La semantica della logica combinatoria è costituita dalle seguenti regole di riduzione:
  - $K\,P\,Q \longrightarrow_{\mathcal{L}} P$.
  - $S\,P\,Q\,R \longrightarrow_{\mathcal{L}} P\,R\,(Q\,R)$.
  - Se $P \longrightarrow_{\mathcal{L}} P'$ allora $P\,Q \longrightarrow_{\mathcal{L}} P'\,Q$.
  - Se $Q \longrightarrow_{\mathcal{L}} Q'$ allora $P\,Q \longrightarrow_{\mathcal{L}} P\,Q'$.
- Dati $P, Q \in \mathcal{L}$ e $y \in Var$, denotiamo con $var(P)$ l'insieme delle variabili che occorrono in P – esse sono tutte libere per via dell'assenza di legatori – e con $P[^Q/_y]$ il termine ottenuto da P sostituendo Q a ogni occorrenza di y in P.
- Teorema: Definiamo i seguenti termini di logica combinatoria che simulano una λ-astrazione in base a tre casi che dipendono dalla forma del corpo della funzione:
  - $\hat{\lambda}x\,.\,x = S\,K\,K$ per ogni $x \in Var$.
  - $\hat{\lambda}x\,.\,P = K\,P$ per ogni $x \in Var$ e $P \in \mathcal{L}$ tali che $x \notin var(P)$.
  - $\hat{\lambda}x\,.\,P_1\,P_2 = S\,(\hat{\lambda}x\,.\,P_1)\,(\hat{\lambda}x\,.\,P_2)$ per ogni $x \in Var$ e $P_1, P_2 \in \mathcal{L}$ tali che $x \notin var(P_1) \cup var(P_2)$.

  Allora $(\hat{\lambda}x\,.\,P)\,Q \longrightarrow_{\mathcal{L}} P[^Q/_x]$ per ogni $x \in Var$ e $P, Q \in \mathcal{L}$.
- Dimostrazione: Procediamo per induzione sulla struttura sintattica di P:
  - Se $P = x$ allora $(\hat{\lambda}x\,.\,P)\,Q = S\,K\,K\,Q \longrightarrow_{\mathcal{L}} K\,Q\,(K\,Q) \longrightarrow_{\mathcal{L}} Q = P[^Q/_x]$.
  - Se $x \notin var(P)$ allora $(\hat{\lambda}x\,.\,P)\,Q = K\,P\,Q \longrightarrow_{\mathcal{L}} P = P[^Q/_x]$.
  - Sia $P = P_1\,P_2$ con $x \notin var(P)$ e supponiamo $(\hat{\lambda}x\,.\,P_1)\,Q \longrightarrow_{\mathcal{L}} P_1[^Q/_x]$ e $(\hat{\lambda}x\,.\,P_2)\,Q \longrightarrow_{\mathcal{L}} P_2[^Q/_x]$. Allora $(\hat{\lambda}x\,.\,P)\,Q = S\,(\hat{\lambda}x\,.\,P_1)\,(\hat{\lambda}x\,.\,P_2)\,Q \longrightarrow_{\mathcal{L}} (\hat{\lambda}x\,.\,P_1)\,Q\,((\hat{\lambda}x\,.\,P_2)\,Q) \longrightarrow_{\mathcal{L}}^* P_1[^Q/_x]\,P_2[^Q/_x] = P[^Q/_x]$. ■fltplf_4

3.3 Ricorsione via Punti Fissi e Calcolabilità

- Le funzioni sono anonime nel λ-calcolo. Pertanto una funzione matematica ricorsiva $f = \cdots f \cdots$ può essere espressa in λ-calcolo solo in modo non ricorsivo tramite una funzione di ordine superiore $\lambda f \,.\, \cdots f \cdots$ a cui è poi applicato un cosiddetto combinatore di punto fisso Ξ ottenendo $\Xi\ \lambda f \,.\, \cdots f \cdots$.
- In generale un punto fisso per una funzione $f : A \to A$ è un elemento $a \in A$ tale che $a = f(a)$, cioè è un elemento di A che è invariante rispetto all'applicazione di f, ovvero è una soluzione dell'equazione $x = f(x)$ e quindi è esprimibile in termini di se stesso. Una funzione può non avere punti fissi, come la funzione successore, oppure può ammetterne soltanto uno o più di uno. Ad esempio, la relazione identità $\mathcal{I}_A = \{(a, a) \mid a \in A\}$ è una funzione per la quale tutti gli elementi di A sono punti fissi.
- Nel λ-calcolo l'equazione di punto fisso per un λ-termine E è espressa come $F \equiv_{\beta\eta} E\,F$ e si dice che $\Xi \in \Lambda$ è un combinatore di punto fisso sse $\Xi\,E \equiv_{\beta\eta} E\,(\Xi\,E)$ per ogni $E \in \Lambda$. Un combinatore di punto fisso è dunque un λ-termine che consente di calcolare il punto fisso di ogni λ-termine E semplicemente applicando il primo al secondo. Esistono combinatori di punto fisso in λ-calcolo?
- Teorema (del punto fisso): Per ogni $E \in \Lambda$ esiste $F \in \Lambda$ tale che $F \equiv_{\beta\eta} E\,F$.
- Combinatori di punto fisso che dimostrano il teorema:
  - Combinatore di punto fisso di Turing: $\Theta = (\lambda x \,.\, \lambda y \,.\, y\,(x\,x\,y))\,(\lambda x \,.\, \lambda y \,.\, y\,(x\,x\,y))$. Per ogni $E \in \Lambda$ risulta $\Theta\,E \longrightarrow_\beta (\lambda y \,.\, y\,((\lambda x \,.\, \lambda y \,.\, y\,(x\,x\,y))\,(\lambda x \,.\, \lambda y \,.\, y\,(x\,x\,y))\,y))\,E \longrightarrow_\beta E\,((\lambda x \,.\, \lambda y \,.\, y\,(x\,x\,y))\,(\lambda x \,.\, \lambda y \,.\, y\,(x\,x\,y))\,E) = E\,(\Theta\,E)$.
  - Combinatore di punto fisso di Curry: $Y = \lambda f \,.\, (\lambda x \,.\, f\,(x\,x))\,(\lambda x \,.\, f\,(x\,x))$. Per ogni $E \in \Lambda$ risulta $Y\,E \longrightarrow_\beta (\lambda x \,.\, E\,(x\,x))\,(\lambda x \,.\, E\,(x\,x)) \longrightarrow_\beta E\,((\lambda x \,.\, E\,(x\,x))\,(\lambda x \,.\, E\,(x\,x))) \equiv_\beta E\,(Y\,E)$ perché nella seconda β-riduzione $x \notin \mathit{varlib}(E)$ altrimenti avremmo ridenominato λx in ognuna delle due sostituzioni $(\lambda x \,.\, f\,(x\,x))[{}^{E}/_{f}]$ della prima β-riduzione. Si noti che $Y\,E \not\longrightarrow_\beta^* E\,(Y\,E)$ perché nel passo finale $E\,((\lambda x \,.\, E\,(x\,x))\,(\lambda x \,.\, E\,(x\,x))) \equiv_\beta E\,(Y\,E)$ abbiamo sfruttato l'esito della β-riduzione iniziale $Y\,E \longrightarrow_\beta (\lambda x \,.\, E\,(x\,x))\,(\lambda x \,.\, E\,(x\,x))$, non la definizione di Y.
- Esempi ($\underline{test}_0$ e $\underline{test}_\leq$ realizzano un if-then-else basato su $n = 0$ ed $m \leq n$ rispettivamente):
  - Vediamo la definizione della funzione fattoriale e la sua applicazione a uno specifico valore:
    - Sia $\underline{test}_0 = \lambda n \,.\, \lambda p \,.\, \lambda q \,.\, n\,(\underline{K}\,q)\,p$ con $\underline{K} = \lambda x \,.\, \lambda y \,.\, x$, così $\underline{test}_0\,\underline{0}\,E\,F \longrightarrow_\beta \underline{0}\,(\underline{K}\,F)\,E \longrightarrow_\beta E$ mentre $\underline{test}_0\,\underline{n}\,E\,F \longrightarrow_\beta \underline{n}\,(\underline{K}\,F)\,E \longrightarrow_\beta \underline{n}\,(\lambda y \,.\, F)\,E \longrightarrow_\beta^* F$ se $n > 0$ perché $y \notin \mathit{varlib}(F)$.
    - Sia $\underline{F} = \lambda r \,.\, \lambda n \,.\, \underline{test}_0\,n\,\underline{1}\,(\underline{molt}\,n\,(r\,(\underline{pred}\,n)))$.
    - Allora $\underline{fatt} = \Theta\,\underline{F}$.
    - $\underline{fatt}\,\underline{2} = \Theta\,\underline{F}\,\underline{2} \longrightarrow_\beta^* \underline{F}\,(\Theta\,\underline{F})\,\underline{2} \longrightarrow_\beta^* \underline{test}_0\,\underline{2}\,\underline{1}\,(\underline{molt}\,\underline{2}\,(\Theta\,\underline{F}\,(\underline{pred}\,\underline{2}))) \longrightarrow_\beta^* \underline{molt}\,\underline{2}\,(\Theta\,\underline{F}\,\underline{1}) \longrightarrow_\beta^* \underline{molt}\,\underline{2}\,(\underline{F}\,(\Theta\,\underline{F})\,\underline{1}) \longrightarrow_\beta^* \underline{molt}\,\underline{2}\,(\underline{test}_0\,\underline{1}\,\underline{1}\,(\underline{molt}\,\underline{1}\,(\Theta\,\underline{F}\,(\underline{pred}\,\underline{1})))) \longrightarrow_\beta^* \underline{molt}\,\underline{2}\,(\underline{molt}\,\underline{1}\,(\Theta\,\underline{F}\,\underline{0})) \longrightarrow_\beta^* \underline{molt}\,\underline{2}\,(\underline{molt}\,\underline{1}\,(\underline{F}\,(\Theta\,\underline{F})\,\underline{0})) \longrightarrow_\beta^* \underline{molt}\,\underline{2}\,(\underline{molt}\,\underline{1}\,(\underline{test}_0\,\underline{0}\,\underline{1}\,(\underline{molt}\,\underline{0}\,(\Theta\,\underline{F}\,(\underline{pred}\,\underline{0}))))) \longrightarrow_\beta^* \underline{molt}\,\underline{2}\,(\underline{molt}\,\underline{1}\,\underline{1}) \longrightarrow_\beta^* \underline{molt}\,\underline{2}\,\underline{1} \longrightarrow_\beta^* \underline{2}$.
  - L'idea alla base della codifica del predecessore di $n \in \mathbb{N}_{\geq 1}$ è di generare la sequenza di coppie $(0,0), (0,1), (1,2), \ldots, (n-1, n)$ per poi restituire la prima componente dell'ultima coppia:
    - Sia $\underline{c}_{E_1,E_2} = \langle E_1; E_2\rangle = \lambda z \,.\, z\,E_1\,E_2$ dove $z \notin \mathit{varlib}(E_1) \cup \mathit{varlib}(E_2)$.
    - Sia $\underline{gc} = \lambda c \,.\, \langle c\,\underline{pro}_{2,2};\ \underline{succ}\,(c\,\underline{pro}_{2,2})\rangle$ dove $\underline{pro}_{k,i} = \lambda x_1 \,.\, \ldots \,.\, \lambda x_k \,.\, x_i$ per $k \geq 1$, $1 \leq i \leq k$.
    - Allora $\underline{pred} = \lambda n \,.\, n\,\underline{gc}\,\langle\underline{0}; \underline{0}\rangle\,\underline{pro}_{2,1}$.
  - Di conseguenza possiamo poi definire sottrazione e divisione come segue:
    - Sia $\underline{S} = \lambda r \,.\, \lambda m \,.\, \lambda n \,.\, \underline{test}_0\,n\,m\,(r\,(\underline{pred}\,m)\,(\underline{pred}\,n))$.
    - Allora $\underline{sottr} = \Theta\,\underline{S}$.
    - Sia $\underline{test}_\leq = \Theta\,\underline{T}_\leq$ con $\underline{T}_\leq = \lambda r \,.\, \lambda m \,.\, \lambda n \,.\, \lambda p \,.\, \lambda q \,.\, \underline{test}_0\,m\,p\,(\underline{test}_0\,n\,q\,(r\,(\underline{pred}\,m)\,(\underline{pred}\,n)\,p\,q))$.
    - Sia $\underline{D} = \lambda r \,.\, \lambda m \,.\, \lambda n \,.\, \underline{test}_\leq\,(\underline{succ}\,m)\,n\,\underline{0}\,(\underline{succ}\,(r\,(\underline{sottr}\,m\,n)\,n))$ dove $m{+}1 \leq n$ significa $m < n$.
    - Allora $\underline{div} = \Theta\,\underline{D}$.

- Tra gli anni 1880 e gli anni 1930 vennero pubblicati molti studi dedicati alla ricorsione e al suo uso nell'ambito dei numeri naturali, tra cui citiamo quelli di Dedekind, Peano, Skolem, Hilbert, Ackermann, Peter, Herbrand e Gödel. Nel 1936 Stephen Kleene dimostrò che le funzioni ricorsive generali di Gödel ed Herbrand sono definibili mediante λ-termini. Nel 1937 Alan Turing dimostrò che le funzioni definibili mediante λ-termini sono calcolabili da macchine di Turing. I due risultati insieme stabiliscono che una funzione $f : \mathbb{N} \to \mathbb{N}$ è calcolabile da una macchina di Turing sse f è definibile mediante un λ-termine, ovvero sse f è una funzione ricorsiva generale. Queste sono *tutte* le funzioni calcolabili?
- Chiamiamo metodo effettivo una sequenza finita di istruzioni interpretabili senza ambiguità la cui esecuzione termina sempre in un tempo finito producendo il risultato corretto. Non tutte le funzioni su $\mathbb{N}$ sono calcolabili attraverso un metodo effettivo; poiché $\mathbb{Z}$ e $\mathbb{Q}$ sono equipotenti a $\mathbb{N}$, e quindi tutti gli interi e tutti i razionali sono codificabili come naturali, ciò vale anche per le funzioni su $\mathbb{Z}$ e su $\mathbb{Q}$. Ad esempio, dato un qualsiasi polinomio $p(x_1, x_2, \ldots, x_n)$ a coefficienti interi di grado arbitrario, il decimo problema di Hilbert richiede di stabilire se l'equazione $p(x_1, x_2, \ldots, x_n) = 0$ ammette soluzioni intere. Poichè tale problema è stato dimostrato essere indecidibile, la funzione su $\mathbb{Z}$ che restituisce 1 o 0 a seconda che l'equazione $p(x_1, x_2, \ldots, x_n) = 0$ ammetta soluzioni intere o meno non è calcolabile.
- I risultati precedentemente citati di Kleene e Turing hanno indotto a ritenere che l'insieme delle funzioni calcolabili tramite un qualsiasi metodo effettivo sia equivalentemente caratterizzato dai concetti di Turing-calcolabilità, λ-definibilità e ricorsione generale, come espresso dalle seguenti tesi:
  - Tesi di Church: Le funzioni su $\mathbb{N}$ che sono calcolabili attraverso un qualsiasi metodo effettivo sono esattamente quelle definibili mediante λ-termini.
  - Tesi di Turing: Le funzioni su $\mathbb{N}$ che sono calcolabili attraverso un qualsiasi metodo effettivo sono esattamente quelle calcolabili mediante macchine di Turing.
- Approfondiamo ora il legame tra funzioni ricorsive generali e funzioni definibili mediante λ-termini, evidenziando il ruolo dei combinatori di punto fisso per catturare la ricorsione generale.
- L'insieme delle funzioni ricorsive primitive è composto dalle seguenti funzioni base su $\mathbb{N}$:
  - $0_k(x_1, \ldots, x_k) = 0$, detta funzione zero su $k \geq 0$ argomenti;
  - $succ(x) = x + 1$, detta funzione successore;
  - $pro_{k,i}(x_1, \ldots, x_k) = x_i$, detta funzione proiezione i-esima su $k \geq 1$ argomenti, dove $1 \leq i \leq k$;

  ed è chiuso rispetto alle seguenti operazioni:
  - composizione: se $h(y_1, \ldots, y_m)$ è ricorsiva primitiva e $g_i(x_1, \ldots, x_k)$ è ricorsiva primitiva per ogni $1 \leq i \leq m$, allora anche $f(x_1, \ldots, x_k) = h(g_1(x_1, \ldots, x_k), \ldots, g_m(x_1, \ldots, x_k))$ è ricorsiva primitiva;
  - ricorsione primitiva: se $h(x_1, \ldots, x_m)$ e $g(y, k, x_1, \ldots, x_m)$ sono ricorsive primitive, allora anche $f(k, x_1, \ldots, x_m)$ definita ponendo:
$$\begin{aligned} f(0, x_1, \ldots, x_m) &= h(x_1, \ldots, x_m) \\ f(k+1, x_1, \ldots, x_m) &= g(f(k, x_1, \ldots, x_m), k, x_1, \ldots, x_m) \end{aligned}$$
    è ricorsiva primitiva (h rappresenta il caso base, g quello induttivo).
- Molte funzioni di largo uso sono ricorsive primitive:
  - $1_k(x_1, \ldots, x_k) = succ(0_k(x_1, \ldots, x_k))$,
    $2_k(x_1, \ldots, x_k) = succ(1_k(x_1, \ldots, x_k))$, e così via.
  - $add(0, x) = pro_{1,1}(x)$,
    $add(k+1, x) = g(add(k, x), k, x)$ dove $g(y, k, x) = succ(pro_{3,1}(y, k, x))$.
  - $molt(0, x) = 0_1(x)$,
    $molt(k+1, x) = g(molt(k, x), k, x)$ dove $g(y, k, x) = add(pro_{3,1}(y, k, x), pro_{3,3}(y, k, x))$.
  - $esp(0, x) = 1_1(x)$,
    $esp(k+1, x) = g(esp(k, x), k, x)$ dove $g(y, k, x) = molt(pro_{3,1}(y, k, x), pro_{3,3}(y, k, x))$.
  - $fatt(0) = 1_0$,
    $fatt(k{+}1) = g(fatt(k), k)$ dove $g(y, k) = molt(pro_{2,1}(y, k), g'(y, k))$ con $g'(y, k) = succ(pro_{2,2}(y, k))$.

- Una funzione $f(k, x_1, \ldots, x_m)$ ottenuta per ricorsione primitiva da $h(x_1, \ldots, x_m)$ e $g(y, k, x_1, \ldots, x_m)$ non è intrinsecamente ricorsiva perché può essere implementata in modo iterativo. In particolare, il valore che f assume in $k \in \mathbb{N}$ dati i valori di $x_1, \ldots, x_m$ coincide col valore f' calcolato come segue:
  - Inizializzare f' ponendo $f' = h(x_1, \ldots, x_m)$.
  - Inizializzare f'' ponendo $f'' = g(f', 0, x_1, \ldots, x_m)$.
  - Per ogni i da 1 a k ripetere:
    * Aggiornare f' ponendo $f' = f''$.
    * Aggiornare f'' ponendo $f'' = g(f', i, x_1, \ldots, x_m)$.

- È quindi ragionevole aspettarsi che non tutte le funzioni ricorsive siano esprimibili come funzioni ricorsive primitive. Un controesempio è dato dalla funzione di Ackermann:
$$\begin{aligned} A(0,0,y) &= y \\ A(0,x+1,y) &= A(0,x,y)+1 \\ A(1,0,y) &= 0 \\ A(k+2,0,y) &= 1 \\ A(k+1,x+1,y) &= A(k, A(k+1,x,y), y) \end{aligned}$$
detta esponenziale generalizzata perché cresce più rapidamente di qualsiasi funzione ricorsiva primitiva:
$$\begin{aligned} A(0,x,y) &= x+y \\ A(1,x,y) &= x \cdot y \\ A(2,x,y) &= y^x \\ A(3,x,y) &= y^{y^{\cdot^{\cdot^{y}}}} \text{ dove il numero di esponenti } y \text{ è pari a } x \end{aligned}$$

- Una funzione $f(x_1, \ldots, x_m)$ è ricorsiva generale sse viene ottenuta per minimizzazione da una funzione ricorsiva primitiva $h(k, x_1, \ldots, x_m)$, cioè $f(x_1, \ldots, x_m) = \min\{k \in \mathbb{N} \mid h(k, x_1, \ldots, x_m) = 0\}$.

- Questi sono i λ-termini che codificano le funzioni ricorsive generali, nel senso che se $f(n_1, \ldots, n_k) = m$ allora $\underline{f}\,\underline{n}_1 \ldots \underline{n}_k \longrightarrow_\beta^* \underline{m}$ per ogni funzione ricorsiva generale f e per ogni $k, n_1, \ldots, n_k, m \in \mathbb{N}$:
  - Funzioni base:
    * $\underline{0}_k = \lambda x_1 . \ldots . \lambda x_k . \lambda s . \lambda z . z$.
    * $\underline{succ} = \lambda n . \lambda x . \lambda y . x\,(n\,x\,y)$.
    * $\underline{pro}_{k,i} = \lambda x_1 . \ldots . \lambda x_k . x_i$.
  - Composizione:
    * $\underline{f} = \lambda x_1 . \ldots . \lambda x_k . \underline{h}\,(\underline{g}_1 x_1 . \ldots . x_k) \ldots (\underline{g}_m x_1 . \ldots . x_k)$.
  - Ricorsione primitiva:
    * L'idea della codifica è di generare come nella versione iterativa la sequenza di $k+1$ triple:
$$\begin{aligned} (0;\, h(x_1,\ldots,x_m);\, g(h(x_1,\ldots,x_m),0,x_1,\ldots,x_m)) &= (0;\, f(0,x_1,\ldots,x_m);\, f(1,x_1,\ldots,x_m)) \\ (1;\, f(1,x_1,\ldots,x_m);\, g(f(1,x_1,\ldots,x_m),1,x_1,\ldots,x_m)) &= (1;\, f(1,x_1,\ldots,x_m);\, f(2,x_1,\ldots,x_m)) \\ &\ldots = \ldots \\ (k;\, f(k,x_1,\ldots,x_m);\, g(f(k,x_1,\ldots,x_m),k,x_1,\ldots,x_m)) &= (k;\, f(k,x_1,\ldots,x_m);\, f(k+1,x_1,\ldots,x_m)) \end{aligned}$$
      per poi restituire la seconda componente dell'ultima tripla generata.
    * Sia $\underline{t}_{E_1,E_2,E_3} = \langle E_1; E_2; E_3\rangle = \lambda z . z\,E_1\,E_2\,E_3$ dove $z \notin \mathit{varlib}(E_1) \cup \mathit{varlib}(E_2) \cup \mathit{varlib}(E_3)$.
    * Notiamo che $\underline{t}_{E_1,E_2,E_3}\,\underline{pro}_{3,1} \longrightarrow_\beta^* E_1$, $\underline{t}_{E_1,E_2,E_3}\,\underline{pro}_{3,2} \longrightarrow_\beta^* E_2$, $\underline{t}_{E_1,E_2,E_3}\,\underline{pro}_{3,3} \longrightarrow_\beta^* E_3$.
    * Sia $\underline{gt} = \lambda t . \langle \underline{succ}\,(t\,\underline{pro}_{3,1});\, t\,\underline{pro}_{3,3};\, \underline{g}\,(t\,\underline{pro}_{3,3})\,(\underline{succ}\,(t\,\underline{pro}_{3,1}))\,x_1\,\ldots\,x_m\rangle$.
    * Allora $\underline{f} = \lambda k . \lambda x_1 . \ldots . \lambda x_m . k\,\underline{gt}\,\langle \underline{0};\, \underline{h}\,x_1\,\ldots\,x_m;\, \underline{g}\,(\underline{h}\,x_1\,\ldots\,x_m)\,\underline{0}\,x_1\,\ldots\,x_m\rangle\,\underline{pro}_{3,2}$.
  - Ricorsione generale:
    * L'idea della codifica è di generare la sequenza $h(0, x_1, \ldots, x_m)$, $h(1, x_1, \ldots, x_m)$, e così via fino a incontrare il primo valore pari a 0 nella sequenza se esiste. La funzione che genera tale sequenza è il punto fisso di un'opportuna funzione di ordine superiore.
    * Sia $\underline{H} = \lambda r . \lambda k . \lambda x_1 . \ldots . \lambda x_m . \underline{test}_0\,(\underline{h}\,k\,x_1\,\ldots\,x_m)\,k\,(r\,(\underline{succ}\,k)\,x_1\,\ldots\,x_m)$.
    * Allora $\underline{f} = \lambda x_1 . \ldots . \lambda x_m . \Theta\,\underline{H}\,\underline{0}\,x_1\,\ldots\,x_m$. ■fltplf_5

3.4 Terminazione e Confluenza nel Lambda Calcolo

- L'assenza di radicali in un λ-termine fornisce una chiara evidenza della finalità del termine stesso dal punto di vista computazionale, così come il numero ottenuto da un'espressione aritmetica dopo aver calcolato tutte le sue operazioni ne rappresenta il valore finale. La procedura di β-riduzione del λ-calcolo non è però terminante, perché la sua applicazione non fa necessariamente diminuire il numero di radicali. Basta prendere $\underline{\omega} = \lambda x \,.\, x\, x$ dato che $\underline{\Omega} = \underline{\omega}\,\underline{\omega} = (\lambda x \,.\, x\, x)\,\underline{\omega} \longrightarrow_\beta \underline{\omega}\,\underline{\omega} \longrightarrow_\beta \ldots \longrightarrow_\beta \underline{\omega}\,\underline{\omega} \longrightarrow_\beta \ldots$

- Un λ-termine E è in forma normale sse non contiene radicali, cioè sse la β-riduzione non è applicabile ad esso, ovvero sse è generato dalla sintassi $N ::= x \mid \lambda x \,.\, N \mid x\, F$ dove $F ::= x \mid \lambda x \,.\, N \mid F\, F$. Diciamo poi che $E \in \Lambda$ ammette forma normale sse esiste $E' \in \Lambda$ in forma normale tale che $E \longrightarrow_\beta^* E'$. Non tutti i λ-termini ammettono forma normale, come dimostra l'autoapplicazione $\underline{\Omega}$.

- La procedura di β-riduzione del λ-calcolo non è deterministica, nel senso che non è definita a priori una strategia di riduzione che, in presenza di più radicali in un termine, stabilisca a quale radicale applicare la β-riduzione per primo. In via di principio non è pertanto scontato che si riesca a raggiungere la forma normale per un termine che la ammette, né che la forma normale sia unica quando esiste.

- In presenza di più radicali, si può scegliere tra ridurre quelli più esterni o quelli più interni rispetto alla struttura sintattica del termine. Per esempio, in $(\lambda x \,.\, E)\, F$ si può applicare la β-riduzione all'intero termine prima oppure dopo averla applicata dentro E e dentro F.

- Se ci sono più radicali allo stesso livello nella struttura sintattica del termine, si può scegliere tra ridurre quello più a sinistra oppure quello più a destra. Per esempio, in $E_1\, E_2$ dove E_1 non è una λ-astrazione, si può applicare la β-riduzione prima dentro E_1 oppure prima dentro E_2.

- Strategie di riduzione comunemente utilizzate:

  - Chiamata per nome: viene sistematicamente ridotto il radicale più a sinistra tra quelli più esterni nella struttura sintattica del termine, che corrisponde a passare gli argomenti alla funzione senza valutarli. Ad esempio, $\underline{\omega}\,(\underline{I}\,\underline{I}) = (\lambda x \,.\, x\, x)\,(\underline{I}\,\underline{I}) \longrightarrow_\beta (\underline{I}\,\underline{I})\,(\underline{I}\,\underline{I}) \longrightarrow_\beta \underline{I}\,(\underline{I}\,\underline{I}) \longrightarrow_\beta \underline{I}\,\underline{I} \longrightarrow_\beta \underline{I}$.
  - Chiamata per valore: viene sistematicamente ridotto il radicale più a sinistra tra quelli più interni nella struttura sintattica del termine, che corrisponde a valutare gli argomenti prima di passarli alla funzione. Ad esempio, $\underline{\omega}\,(\underline{I}\,\underline{I}) = \underline{\omega}\,((\lambda x \,.\, x)\,\underline{I}) \longrightarrow_\beta \underline{\omega}\,\underline{I} \longrightarrow_\beta \underline{I}\,\underline{I} \longrightarrow_\beta \underline{I}$.

- Quando la forma normale esiste, la sua unicità può essere garantita dalla proprietà di confluenza, la quale può essere formulata in modi diversi:

  - Proprietà di confluenza forte o del diamante: per ogni $E \in \Lambda$, se $E \longrightarrow_\beta E_1$ ed $E \longrightarrow_\beta E_2$, allora esiste $E' \in \Lambda$ tale che $E_1 \longrightarrow_\beta E'$ ed $E_2 \longrightarrow_\beta E'$.
  - Proprietà di confluenza debole o di Church-Rosser: per ogni $E \in \Lambda$, se $E \longrightarrow_\beta^* E_1$ ed $E \longrightarrow_\beta^* E_2$, allora esiste $E' \in \Lambda$ tale che $E_1 \longrightarrow_\beta^* E'$ ed $E_2 \longrightarrow_\beta^* E'$.

  La proprietà di confluenza forte implica quella debole, ma la prima non vale nel λ-calcolo come si vede dall'applicazione delle due strategie di riduzione a $\underline{\omega}\,(\underline{I}\,\underline{I})$.

- Lemma (strip lemma): Per ogni $E \in \Lambda$, se $E \longrightarrow_\beta E_1$ ed $E \longrightarrow_\beta^* E_2$, allora esiste $E' \in \Lambda$ tale che $E_1 \longrightarrow_\beta^* E'$ ed $E_2 \longrightarrow_\beta E'$.

- Teorema (di Church-Rosser): Il λ-calcolo gode della proprietà di confluenza debole.

- Corollario: Se $E \in \Lambda$ ammette forma normale, allora questa è unica a meno di α-conversione.

- Corollario: La teoria della β-equivalenza è coerente, cioè non vale che $E_1 \equiv_\beta E_2$ per ogni $E_1, E_2 \in \Lambda$.

- Curry dimostrò che, quando esiste, la forma normale viene sempre raggiunta con la strategia di chiamata per nome. Ciò non vale per la strategia di chiamata per valore. Per esempio, il termine $(\lambda y \,.\, z)\,(\underline{\omega}\,\underline{\omega})$ β-riduce alla forma normale z con la prima strategia, mentre β-riduce a se stesso con la seconda.

- Di contro, quando conduce alla forma normale, la strategia di chiamata per valore potrebbe risultare più efficiente della strategia di chiamata per nome. Per esempio, il termine $(\lambda n \,.\, \underline{\mathit{add}}\, n\, n)\,(\underline{\mathit{molt}}\,\underline{5}\,\underline{4})$ β-riduce più rapidamente a $\underline{40}$ con la prima perché la seconda calcola due volte $\underline{\mathit{molt}}\,\underline{5}\,\underline{4}$.

- Per coniugare la raggiungibilità della forma normale garantita dalla strategia di chiamata per nome con la maggiore efficienza della strategia di chiamata per valore, conviene considerare sistematicamente solo i radicali al primo livello. Ciò conduce alla cosiddetta forma normale di testa debole, la quale è una forma normale di testa oppure una λ-astrazione.

- Il formato di una forma normale di testa è $\lambda x_1 \,.\, \ldots \,.\, \lambda x_n \,.\, y\, E_1\, \ldots\, E_k$ – dove $n, k \in \mathbb{N}$ e la variabile di testa y può essere diversa da $x_1, \ldots, x_n$ – in cui le sole β-riduzioni possibili sono interne ai singoli sottotermini E_i data la presenza della variabile y all'inizio e l'associatività da sinistra dell'applicazione. Il termine più semplice in questo formato è y.

- La forma normale di testa può essere vista come un'approssimazione della forma normale. Ciò deriva dal fatto che un termine in forma normale è anche in forma normale di testa (perché se non ha radicali allora in particolare non ne ha all'inizio) e dalla seguente proprietà di risolvibilità: $E \in \Lambda$ ha forma normale di testa sse esiste $h \in \mathbb{N}$ tale che $E\, E_1\, \ldots\, E_h \equiv_\beta \underline{I}$, ossia $E\, E_1\, \ldots\, E_h\, F \equiv_\beta F$ per ogni $F \in \Lambda$.

- Se un λ-termine non è in forma normale di testa allora è nel formato $\lambda x_1 \,.\, \ldots \,.\, \lambda x_n \,.\, (\lambda y \,.\, E_0)\, E_1\, \ldots\, E_k$ con $k \geq 1$, dove $(\lambda y \,.\, E_0)\, E_1$ è detto radicale di testa ed è il termine più semplice in questo formato. Quando $n \geq 1$, cioè il radicale di testa è preceduto da almeno un legatore, tale formato è un esempio di λ-astrazione che rappresenta una forma normale di testa debole che non è in forma normale di testa, nel qual caso $(\lambda y \,.\, E_0)\, E_1\, \ldots\, E_k$ costituisce il prossimo livello al quale applicare la β-riduzione.

- In alternativa, si può adattare la strategia di chiamata per nome ragionando in termini di riscrittura di grafi anziché riscrittura di termini come scoperto nel 1971 da Christopher Wadsworth. Precisamente, se si considera l'albero di sintassi astratta di un λ-termine, è possibile individuare eventuali sottotermini ripetuti e rimpiazzare i rispettivi sottoalberi con un unico sottoalbero condiviso da tutti gli archi entranti nelle radici di quei sottoalberi – rendendo l'albero originario un grafo – in modo da valutare quei sottotermini una sola volta.

- La strategia risultante viene detta strategia di chiamata per necessità o valutazione pigra, in quanto la valutazione degli argomenti di una funzione viene ritardata il più possibile, e memorizza i valori degli argomenti per usi successivi, così da non doverli ricalcolare. Ad esempio, in $\lambda n \,.\, \underline{\mathit{add}}\, n\, n$ si avrebbe un unico sottoalbero condiviso per n e quindi $\underline{\mathit{molt}}\,\underline{5}\,\underline{4}$ verrebbe calcolato una sola volta.

3.5 Lambda Calcolo con Tipi

- I tipi vennero introdotti tra il 1910 e il 1913 da Russell e Whitehead nel loro libro "Principia Mathematica" per evitare i paradossi logici. I tipi permettono di classificare i termini, cioè di individuare insiemi di termini con proprietà computazionali simili, così da prevenire comportamenti indesiderati come ad esempio la non terminazione. Essi sono necessari anche in λ-calcolo (e in logica combinatoria) per evitare alcuni paradossi, come quello scoperto da Kleene e Rosser il quale essenzialmente replica il paradosso di Richard in teoria dei linguaggi formali e poi venne riformulato da Church.

- Consideriamo un insieme $\mathbb{T}$ formato da un insieme numerabile di simboli di tipo – i quali verranno indicati con le lettere τ, σ, ρ – e chiuso rispetto al tipo funzione $\tau \to \sigma$, dove τ rappresenta il tipo dell'argomento e σ il tipo del risultato. Per evitare l'uso sistematico delle parentesi assumiamo che l'operatore $\to$ sui tipi sia associativo da destra come la λ-astrazione per coerenza col currying.

- Nel λ-calcolo esistono due diversi generi di sistemi di tipi:
  - I sistemi di tipi alla Church stabiliscono che ogni λ-termine venga definito assieme al suo tipo cosicché sintassi e semantica includono il controllo dei tipi.
  - I sistemi di tipi alla Curry stabiliscono che il tipo di ogni λ-termine sia attribuito mediante regole formali che inferiscono il tipo dal formato del termine.
- L'insieme Λ' dei λ-termini con tipi alla Church è il più piccolo linguaggio su $\mathit{Var} \cup \{\lambda, .,(,)\} \cup \mathbb{T}$ tale che:
  - $x^\tau \in \Lambda'$ per ogni $x \in \mathit{Var}$ e $\tau \in \mathbb{T}$.
  - $(\lambda x^\tau . E^\sigma)^{\tau\to\sigma} \in \Lambda'$ per ogni $x^\tau, E^\sigma \in \Lambda'$.
  - $(E_1^{\tau\to\sigma}\, E_2^\tau)^\sigma \in \Lambda'$ per ogni $E_1^{\tau\to\sigma}, E_2^\tau \in \Lambda'$.
- Le regole semantiche su Λ' si modificano di conseguenza includendo il controllo dei tipi come segue:
  - α-conversione con tipi: $\lambda x^\tau . E^\sigma =_\alpha \lambda y^\tau . (E^\sigma[{}^{y^\tau}/_{x^\tau}])$ purché $y^\tau \notin \mathit{varlib}(E^\sigma)$.
  - β-conversione con tipi: $(\lambda x^\tau . E^\sigma)\, F^\tau =_\beta E^\sigma[{}^{F^\tau}/_{x^\tau}]$.
  - η-conversione con tipi: $\lambda x^\tau . E^\sigma\, x^\tau =_\eta E^\sigma$ purché $x^\tau \notin \mathit{varlib}(E^\sigma)$.
- Nel caso dei sistemi di tipi alla Curry, sintassi e semantica del λ-calcolo non cambiano, ma si aggiunge un sistema di attribuzione dei tipi. Data una base $\Gamma = \{x_i : \tau_i \mid x_i \in \mathit{Var}, \tau_i \in \mathbb{T}, 1 \le i \le n, x_i \neq x_j \text{ se } i \neq j\}$ di dichiarazioni di tipo per le variabili, il sistema consiste nelle seguenti tre regole di inferenza dei tipi dove $\Gamma \vdash E : \tau$ significa che, partendo da Γ, al λ-termine E viene attribuito il tipo τ:

$$\frac{x : \tau \in \Gamma}{\Gamma \vdash x : \tau} \qquad \frac{\Gamma \cup \{x : \tau\} \vdash E : \sigma}{\Gamma \vdash (\lambda x\,.\,E) : \tau \to \sigma} \qquad \frac{\Gamma \vdash E_1 : \tau \to \sigma \quad \Gamma \vdash E_2 : \tau}{\Gamma \vdash (E_1\, E_2) : \sigma}$$

- Teorema: $E^\tau \in \Lambda'$ sse $\{x_i : \tau_i \mid x_i^{\tau_i} \in \mathit{varlib}(E^\tau)\} \vdash |E^\tau| : \tau$, dove $|E^\tau|$ è il λ-termine ottenuto da E^τ eliminando tutti i tipi presenti in esso.
- Ogni termine in Λ' è fortemente normalizzabile, cioè qualsiasi strategia di riduzione applicata ad esso termina e produce la sua forma normale. Di conseguenza $\Lambda' \subsetneq \Lambda$, ossia non tutti i λ-termini possono essere estesi coi tipi. Per esempio, la funzione identità è definibile in Λ' come $\lambda x^\tau . x^\tau$, mentre $\underline{\omega} = \lambda x\,.\,x\,x$ non lo è perché x dovrebbe essere contemporaneamente di tipo $\tau \to \sigma$ e di tipo τ.
- Nemmeno i combinatori di punto fisso Θ e Y sono definibili in Λ'. Tuttavia, poiché sussiste la necessità di implementare la ricorsione, Λ' viene esteso con delle costanti – così da poter trattare anche i numeri senza dover ricorrere ai numerali di Church – tra le quali F_τ di tipo $(\tau \to \tau) \to \tau$ per ogni $\tau \in \mathbb{T}$, assieme alla seguente ulteriore regola di punto fisso:
  - δ-conversione: $F_\tau\, E^{\tau\to\tau} =_\delta E^{\tau\to\tau}\, (F_\tau\, E^{\tau\to\tau})$.
- Esempi di funzioni definibili in Λ':
  - $\underline{I} = \lambda x\,.\,x$ è definibile in Λ' con tipo $\tau \to \tau$ usando x^τ.
  - $\underline{K} = \lambda x\,.\,\lambda y\,.\,x$ è definibile in Λ' con tipo $\tau \to \sigma \to \tau$ usando x^τ e y^σ.
  - $\underline{S} = \lambda x\,.\,\lambda y\,.\,\lambda z\,.\,x\,z\,(y\,z)$ è definibile in Λ' con tipo $(\tau \to \sigma \to \rho) \to (\tau \to \sigma) \to \tau \to \rho$ usando $x^{\tau\to\sigma\to\rho}$, $y^{\tau\to\sigma}$, z^τ.
  - $\underline{n} = \lambda s\,.\,\lambda z\,.\,s\,(\ldots(s\,z)\ldots)$ è definibile in Λ' con tipo $\nu = (\tau \to \tau) \to \tau \to \tau$ usando $s^{\tau\to\tau}$ e z^τ.
  - $\underline{succ} = \lambda n\,.\,\lambda x\,.\,\lambda y\,.\,x\,(n\,x\,y)$ è definibile in Λ' con tipo $\nu \to \nu$ usando n^ν, $x^{\tau\to\tau}$, y^τ.
  - $\underline{add} = \lambda m\,.\,\lambda n\,.\,\lambda x\,.\,\lambda y\,.\,m\,x\,(n\,x\,y)$ e $\underline{molt} = \lambda m\,.\,\lambda n\,.\,\lambda x\,.\,m\,(n\,x)$ sono entrambe definibili in Λ' con tipo $\nu \to \nu \to \nu$ usando m^ν, n^ν, $x^{\tau\to\tau}$, y^τ.
  - $\underline{esp} = \lambda m\,.\,\lambda n\,.\,m\,n$ è definibile in Λ' con tipo $(\tau \to \sigma) \to \tau \to \sigma$ usando $m^{\tau\to\sigma}$ ed n^τ.

- Le regole di inferenza dei tipi alla Curry sono interessanti perché, oltre a coincidere con le regole di inferenza per il connettivo di implicazione della logica intuizionista di Heyting (in cui non vale il principio del terzo escluso) se si omettono i λ-termini (notare che la terza regola è il modus ponens), esse danno luogo al cosiddetto isomorfismo di Curry-Howard o analogia formule-tipi secondo cui:
  - I tipi corrispondono alle formule. Notiamo che tutti i tipi negli esempi di cui sopra sono tautologie, in particolare i tipi di $\underline{K}$ ed $\underline{S}$ sono assiomi usati nei sistemi deduttivi alla Hilbert (v. Sez. 5.5).
  - I λ-termini di quei tipi corrispondono alle dimostrazioni della validità di quelle formule.
  - La β-riduzione corrisponde alla composizione delle dimostrazioni basata sulla regola del taglio, perché se da un lato si deriva $\dfrac{\Gamma \cup \{x : \tau\} \vdash E : \sigma}{\Gamma \vdash (\lambda x \,.\, E) : \tau \to \sigma}$ e dall'altro $\Gamma \vdash F : \tau$ allora $\Gamma \vdash (\lambda x \,.\, E)\, F : \sigma$. ■fltplf_6

Capitolo 4

Programmazione Funzionale: Il Linguaggio Haskell

4.1 Dal Lambda Calcolo alla Programmazione Funzionale

- La programmazione funzionale fa parte del paradigma dichiarativo, radicalmente diverso dal paradigma imperativo originariamente sviluppato come mezzo per determinare il comportamento delle macchine programmabili riflettendone la struttura. In quest'ultimo paradigma il contenuto della memoria – divenuto fondamentale a seguito dell'idea di computer a programma memorizzato di Von Neumann – costituisce implicitamente lo stato della computazione, che quindi può essere modificato attraverso specifiche istruzioni di assegnamento alle variabili, da eseguire in una certa sequenza.
- In programmazione funzionale la computazione ha invece luogo attraverso la valutazione di espressioni basate su funzioni. I programmi risultanti sono in genere più concisi dei corrispondenti programmi imperativi e, soprattutto, sono rappresentati a un livello di astrazione più elevato cosicché sono più leggibili e meglio si prestano a essere analizzati con metodi formali. I valori delle variabili non cambiano più una volta che sono stati attribuiti (quindi non è possibile modificare ripetutamente il valore di una variabile) e le valutazioni di espressioni basate su funzioni non possono dar luogo a effetti collaterali al di fuori delle espressioni stesse (come la modifica di variabili globali o parametri passati per indirizzo).
- La programmazione funzionale raggiunge pertanto la trasparenza referenziale che sta alla base del ragionamento equazionale: eguali possono essere sostituiti a eguali. Poiché viene restituito sempre lo stesso risultato ogni volta che un'espressione basata su funzioni viene valutata sugli stessi argomenti, il valore di un'espressione complessa calcolato su determinati argomenti può essere memorizzato per usi successivi in modo da non doverlo ricalcolare.
- Il λ-calcolo è il prototipo di linguaggio di programmazione funzionale, cioè di linguaggio nel quale le funzioni sono entità di prima classe dotate di appositi tipi e possono essere ricorsive (cioè richiamare se stesse), polimorfe (cioè essere applicate ad argomenti di tipi diversi e/o restituire risultati di tipi diversi) e di ordine superiore (cioè avere funzioni come argomenti e/o restituire funzioni come risultati).
- Il primo linguaggio di programmazione funzionale è stato Lisp (List Processor), sviluppato negli anni 1950 da John McCarthy come linguaggio algebrico per la manipolazione di liste da usare per elaborazioni simboliche in intelligenza artificiale. Tra le sue caratteristiche principali citiamo l'uso di espressioni condizionali per descrivere agevolmente funzioni ricorsive, l'impiego di funzioni di ordine superiore sulle liste e la gestione dinamica delle liste completamente automatica e trasparente al programmatore sia in fase di allocazione che di disallocazione (garbage collection), oltre alla possibilità di rappresentare sia dati che programmi come espressioni. Il λ-calcolo ebbe un impatto minimale sul Lisp, che comprende anche costrutti imperativi come le istruzioni di assegnamento. Una maggiore purezza si trova nel suo principale dialetto, Scheme, sviluppato negli anni 1970 da Steele e Sussman. Esso ammette chiamate ricorsive solo in coda alla definizione delle funzioni (per motivi di efficienza) e la visibilità delle variabili solo all'interno delle funzioni in cui sono dichiarate (e non anche in quelle invocate da tali funzioni).

- Negli anni 1960 Peter Landin, dopo aver formalizzato la semantica di Algol nel λ-calcolo, sviluppò il linguaggio Iswim (If you See What I Mean) ispirandosi al λ-calcolo esteso con costanti. Sul piano sintattico, Iswim abbandona la notazione prefissa in favore di quella infissa, evita l'uso pesante delle parentesi tonde di Lisp e introduce altresì i costrutti `let-in` e `where`, per dichiarare identificatori all'interno di definizioni di funzioni, come pure l'uso dell'indentazione, al posto di terminatori o separatori quali virgola e punto e virgola. Sul piano semantico, Iswim si fonda sull'idea di costruire linguaggi sintatticamente ricchi sulla base di linguaggi sintatticamente parsimoniosi ma sufficientemente espressivi, sul ragionamento equazionale e sul concetto di macchina astratta su cui eseguire i programmi.
- Una spinta fondamentale alla diffusione della programmazione funzionale venne data negli anni 1970 da John Backus nel suo famoso articolo intitolato "Can Programming Be Liberated from the Von Neumann Style? A Functional Style and Its Algebra of Programs". Backus, sviluppatore del Fortran e tra i principali contributori di Algol, non solo esaltò i pregi della programmazione funzionale, ma soprattutto evidenziò le limitazioni della programmazione imperativa, ancora troppo vicina all'architettura della macchina fisica. Rispetto al λ-calcolo, osservò come la sua notevole espressività conduce a una molteplicità di combinatori invece di concentrarsi su quei pochi che risultano adeguati nella maggior parte dei casi. Ispirandosi al linguaggio APL (A Programming Language) sviluppato negli anni 1960 da Kenneth Iverson per la programmazione algebrica su array, Backus propose FP (Functional Programming) come linguaggio algebrico basato su un piccolo insieme di combinatori dai nomi brevi che rendono i programmi particolarmente compatti.
- Negli anni 1970 Robin Milner sviluppò ML (MetaLanguage), nato originariamente come linguaggio di comandi per interagire con il dimostratore automatico di teoremi LCF. Esso include funzioni di ordine superiore, strutturazione a moduli e gestione delle eccezioni, nonché il pattern matching per gli argomenti delle funzioni e la possibilità di definire tipi di dati, come nel linguaggio Hope sviluppato in quegli stessi anni da Rodney Burstall. L'aspetto più significativo di ML è il suo ricco sistema di tipi, che supporta il polimorfismo per funzioni e strutture dati ed è in grado di inferire automaticamente il tipo di ogni espressione senza richiedere dichiarazioni esplicite di tipo. ML presenta anche aspetti imperativi come riferimenti a celle di memoria e relative istruzioni di assegnamento, oltre a istruzioni di ripetizione e di input/output.
- All'inizio degli anni 1980 David Turner sviluppò i linguaggi SASL (Saint Andrews Static Language), KRC (Kent Recursive Calculator) e Miranda in quest'ordine, dove ognuno costituisce un'evoluzione del precedente. Essi mettono a disposizione equazioni ricorsive come zucchero sintattico per il λ-calcolo, apposite espressioni per selezionare agevolmente elementi da liste, funzioni di ordine superiore basate sul currying e la valutazione pigra degli argomenti delle funzioni.

4.2 Haskell: Assemblaggio di Caratteristiche Funzionali

- Il linguaggio Haskell, così denominato in onore di Haskell Curry, venne progettato verso la fine degli anni 1980 da un comitato accademico europeo e americano per mettere a fattor comune gli aspetti più utili e innovativi derivanti dalla proliferazione di linguaggi funzionali più o meno puri durante quei decenni. La sua principale implementazione è il Glasgow Haskell Compiler di Peyton-Jones e Marlow.
- Diversamente dalla programmazione imperativa, in Haskell:
  - I programmi non sono istruzioni di assegnamento da eseguire nell'ordine stabilito dalle istruzioni di controllo del flusso, ma collezioni di funzioni da valutare.
  - Le esecuzioni dei programmi non sono sequenze di passi che modificano il contenuto della memoria, ma creano legami immutabili per le variabili e restituiscono i risultati della valutazione di funzioni.
  - Non ci sono effetti collaterali in quanto la valutazione di una funzione non può modificare variabili al di fuori del suo ambiente locale.
  - La conseguente trasparenza referenziale fa sì che venga restituito sempre lo stesso risultato ogni volta che una funzione viene valutata sugli stessi argomenti.
  - La ricorsione è il principale strumento linguistico per rappresentare l'iterazione e il tipo di dato ricorsivo lista è il principale tipo di dato strutturato disponibile, con la gestione dinamica della memoria per le liste completamente automatizzata.

- L'innalzamento del livello di astrazione verso l'uso di espressioni matematico-logiche al posto di sequenze di istruzioni è la caratteristica comune alle fasi dell'evoluzione della programmazione, che ha avuto come passaggi i linguaggi macchina e assemblativi, i primi linguaggi di programmazione imperativi, la programmazione strutturata e infine i linguaggi di programmazione dichiarativi come quelli funzionali. Ecco come viene programmato un ciclo nelle diverse fasi evolutive:

```
/* assemblativo */              /* strutturato */  /* funzionale */
       x := val_iniz;           x := val_iniz;     x = ciclo fun val_iniz 0 10
       i := 0;                  i := 0;              where
ciclo: x := fun(x, i);          while (i < 10)         ciclo f v i1 i2 = if (i1 >= i2)
       i := i + 1;                x := fun(x, i);        then v
       if (i < 10) goto ciclo;    i := i + 1;            else ciclo f (f v i1) (i1 + 1) i2
```

- Queste sono le principali caratteristiche del linguaggio funzionale puro Haskell:

  - Sintassi equazionale: oltre che come λ-astrazioni, le funzioni possono essere espresse tramite una notazione algebrica che riduce al minimo l'uso delle parentesi e supporta in maniera naturale la ricorsione senza dover utilizzare operatori di punto fisso.
  - Stile dichiarativo: anziché mediante passi sequenziali, ogni funzione viene definita distinguendo tra molteplici casi sulla base di apposite guardie e posticipando la definizione di eventuali identificatori presenti nei vari casi ad apposite sezioni locali le cui linee sono indentate e senza terminatori.
  - Pattern matching: la possibilità di usare modelli per esprimere gli argomenti formali di una funzione semplifica i casi nella definizione della funzione, poi a tempo d'esecuzione gli argomenti effettivi vengono confrontati con tali modelli per determinare quale caso si applica.
  - Funzioni di ordine superiore: viene raggiunto un elevato livello di astrazione e di riuso del software grazie al fatto che una funzione da un lato può accettare funzioni come argomenti e dall'altro può restituire funzioni come risultati.
  - Valutazione pigra: gli argomenti effettivi vengono passati alle funzioni in base alla chiamata per necessità, quindi la loro valutazione viene ritardata il più possibile, i loro valori possono essere memorizzati per usi futuri ed è possibile gestire strutture dati illimitate.
  - Tipi e classi di tipi: oltre ai tipi scalari numerici, logici e carattere e ai tipi strutturati lista, stringa e tupla, sono presenti meccanismi per raccogliere i tipi in classi di tipi e costruire tipi enumerati, tipi unione e tipi ricorsivi assieme a eventuali operazioni su di essi.
  - Sistema di tipi forte ed espressivo: il sistema di tipi applica un numero significativo di controlli alle espressioni che compongono il programma, dando pertanto una certa garanzia di correttezza, ed è espressivo al punto da ammettere predicati sui tipi.
  - Inferenza di tipi: sebbene sia possibile accompagnare la definizione di ogni identificatore con la sua dichiarazione di tipo, quest'ultima non è obbligatoria in quanto il sistema di tipi è in grado di dedurre il tipo di qualunque espressione.
  - Polimorfismo di funzioni e strutture dati e overloading di operatori: una singola definizione per una funzione può trattare argomenti e risultati di tipi generici, così come una struttura può contenere dati di un tipo arbitrario e certi operatori sono applicabili a espressioni di tipi della stessa classe.
  - Gestione automatica della memoria: non ci sono puntatori né direttive esplicite per allocare o disallocare la memoria dinamicamente, in quanto la memoria viene automaticamente allocata quando serve e poi rilasciata quando non è più necessaria (garbage collection).
  - Sistema di input/output e strutturazione a moduli: queste due caratteristiche sono importanti affinché il linguaggio sia utilizzato in pratica, dove la prima è implementata in maniera funzionale mentre la seconda consente lo sviluppo di progetti software di grandi dimensioni.

- Esempio: Consideriamo il seguente programma Haskell per l'algoritmo di ordinamento quicksort:

```
quicksort :: (Ord a) => [a] -> [a]
quicksort []       = []
quicksort (x : xs) = quicksort xs_inf ++ [x] ++ quicksort xs_sup
  where
    xs_inf = filter (<= x) xs  -- in alternativa: xs_inf = [x' | x' <- xs, x' <= x]
    xs_sup = filter (> x) xs   -- in alternativa: xs_sup = [x' | x' <- xs, x' > x]
```

Esso risulta essere estremamente conciso, se confrontato col corrispondente programma imperativo, e non necessita di puntatori per scorrere la lista che contiene gli elementi da ordinare, né presenta indicazioni per allocare la memoria necessaria per creare la versione ordinata della lista.
La prima linea dichiara (`::`) il tipo dell'identificatore `quicksort` stabilendo che quest'ultimo è una funzione (`[a] -> [a]`) che ha una lista come argomento (`[a]` a sinistra di `->`) e restituisce una lista come risultato (`[a]` a destra di `->`), dove `a` è una variabile di tipo. Gli elementi di queste liste sono di un generico tipo `a` su cui sono definiti operatori d'ordine (la premessa `(Ord a) =>` significa che `a` è un'istanza della classe di tipi `Ord`), quindi l'identificatore `quicksort` rappresenta una funzione soggetta a overloading, che è una forma ristretta di polimorfismo.
La seconda e la terza linea utilizzano il pattern matching per esprimere il caso base e il caso generale della definizione ricorsiva. Il primo caso stabilisce che se la lista argomento è vuota (`[]` a sinistra di `=`) allora anche la lista risultato è vuota (`[]` a destra di `=`). Il secondo caso stabilisce che se la lista argomento non è vuota, precisamente se ha un primo elemento indicato con l'identificatore `x` seguito (operatore infisso `:`) dalla lista eventualmente vuota dei successivi elementi indicata con l'identificatore `xs`, allora la lista risultato è ottenuta concatenando (operatore infisso `++`) il risultato dell'applicazione di `quicksort` a `xs_inf` con la lista `[x]` e poi con il risultato dell'applicazione di `quicksort` a `xs_sup`. L'applicazione di una funzione in notazione prefissa ha precedenza su tutti gli operatori infissi (da cui le parentesi tonde attorno a `x : xs`) ed è associativa da sinistra (come nel λ-calcolo).
La sezione `where` definisce gli identificatori `xs_inf` e `xs_sup` stabilendo che essi sono ottenuti da `xs` selezionando gli elementi opportuni tramite le specializzazioni `(<= x)` e `(> x)` degli operatori `<=` e `>`.
La funzione `filter` è predefinita, ma se dovessimo definirla procederemmo nel seguente modo:

```
filter :: (a -> Bool) -> [a] -> [a]
filter _ []                   = []
filter p (x : xs) | p x       = x : filter p xs  -- in alternativa: [x] ++ filter p xs
                  | otherwise = filter p xs
```

La funzione `filter` è polimorfa e, come dichiarato nella sua prima linea, ha come argomenti un predicato, cioè una funzione che restituisce un valore di verità (`a -> Bool`), e una lista (prima occorrenza di `[a]`), poi restituisce come risultato una lista (seconda occorrenza di `[a]`), pertanto è una funzione di ordine superiore. Nella dichiarazione del tipo di una funzione, il numero di argomenti della funzione è pari al numero di frecce che si trovano fuori da eventuali parentesi tonde.
Nella definizione ricorsiva viene usato `_` nel caso base perché il predicato non compare a destra di `=` e quindi non è necessario dargli un nome. Il caso generale comprende due sottocasi da valutare nell'ordine in cui sono scritti, ciascuno dei quali presenta una guardia tra `|` e `=`. Nel primo sottocaso si controlla se `p` è vero in `x` così da includere `x` nella lista risultato seguita (operatore infisso `:`) dall'applicazione ricorsiva di `filter` a `xs`, mentre nel secondo sottocaso (`otherwise`) l'elemento `x` non viene incluso.
Il tipo generico di `quicksort`, dichiarato per chiarezza anche se non obbligatorio dato che Haskell supporta l'inferenza di tipi, consente alla funzione, definita una sola volta, di essere applicata a liste di elementi di tipi specifici di volta in volta diversi come valori numerici, logici e caratteri:

```
quicksort [3, 2, 1]                    [1,2,3]
quicksort [3, 2.5, -1]                 [-1.0,2.5,3.0]
quicksort [True, False]                [False,True]
quicksort ['c', 'i', 'a', 'o']         "acio"
quicksort "ciao"                       "acio"
```

■fltplf_7

4.3 Haskell: Espressioni, Tipi di Dati, Classi di Tipi

- Un'espressione in Haskell è formata da valori, operatori e identificatori, i quali determinano il tipo e il valore dell'espressione. L'espressione viene valutata, se e solo se è ben tipata, applicando gli operatori nel rispetto delle regole di precedenza e associatività (che è sempre da sinistra anche per quelli unari).
- I tipi di dati scalari predefiniti disponibili in Haskell sono i seguenti:
  - `Int` e `Integer` denotano l'insieme dei numeri interi, cioè i numeri espressi senza punto decimale, rappresentati in memoria in precisione fissa o arbitraria rispettivamente.
  - `Float` e `Double` denotano l'insieme dei numeri reali in precisione singola o doppia rispettivamente, espressi in virgola fissa o mobile (con la prima cifra della mantissa a sinistra del punto decimale).
  - `Bool` denota l'insieme dei valori di verità `False` e `True`.
  - `Char` denota l'insieme dei caratteri stampabili, di spaziatura e di controllo, espressi tra apici.
- Haskell introduce il concetto di classe di tipi come collezione di tipi correlati tra loro, nel senso che i loro insiemi di valori sono in qualche misura compatibili e condividono determinati operatori che quindi sono soggetti a overloading. Le classi formano una gerarchia dotata dei consueti meccanismi di ereditarietà: se S è una sottoclasse di C allora tutte le istanze di S sono istanze anche di C, così come tutti gli operatori disponibili in C sono disponibili anche in S.
- Questa è la gerarchia delle classi con i loro operatori principali per i tipi scalari summenzionati:
  - `Eq`:
    ```
    (==)     :: a -> a -> Bool
    (/=)     :: a -> a -> Bool
    ```
  - `Ord` sottoclasse di `Eq`:
    ```
    (<)      :: a -> a -> Bool
    (<=)     :: a -> a -> Bool
    (>)      :: a -> a -> Bool
    (>=)     :: a -> a -> Bool
    min      :: a -> a -> a
    max      :: a -> a -> a
    ```
  - `Bounded`:
    ```
    minBound :: a
    maxBound :: a
    ```
  - `Enum`:
    ```
    succ     :: a -> a
    pred     :: a -> a
    ```
  - `Num` sottoclasse di `Eq` (non di `Ord` perché i complessi non sono ordinabili) comprensiva degli interi:
    ```
    (+)      :: a -> a -> a
    (-)      :: a -> a -> a
    (*)      :: a -> a -> a
    negate   :: a -> a
    abs      :: a -> a
    signum   :: a -> a
    ```
  - `Integral` sottoclasse di `Num`, `Enum`, `Ord` (l'operatore `^` richiede che soltanto l'esponente sia intero):
    ```
    div      :: a -> a -> a
    mod      :: a -> a -> a
    (^)      :: (Num b) => b -> a -> b
    gcd      :: a -> a -> a
    lcm      :: a -> a -> a
    even     :: a -> Bool
    odd      :: a -> Bool
    ```

- `Fractional` sottoclasse di `Num` comprensiva dei reali:
  ```
  (/)      :: a -> a -> a
  recip    :: a -> a
  ```
- `RealFrac` sottoclasse di `Fractional`:
  ```
  truncate :: (Integral b) => a -> b
  round    :: (Integral b) => a -> b
  ceiling  :: (Integral b) => a -> b
  floor    :: (Integral b) => a -> b
  ```
- `Floating` sottoclasse di `Fractional`:
  ```
  pi       :: a
  exp      :: a -> a
  log      :: a -> a
  (**)     :: a -> a -> a
  logBase  :: a -> a -> a
  sqrt     :: a -> a
  sin      :: a -> a
  cos      :: a -> a
  tan      :: a -> a
  ```

dove:

- `Int` e `Integer` sono istanze di `Integral` e quindi anche di `Num`, `Enum`, `Ord`, `Eq`; inoltre `Int` è anche istanza di `Bounded`.
- `Float` e `Double` sono istanze di `Floating`, e quindi anche di `Fractional`, `Num`, `Eq`, oltre che di `Enum` e `Ord`.
- `Bool` e `Char` sono istanze di `Eq`, `Ord`, `Bounded`, `Enum`; inoltre `Bool` possiede i seguenti operatori specifici (quelli binari sono soggetti a cortocircuitazione per coerenza con la valutazione pigra):
  ```
  (||)     :: Bool -> Bool -> Bool
  (&&)     :: Bool -> Bool -> Bool
  not      :: Bool -> Bool
  ```
  ed è alla base dell'espressione condizionale `if` c `then` e_1 `else` e_2 dove c deve essere di tipo `Bool` mentre e_1 ed e_2 devono essere dello stesso tipo (operatore ternario infisso cortocircuitato).

- Nelle espressioni gli operatori denotati da identificatori vengono scritti in notazione prefissa a meno che non siano binari e vengano racchiusi tra apici inversi, mentre gli operatori binari denotati da simboli vengono scritti in notazione infissa a meno che non vengano racchiusi tra parentesi tonde.
- L'applicazione di funzione, e quindi di operatore prefisso, ha precedenza sugli operatori binari infissi. I due operandi di un operatore binario possono essere di classi di tipi diverse a patto che una delle due classi sia sottoclasse dell'altra, nel qual caso è la sottoclasse a determinare il tipo dell'espressione.
- Esempi:
  - `div 7 2` equivale a `7 ‘div‘ 2` e restituisce `3` perché `div :: (Integral a) => a -> a -> a`.
  - `7 / 2` equivale a `(/) 7 2` e restituisce `3.5` perché `(/) :: (Fractional a) => a -> a -> a`.
  - `4 + 1.0` produce `5.0` perché `4 :: Num` e `1.0 :: Fractional` con `Fractional` sottoclasse di `Num`.
  - `9 :: Int` converte `9` dal tipo `Num` al tipo `Int` mentre `fromIntegral (9 :: Int)` lo riporta a `Num`.
  - `minBound :: Int` e `maxBound :: Int` restituiscono il minimo e il massimo intero rappresentabile.
  - `0.026` in virgola fissa è equivalente a `2.6e-2` in virgola mobile.
  - `abs -8.3` viene interpretata come `abs` applicato a `-`, quindi va riscritta come `abs (-8.3)`.
  - `False < True` è ben tipata, perché `Bool` è istanza di `Ord`, e restituisce `True`.
  - `not not True` va riscritta come `not (not True)`, altrimenti prevale l'associatività da sinistra.

- Il programmatore può definire un nuovo tipo di dato scalare attraverso il meccanismo dell'enumerazione, specificando dopo `data` l'identificatore del nuovo tipo seguito da `=` e poi dagli identificatori dei costruttori dei valori del nuovo tipo separati da `|`. Ciascuno di questi identificatori, che devono essere tutti diversi tra loro, è una sequenza di lettere, cifre, sottotratti e apici che inizia con una maiuscola. La definizione del nuovo tipo può inoltre comprendere la clausola `deriving` per indicare le classi di tipi di cui il nuovo tipo è istanza (e di cui eredita pertanto gli operatori), nonché la definizione di operatori specifici del nuovo tipo. Un esempio di tipo di dato predefinito enumerato è `Bool`.
- Esempio di tipo di dato enumerato per i giorni della settimana:

```
data Giorno = Lunedi | Martedi | Mercoledi | Giovedi | Venerdi | Sabato | Domenica
  deriving (Eq, Ord, Enum)
domani :: Giorno -> Giorno
domani Domenica = Lunedi
domani g        = succ g
ieri :: Giorno -> Giorno
ieri Lunedi = Domenica
ieri g      = pred g
```

- I tipi di dati strutturati predefiniti disponibili in Haskell sono tupla, lista e stringa. Le tuple sono istanze di `Eq`, `Ord`, `Bounded` mentre le liste sono istanze di `Eq`, `Ord`. Le stringhe sono implementate come liste di `Char`.
- Una tupla è una sequenza di un prefissato numero $n \in \mathbb{N}_{\geq 2}$ di espressioni separate da virgole e racchiuse tra parentesi tonde. Queste espressioni possono essere di tipi diversi tra loro, quindi se l'espressione i ha tipo t_i allora la tupla ha tipo `(`t_1`, ..., `t_n`)`. Le tuple possono annidarsi all'interno di altre tuple e sono utili ad esempio quando una funzione deve restituire più risultati. Due tuple sono confrontabili tramite gli operatori di `Eq` e `Ord` solo se sono dello stesso tipo, nel qual caso l'ordinamento considerato è quello lessicografico. Esistono inoltre degli operatori specifici per estrarre gli elementi delle coppie:

```
fst      :: (a, b) -> a
snd      :: (a, b) -> b
```

- Una lista è una sequenza di un arbitrario numero di espressioni tutte dello stesso tipo separate da virgole e racchiuse tra parentesi quadre, dove la lista vuota è rappresentata da `[]`. Se le espressioni sono di tipo t, allora la lista è di tipo `[`t`]`; in particolare, una stringa è di tipo `[Char]` e i caratteri che la costituiscono possono essere equivalentemente racchiusi tra virgolette senza separarli con virgole. Le liste possono annidarsi all'interno di altre liste e sono confrontabili tramite gli operatori di `Eq` e `Ord` anche se di lunghezze diverse a patto che siano dello stesso tipo, nel qual caso l'ordinamento considerato è quello lessicografico. Esistono inoltre degli operatori specifici per costruire liste:

```
(:)      :: a -> [a] -> [a]
(++)     :: [a] -> [a] -> [a]
```

  dove `:` è il costruttore canonico giacché `[`v_1`, ..., `v_n`]` si scompone unicamente in v_1 `: ... :` v_n `: []`. Poiché i suoi due argomenti sono di tipi diversi, di fatto l'operatore `:` è associativo da destra.
- Il programmatore può ridenominare un tipo strutturato specificando dopo `type` il nuovo identificatore di tipo seguito da `=` e poi dal tipo da ridenominare. Ad esempio `String` è un alias di `[Char]` mentre `type Relazione a = [(a, a)]` crea un nuovo alias, facendo in entrambi i casi aumentare la leggibilità.
- È inoltre possibile definire i tipi unione e i tipi ricorsivi tramite il meccanismo dell'enumerazione esteso con la specifica di zero o più tipi dopo ogni costruttore. Ad esempio `data Numero = I Int | F Float` cui appartengono `I 9` ed `F 3.5`, oppure `data Nat = Z | S Nat` che dà luogo a `Z`, `S Z`, `S (S Z)`, ecc.
- In Haskell il valore di un'espressione può essere legato a una variabile mediante una definizione, il che è utile sia per incrementare la leggibilità che per non dover ricalcolare quel valore. L'identificatore di una variabile è una sequenza di lettere, cifre, sottotratti e apici che inizia con una minuscola o un sottotratto (in quest'ultimo caso il sottotratto deve essere seguito da almeno un altro carattere tra quelli ammessi); all'interno degli identificatori le lettere minuscole sono considerate diverse dalle corrispondenti lettere maiuscole (case sensitivity). Ad esempio `pi` è definita come `3.141592653589793` mentre `otherwise` è definita come `True`.

- La creazione del legame tra una variabile e il valore di un'espressione tramite `=` è di solito preceduta dalla dichiarazione del tipo della variabile tramite `::`, sebbene ciò non sia obbligatorio grazie al meccanismo di inferenza di tipi. Ogni variabile può comparire sulla sinistra di una sola definizione e il suo valore non può più cambiare (costante simbolica). L'ordine delle definizioni è irrilevante, quindi il programmatore deve solo preoccuparsi che non ci siano dipendenze circolari che farebbero divergere la valutazione.
- Le definizioni si dividono in globali e locali e possono essere arricchite con commenti. Questi devono essere racchiusi tra `{-` e `-}` oppure iniziare con `--` se residenti su un'unica linea.
- Esempio di definizioni globali equivalenti tra loro:

```
{- dall'alto verso il basso -}   {- dal basso verso l'alto -}

circonferenza :: Float           raggio :: Float
circonferenza = pi * diametro    raggio = sqrt 15.196

diametro :: Float                diametro :: Float
diametro = 2 * raggio            diametro = 2 * raggio

raggio :: Float                  circonferenza :: Float
raggio = sqrt 15.196             circonferenza = pi * diametro
```

- Una definizione locale consente di creare legami che hanno validità solo all'interno della definizione globale a cui sono associati. Essa si esprime attraverso il costrutto `where` o il costrutto `let-in`, ciascuno dei quali deve essere indentato e può contenere molteplici legami tra variabili e valori di espressioni a patto che tali legami stiano su linee diverse in quanto privi di terminatori.
- Esempio di definizioni globali con definizioni locali associate equivalenti tra loro:

```
{- dall'alto verso il basso -}   {- dal basso verso l'alto -}

circonferenza :: Float           circonferenza :: Float
circonferenza = pi * diametro    circonferenza = let
  where                                            raggio = sqrt 15.196
    diametro = 2 * raggio                          diametro = 2 * raggio
    raggio = sqrt 15.196                         in
                                                   pi * diametro
```

■fltplf_8

4.4 Haskell: Funzioni, Guardie, Pattern Matching

- In Haskell il formato della definizione di una funzione è un'estensione del meccanismo per legare il valore di un'espressione a una variabile. Le due differenze più significative sono la presenza dell'operatore $\rightarrow$ nell'eventuale dichiarazione di tipo e la presenza degli argomenti formali nella definizione. Gli argomenti formali, che devono avere identificatori diversi tra di loro, non sono racchiusi tra parentesi tonde né sono separati da virgole. Il tipo dell'i-esimo argomento è quello compreso tra l'$i-1$-esima $\rightarrow$ e l'i-esima $\rightarrow$, mentre il tipo del risultato è quello che segue l'ultima $\rightarrow$, dove le frecce da considerare sono quelle che si trovano fuori da eventuali parentesi tonde e quindi gli argomenti sono tanti quanti queste frecce.
- La definizione di una funzione consiste in una o più equazioni (casi base e generale della ricorsione) ciascuna delle quali stabilisce che l'applicazione della funzione ai suoi argomenti, che si trova a sinistra di `=`, e il corpo della funzione, che si trova a destra di `=`, denotano lo stesso valore. In alternativa, può essere vista come una λ-astrazione, del resto l'applicazione della funzione ad argomenti effettivi segue la β-riduzione, cioè comporta la sostituzione degli argomenti effettivi ai corrispondenti argomenti formali all'interno del corpo della funzione.
- La definizione di una funzione può essere globale o locale e può stabilire comportamenti diversi a seconda del valore dei suoi argomenti. Questo può essere descritto ricorrendo a espressioni condizionali eventualmente annidate oppure, secondo uno stile più dichiarativo, dividendo la definizione in più casi, ciascuno dei quali è subordinato a una guardia, cioè un'espressione di tipo `Bool` scritta tra `|` e `=` che di solito è `otherwise` per l'ultimo caso, i quali vengono esaminati nell'ordine in cui sono elencati.

- Esempio: La funzione predefinita `signum :: (Num a, Ord a) => a -> a` potrebbe essere definita nel seguente modo attraverso espressioni condizionali annidate:

```
signum x = if (x > 0)
             then 1
             else
               if (x == 0)
                 then 0
                 else -1
```

  ma in realtà è definita in uno stile più dichiarativo (e più leggibile) come segue:

```
signum x | x > 0  = 1
         | x == 0 = 0
         | x < 0  = -1
```

  che è equivalente a:

```
signum x | x > 0     = 1
         | x == 0    = 0
         | otherwise = -1
```

  L'argomento effettivo a cui è applicata va racchiuso tra parentesi tonde qualora sia un numero negativo oppure contenga operatori binari infissi, come in `signum (-1)` e `signum (2 * 5)`.

- Uno stile ancor più dichiarativo consiste nel definire le funzioni ricorrendo al pattern matching anziché alle guardie per la formalizzazione dei vari casi. Un pattern (modello) è un costrutto sintattico che specifica valori correlati e fornisce nomi per accedere a parti di tali valori, da usare nelle equazioni a livello di argomenti formali. In fase di applicazione della funzione considerata verrà verificato il matching (corrispondenza) degli argomenti effettivi ai pattern presenti a sinistra di `=`:
  - Ogni valore numerico, logico, carattere, enumerato, tupla, lista o stringa è un pattern che intercetta unicamente il valore stesso (utile per definire i casi base della ricorsione).
  - Ogni identificatore è un pattern che intercetta un qualsiasi valore, il quale viene legato all'identificatore così da potervi far riferimento a destra di `=` nella definizione del caso in esame.
  - Il simbolo `_` è un pattern che intercetta un qualsiasi valore al quale non si fa più riferimento a destra di `=` nella definizione del caso in esame.
  - Se p_1 e p_2 sono pattern, allora anche p_1 `:` p_2 è un pattern che intercetta una lista il cui primo elemento corrisponde a p_1 e la cui sottolista degli elementi successivi corrisponde a p_2.
  - Se $p_1, \ldots, p_n$ sono pattern, dove $n \in \mathbb{N}_{\geq 2}$, allora anche `(`$p_1$`, ..., `$p_n$`)` è un pattern che intercetta una tupla il cui i-esimo elemento corrisponde a p_i.
  - Se p è un pattern e x è un identificatore, allora anche x`@`p è un pattern, detto ascrizione, che è utile nel caso in cui p non sia un valore scalare o un identificatore perché in quel caso a destra di `=` si può far riferimento tramite x all'intero valore intercettato senza dover ripetere p.

- Ogni pattern ha un tipo che è determinato dal tipo dei valori che corrispondono al pattern. Tutti i pattern per un argomento formale all'interno delle varie equazioni per una funzione devono essere dello stesso tipo e questo tipo deve coincidere con quello dell'argomento. Un identificatore può comparire al più una volta all'interno di un singolo pattern così come nella sequenza dei pattern per gli argomenti formali di una singola equazione (vincolo di linearità).

- I casi definiti tramite equazioni, pattern matching, guardie o una loro combinazione sono esaustivi quando coprono ogni possibilità rispetto ai tipi degli argomenti; se non sono esaustivi la funzione risulta essere parziale, cioè non definita per certi valori degli argomenti. Due casi si sovrappongono quando sono entrambi verificati per almeno una combinazione degli argomenti; in tale eventualità viene seguito il caso elencato per primo. I pattern `[]` e `_ : _` non si sovrappongono mai.

- Esempi relativi alle funzioni su valori numerici, logici e carattere:
  - Funzione per calcolare il fattoriale di un numero naturale (guardia necessaria per escludere $n < 0$):

```
fattoriale :: Int -> Int
fattoriale 0         = 1
fattoriale n | n > 0 = n * fattoriale (n - 1)
```

  - Funzione per calcolare l'n-esimo numero di Fibonacci (guardia necessaria per escludere $n < 0$):

```
fibonacci :: Int -> Int
fibonacci 0         = 0
fibonacci 1         = 1
fibonacci n | n > 1 = fibonacci (n - 1) + fibonacci (n - 2)
```

  - Funzioni per implementare gli operatori binari e ternari cortocircuitati:

```
disg :: Bool -> Bool -> Bool  cong :: Bool -> Bool -> Bool  cond :: Bool -> a -> a -> a
disg True _  = True           cong False _ = False          cond True e _  = e
disg False x = x              cong True x  = x              cond False _ e = e
```

  - Funzione per confrontare due caratteri (la prima versione viola il vincolo di linearità):

```
confronta :: Char -> Char -> Int    confronta :: Char -> Char -> Int
confronta c c             = 0       confronta c1 c2 | c1 == c2 = 0
confronta c1 c2 | c1 < c2 = -1                    | c1 < c2  = -1
                | c1 > c2 = 1                     | c1 > c2  = 1
```

- Esempi relativi alle liste:
  - Funzione per calcolare il numero di elementi presenti in una lista:

```
lunghezza :: [a] -> Int
lunghezza []       = 0
lunghezza (_ : xs) = 1 + lunghezza xs
```

  - Funzioni parziali per estrarre il primo elemento di una lista e la sottolista degli elementi successivi:

```
testa :: [a] -> a    coda :: [a] -> [a]
testa (x : _) = x    coda (_ : xs) = xs
```

  - Funzione per stabilire se un elemento appartiene a una lista:

```
membro :: (Eq a) => a -> [a] -> Bool
membro _ []                   = False
membro x (y : ys) | x == y    = True
                  | otherwise = membro x ys
```

  - Funzione per stabilire se una lista è un prefisso di un'altra lista (cioè compare all'inizio):

```
prefisso :: (Eq a) => [a] -> [a] -> Bool
prefisso [] _                          = True
prefisso (_ : _) []                    = False
prefisso (x : xs) (y : ys) | x == y    = prefisso xs ys
                           | otherwise = False
```

  - Funzione per stabilire se una lista è un suffisso di un'altra lista (cioè compare alla fine):

```
suffisso :: (Eq a) => [a] -> [a] -> Bool
suffisso [] _                                  = True
suffisso (_ : _) []                            = False
suffisso lx@(x : xs) ly@(y : ys) | lx == ly  = True
                                 | otherwise = suffisso lx ys
```

- Funzione per stabilire se una lista è una sottolista di un'altra lista (cioè compare all'interno):

```
sottolista :: (Eq a) => [a] -> [a] -> Bool
sottolista [] _                                    = True
sottolista (_ : _) []                              = False
sottolista lx@(x : xs) ly@(y : ys) | prefisso lx ly = True
                                   | otherwise      = sottolista lx ys
```

- Funzione per aggiungere un elemento all'inizio di una lista:

```
inserisci_elem :: a -> [a] -> [a]
inserisci_elem x xs = x : xs
```

- Funzione per aggiungere un elemento alla fine di una lista:

```
accoda_elem :: a -> [a] -> [a]
accoda_elem x xs = xs ++ [x]
```

- Funzione per concatenare due liste:

```
concatena :: [a] -> [a] -> [a]
concatena xs ys = xs ++ ys
```

- Funzione per invertire l'ordine degli elementi di una lista:

```
inverti :: [a] -> [a]
inverti []       = []
inverti (x : xs) = inverti xs ++ [x]
```

- Funzione per rimuovere tutte le occorrenze di un elemento da una lista:

```
rimuovi_elem :: (Eq a) => a -> [a] -> [a]
rimuovi_elem _ []                    = []
rimuovi_elem x (y : ys) | x == y    = rimuovi_elem x ys
                        | otherwise = y : rimuovi_elem x ys
```

- Funzione per fondere ordinatamente due liste ordinate:

```
fondi :: (Ord a) => [a] -> [a] -> [a]
fondi xs []                               = xs
fondi [] ys                               = ys
fondi lx@(x : xs) ly@(y : ys) | x < y  = x : fondi xs ly
                              | x == y = x : y : fondi xs ys
                              | x > y  = y : fondi lx ys
```

- Esempi relativi agli algoritmi di ordinamento:
  - Mergesort:

```
mergesort :: (Ord a) => [a] -> [a]
mergesort []                = []
mergesort [x]               = [x]
mergesort lx@(x1 : x2 : xs) = fondi (mergesort xs1) (mergesort xs2)
  where
    (xs1, xs2) = dimezza lx

dimezza :: [a] -> ([a], [a])
dimezza []             = ([], [])
dimezza [x]            = ([x], [])
dimezza (x1 : x2 : xs) = (x1 : xs1, x2 : xs2)
  where
    (xs1, xs2) = dimezza xs
```

  - Quicksort: vedi fine Sez. 4.2.

4.5 Haskell: Funzioni Polimorfe, di Ordine Superiore, Anonime

- Il polimorfismo rappresenta la possibilità per una funzione di lavorare su argomenti di tipi di volta in volta diversi e per una struttura dati di contenere valori di tipi di volta in volta diversi. Ad esempio, grazie al supporto al polimorfismo disponibile in Haskell, la funzione identità può essere definita una volta per tutte come `identita x = x` favorendo pertanto il riuso di codice, invece di doverla ridefinire per ogni possibile diverso tipo del suo unico argomento formale `x`. Se il suo tipo non viene dichiarato, allora viene inferito il tipo più generale o principale, che è `identita :: t -> t`, cioè il tipo dal quale sono ricavabili tutte le istanze valide di tipi specifici per quella funzione.
- In aggiunta a tipi predefiniti, tipi definiti dal programmatore e costruttori di tipo come `->` per le funzioni, `()` per le tuple e `[]` per le liste, la dichiarazione del tipo di una funzione polimorfa contiene variabili di tipo, come ad esempio `a` in `identita :: a -> a`. Esse sono identificatori che rappresentano tipi generici, i quali verranno istanziati all'atto dell'applicazione della funzione ad argomenti effettivi. Ogni variabile di tipo è rigida all'interno del corpo della funzione, nel senso che tutte le occorrenze di argomenti formali di quel tipo dovranno poter essere istanziate allo stesso tipo specifico.
- Esempi:
  - `identita 1 :: (Num t) => t`
  - `identita 2.5 :: (Fractional t) => t`
  - `identita True :: Bool`
  - `identita 'Z' :: Char`
  - `identita ("ingresso", "uscita") :: ([Char], [Char])`
  - La variante `f x = if x then x else x` non può essere di tipo `a -> a` perché la prima occorrenza di `x` a destra di `=` deve essere di tipo `Bool`, quindi `f` non può essere polimorfa.
- L'overloading di una funzione può essere visto come una forma ristretta di polimorfismo. Il motivo è che nella dichiarazione di tipo di una funzione soggetta a overloading le variabili di tipo non denotano tipi completamente generici, ma istanze di una o più classi di tipi. Questo viene descritto facendo precedere l'espressione di tipo per la funzione dalle suddette classi, separate da virgole e racchiuse tra parentesi tonde, seguite da `=>`, come ad esempio in `(Num a, Ord a) => a -> a`.
- Poiché in Haskell non c'è differenza concettuale tra funzioni e valori, una funzione può essere passata come argomento, restituita come risultato o contenuta in una struttura dati. In particolare, una funzione che accetta funzioni come argomenti oppure restituisce funzioni come risultati è detta essere una funzione di ordine superiore. Il suo tipo contiene almeno un'occorrenza dell'operatore $\rightarrow$ racchiusa tra parentesi tonde.
- Esempi:
  - Funzione predefinita `filter` che seleziona gli elementi di una lista in base a un certo predicato:

```
filter :: (a -> Bool) -> [a] -> [a]
filter _ []                   = []
filter p (x : xs) | p x       = x : filter p xs
                  | otherwise = filter p xs
```

  - Funzione predefinita `map` che applica una certa funzione a tutti gli elementi di una lista:

```
map :: (a -> b) -> [a] -> [b]
map _ []       = []
map f (x : xs) = f x : map f xs
```

- Funzioni predefinite che riducono gli elementi di una lista a un singolo valore combinandoli a due a due da sinistra verso destra o viceversa tramite una certa funzione e un eventuale valore iniziale:

```
foldl :: (b -> a -> b) -> b -> [a] -> b    foldr :: (a -> b -> b) -> b -> [a] -> b
foldl _ i []       = i                     foldr _ i []       = i
foldl f i (x : xs) = foldl f (f i x) xs    foldr f i (x : xs) = f x (foldr f i xs)

foldl1 :: (a -> a -> a) -> [a] -> a        foldr1 :: (a -> a -> a) -> [a] -> a
foldl1 f (x : xs) = foldl f x xs           foldr1 _ [x]      = x
                                           foldr1 f (x : xs) = f x (foldr1 f xs)
```

- Funzioni predefinite sulle liste basate sulle funzioni precedenti che combinano elementi:

```
sum :: (Num a) => [a] -> a        product :: (Num a) => [a] -> a
sum = foldl (+) 0                 product = foldl (*) 1

minimum :: (Ord a) => [a] -> a    maximum :: (Ord a) => [a] -> a
minimum = foldl1 min              maximum = foldl1 max

or :: [Bool] -> Bool              and :: [Bool] -> Bool
or = foldr (||) False             and = foldr (&&) True

reverse :: [a] -> [a]             flip :: (a -> b -> c) -> b -> a -> c
reverse = foldl (flip (:)) []     flip f x y = f y x
```

  e loro applicazioni:

```
sum [4, 5, 6] = foldl (+) 0 [4, 5, 6] = foldl (+) 4 [5, 6]
              = foldl (+) 9 [6] = foldl (+) 15 [] = 15

reverse [4, 5, 6] = foldl (flip (:)) [] [4, 5, 6] = foldl (flip (:)) [4] [5, 6]
                  = foldl (flip (:)) [5, 4] [6] = foldl (flip (:)) [6, 5, 4] []
                  = [6, 5, 4]
```

- Nelle liste capita spesso di dover ripetere la medesima computazione su ciascun elemento della struttura, trasformando tutti gli elementi allo stesso modo o filtrando ogni elemento in base alla stessa condizione. In Haskell ciò può essere anche espresso ricorrendo alle list comprehension, molto simili alla notazione insiemistica, con `[f x | x <- xs]` dove `f` è una funzione o `[x | x <- xs, p x]` dove `p` è un predicato. Inoltre è disponibile l'abbreviazione `[e_1 .. e_2]`, la quale genera la lista di tutti i valori compresi tra i valori di due espressioni di un tipo che è istanza di `Enum`.

- Esempi:

  - Funzione per calcolare il successore di ogni elemento di una lista di valori di tipo enumerato:

```
lista_succ :: (Enum a) => [a] -> [a]     {- versione equivalente con map -}
lista_succ xs = [succ x | x <- xs]       lista_succ = map succ
```

  - Funzione per filtrare gli elementi pari di una lista di interi:

```
lista_pari :: [Int] -> [Int]             {- versione equivalente con filter -}
lista_pari xs = [x | x <- xs, even x]    lista_pari = filter even
```

  - Generazione della lista di coppie di cifre decimali di cui la seconda non è maggiore della prima:

```
coppie_cifre :: [(Int, Int)]
coppie_cifre = [(x, y) | x <- [0 .. 9], y <- [0 .. x]]
```

- In Haskell le funzioni possono essere definite anche al volo senza dare loro un nome seguendo quasi la stessa sintassi delle λ-astrazioni, dove λ è rimpiazzato da \ mentre . è rimpiazzato da ->. Ad esempio `\x -> x + 1` è una funzione anonima che calcola il successore e la sua applicazione `(\x -> x + 1) 2` restituisce `3`.
- Queste λ-espressioni possono essere usate pure all'interno delle definizioni di funzioni. L'effetto è quello di spostare gli argomenti formali dalla sinistra alla destra di = aprendo di conseguenza la via al currying, cioè alla possibilità di esprimere `f x_1 ... x_n = e` come `f = \x_1 -> ... \x_n -> e` e quindi di vedere alcune funzioni come specializzazioni di altre. Ad esempio `(==) 0` è il test di uguaglianza a `0` – manca il secondo argomento – mentre `(< 0)` è il test di negatività – manca il primo argomento.
- Esempio:

```
successore :: (Num a) => a -> a    {- lambda espressione -}    {- specializzazione -}
successore x = x + 1                successore = \x -> x + 1    successore = (+) 1
```

- Haskell fornisce l'operatore binario infisso . per la composizione di funzioni e l'operatore binario infisso $ come applicazione di funzione su cui tutti gli altri operatori binari infissi hanno precedenza (diversamente dall'applicazione standard), i quali sono entrambi definiti tramite λ-espressioni:

```
(.) :: (b -> c) -> (a -> b) -> a -> c
f . g = \x -> f (g x)

($) :: (a -> b) -> a -> b
f $ x = f (x)
```

- Esempi:
  - `testa . coda` è una funzione che restituisce il secondo elemento di una lista.
  - `signum 2 * 5` vale `5`, `signum (2 * 5)` vale `1`, `signum $ 2 * 5` vale `1`.

4.6 Haskell: Valutazione Pigra, Input/Output, Moduli

- In Haskell gli argomenti effettivi di una funzione non vengono valutati all'atto della chiamata della funzione, ma nel momento in cui vengono usati dalla funzione invocata (chiamata per necessità e conseguente valutazione pigra). Abbiamo già visto che questo meccanismo dà naturalmente luogo alla cortocircuitazione degli operatori logici binari e dell'espressione condizionale. Esso consente inoltre la gestione di strutture dati illimitate interfogliando la loro generazione con la loro elaborazione.
- Esempio: La seguente funzione genera la lista di tutti gli interi compresi tra due interi dati:

```
genera_da_a :: Int -> Int -> [Int]
genera_da_a m n | m > n     = []
                | otherwise = m : genera_da_a (m + 1) n
```

mentre la seguente funzione genera la lista illimitata di tutti gli interi a partire da un intero dato:

```
genera_da :: Int -> [Int]
genera_da m = m : genera_da (m + 1)
```

La prima funzione può essere ridefinita nel seguente modo più compatto facendo uso della seconda:

```
genera_da_a' :: Int -> Int -> [Int]
genera_da_a' m n = take (n - m + 1) (genera_da m)
```

in quanto quest'ultima genera uno alla volta solo gli interi che sono man mano necessari all'esecuzione della funzione predefinita `take`, la quale restituisce la lista di un certo numero di elementi iniziali di una lista data.

- Mentre la valutazione pigra di espressioni genera valori seguendo le dipendenze tra i dati e senza creare effetti collaterali, le operazioni di input/output danno necessariamente luogo a effetti al di fuori della loro definizione. In Haskell sono rappresentate come azioni con effetto immediato che modificano in un certo ordine l'ambiente con cui un programma interagisce tramite l'acquisizione o la stampa di valori.
- Il costruttore di tipo `IO a` viene usato per denotare un'azione di input/output che produce un valore di tipo `a`. Esso si riduce a `IO ()`, dove `()` è il tipo unitario, quando il tipo dell'azione è irrilevante, altrimenti l'azione deve terminare con `return`. Le azioni di input/output non possono mischiarsi con le espressioni puramente funzionali e pertanto sono di solito raccolte all'inizio dei programmi Haskell.
- Le operazioni di input sono azioni che restituiscono i valori che sono stati acquisiti. Le operazioni di output non stampano direttamente valori, ma sono funzioni che producono azioni che stampano quei valori quando vengono eseguite. Queste sono le principali operazioni di input/output predefinite, dove l'acquisizione termina premendo invio nel caso di `getLine` o `<ctrl>d` nel caso di `getContents`:

```
getChar     :: IO Char                putChar  :: Char -> IO ()
getLine     :: IO String              putStr   :: String -> IO ()
getContents :: IO String              putStrLn :: String -> IO ()
```

- I valori acquisiti possono essere collocati in variabili tramite `<-` all'interno del costrutto indentato `do`. Questo costrutto consente in generale di combinare in una singola azione più azioni di input/output, le quali vengono eseguite nell'ordine in cui sono scritte quando viene richiamata l'azione combinata, insieme a eventuali espressioni funzionali dove le variabili vengono legate ai relativi valori con `let` e `=`.
- Esempi:
  - Azioni di input/output (le prime tre sono equivalenti, la prima e l'ultima sono combinate):

```
saluti :: IO ()           putStr "ciao\n"  putStrLn "ciao"  due_volte :: IO ()
saluti = do putChar 'c'                                     due_volte = do c <- getChar
            putChar 'i'                                                    putChar '\n'
            putChar 'a'                                                    putChar c
            putChar 'o'                                                    putChar c
            putChar '\n'                                                   putChar '\n'
```

  - Programma per risolvere il problema delle torri di Hanoi (dischi, partenza, arrivo, intermedia):

```
main :: IO ()
main = do let ms = hanoi 10 'a' 'c' 'b'
          putStrLn $ show ms

hanoi :: Int -> Char -> Char -> Char -> [(Char, Char)]
hanoi 1 p a _         = [(p, a)]
hanoi n p a i | n > 1 = hanoi (n - 1) p i a ++ [(p, a)] ++ hanoi (n - 1) i a p
```

- Poiché le operazioni di input/output predefinite trattano solo caratteri e stringhe, la gerarchia delle classi di tipi comprende anche le classi `Read` e `Show` che sono in qualche modo collegate all'input/output. Ad esempio, è necessario aggiungere la derivazione da esse alla definizione del tipo enumerato `Giorno` altrimenti i valori di questo tipo non sono acquisibili né stampabili. Ne sono istanze `Int`, `Integer`, `Float`, `Double`, `Bool`, `Char`, i tipi definiti dal programmatore che sono esplicitamente dichiarati come derivazioni di `Read` e `Show` e le tuple e le liste di valori di tutti i tipi precedenti.
- Le classi di tipi `Read` e `Show` contengono operatori per convertire rispettivamente le stringhe acquisite in valori di determinati tipi oppure i valori di determinati tipi in stringhe da stampare. Queste sono le loro principali funzioni predefinite, dove la prima richiede di solito una conversione di tipo con `::` per il valore estratto dalla stringa acquisita:

```
read :: (Read a) => String -> a       show :: (Show a) => a -> String
```

- Esempio: Se la funzione `quicksort` (vedi fine Sez. 4.2) fa parte di un programma Haskell, la seguente azione combinata di input/output chiamata `main` deve essere aggiunta all'inizio del programma:

```
main :: IO ()
main = do putStrLn "Digitare la lista dei valori da ordinare:"
          s <- getLine
          putStrLn "Lista ordinata:"
```

la cui ultima azione è diversa a seconda del tipo dei valori da ordinare ed è una delle seguenti:

```
putStrLn $ show (quicksort (read s :: [Int]))
putStrLn $ show (quicksort (read s :: [Float]))
putStrLn $ show (quicksort (read s :: [Bool]))
putStrLn $ show (quicksort (read s :: [Char]))
putStrLn $ quicksort s                              -- stringa tra virgolette
putStrLn $ show (quicksort (read s :: [Giorno]))  -- deriving (Read, Show)
```

- Un programma Haskell corposo viene solitamente organizzato in moduli che contengono definizioni di tipi, funzioni e variabili logicamente correlate tra loro. Tra questi moduli ce ne deve essere uno chiamato `Main` che esporta `main` di tipo `IO`, dove il valore di `main` è il valore dell'intero programma. Tutto ciò che è predefinito in Haskell viene messo a disposizione tramite il modulo `Prelude`.
- La definizione di un modulo comincia con `module` seguito dall'identificatore del modulo e poi da `where`. L'identificatore del modulo deve iniziare con una maiuscola e può essere accompagnato dall'elenco degli identificatori definiti nel modulo, separati da virgole e racchiusi tra parentesi tonde, che vengono esportati, altrimenti tutti gli identificatori globali definiti nel modulo sono considerati da esportare.
- Esempio di modulo per la gestione degli alberi binari:

```
module AlberoBin where

  data AlberoBin a = Nil | Nodo a (AlberoBin a) (AlberoBin a)
    deriving (Read, Show)

  cerca_albero_bin_ant :: (Eq a) => a -> AlberoBin a -> Bool
  cerca_albero_bin_ant _ Nil            = False
  cerca_albero_bin_ant n (Nodo m sx dx) = n == m ||
                                          cerca_albero_bin_ant n sx ||
                                          cerca_albero_bin_ant n dx

  cerca_albero_bin_post :: (Eq a) => a -> AlberoBin a -> Bool
  cerca_albero_bin_post _ Nil            = False
  cerca_albero_bin_post n (Nodo m sx dx) = cerca_albero_bin_post n sx ||
                                           cerca_albero_bin_post n dx ||
                                           n == m

  cerca_albero_bin_simm :: (Eq a) => a -> AlberoBin a -> Bool
  cerca_albero_bin_simm _ Nil            = False
  cerca_albero_bin_simm n (Nodo m sx dx) = cerca_albero_bin_simm n sx ||
                                           n == m ||
                                           cerca_albero_bin_simm n dx

  albero_bin_ric :: (Ord a) => AlberoBin a -> Bool
  albero_bin_ric Nil            = True
  albero_bin_ric (Nodo n sx dx) = albero_bin_ric sx && minore_ug sx n &&
                                  albero_bin_ric dx && maggiore_ug dx n
    where
      minore_ug Nil _            = True
      minore_ug (Nodo m sx dx) n = m <= n && minore_ug sx n && minore_ug dx n
      maggiore_ug Nil _            = True
      maggiore_ug (Nodo m sx dx) n = m >= n && maggiore_ug sx n && maggiore_ug dx n

  cerca_albero_bin_ric :: (Ord a) => a -> AlberoBin a -> Bool
  cerca_albero_bin_ric _ Nil                      = False
  cerca_albero_bin_ric n (Nodo m sx dx) | n == m = True
                                        | n < m  = cerca_albero_bin_ric n sx
                                        | n > m  = cerca_albero_bin_ric n dx
```

- In un modulo è possibile far riferimento agli identificatori esportati da un altro modulo. Se quest'ultimo non è `Prelude`, allora va importato all'inizio scrivendo `import` seguito dal suo identificatore.
- Poiché un modulo può ridefinire un identificatore definito in `Prelude` o in un modulo importato, al fine di evitare ambiguità ogni occorrenza dell'identificatore deve essere concatenato tramite `.` all'identificatore del modulo in cui è definito. In alternativa, se l'identificatore è originariamente definito in un modulo importato e dopo `import` si scrive `qualified`, la concatenazione non serve per le occorrenze dell'identificatore ridefinito, però ogni occorrenza di ogni identificatore del modulo importato deve essere concatenato tramite `.` all'identificatore del modulo. A tale modulo si può dare un nome più breve scrivendolo dopo `as` alla fine dell'importazione qualificata.
- Esempio di modulo per la gestione dei grafi diretti:

```
module GrafoDir (GrafoDir, grafo_dir, adiac,
                 cerca_grafo_dir_amp, cerca_grafo_dir_prof) where

  import qualified Lista as L

  type GrafoDir a = ([a], [(a, a)])

  grafo_dir :: (Eq a) => GrafoDir a -> Bool
  grafo_dir (vs, as) = controlla_v vs && controlla_a as vs
    where
      controlla_v []       = True
      controlla_v (v : vs) = not (L.membro v vs) && controlla_v vs
      controlla_a [] _               = True
      controlla_a ((v1, v2) : as) vs = L.membro v1 vs && L.membro v2 vs &&
                                       not (L.membro (v1, v2) as) && controlla_a as vs

  adiac :: (Eq a) => a -> [(a, a)] -> [a]
  adiac _ []                        = []
  adiac v ((v1, v2) : as) | v == v1 = v2 : adiac v as
                          | v /= v1 = adiac v as

  cerca_grafo_dir_amp :: (Eq a) => a -> GrafoDir a -> a -> Bool
  cerca_grafo_dir_amp v (vs, as) i = L.membro i vs && cerca v [i] as [] 'a'

  cerca_grafo_dir_prof :: (Eq a) => a -> GrafoDir a -> a -> Bool
  cerca_grafo_dir_prof v (vs, as) i = L.membro i vs && cerca v [i] as [] 'p'

  cerca :: (Eq a) => a -> [a] -> [(a, a)] -> [a] -> Char -> Bool
  cerca _ [] _ _ _                   = False
  cerca v (u : us) as vis t | v == u = True
                            | v /= u = if (L.membro u vis)
                                       then
                                         cerca v us as vis t
                                       else
                                         if (t == 'a')
                                           then
                                             cerca v (us ++ adiac u as) as (u : vis) t
                                           else
                                             cerca v (adiac u as ++ us) as (u : vis) t
```

dove l'ultima funzione può essere definita in modo più dichiarativo come segue:

```
cerca :: (Eq a) => a -> [a] -> [(a, a)] -> [a] -> Char -> Bool
cerca _ [] _ _ _                  = False
cerca v (u : us) as vis t | v == u = True
                          | v /= u = cerca' v (u : us) as vis t
  where
    cerca' v (u : us) as vis t | L.membro u vis = cerca v us as vis t
                               | otherwise      = cerca'' v u us as vis t
    cerca'' v u us as vis t | t == 'a' = cerca v (us ++ adiac u as) as (u : vis) t
                            | t == 'p' = cerca v (adiac u as ++ us) as (u : vis) t
```

- Oltre al modulo predefinito `Prelude`, il quale viene implicitamente importato in ogni altro modulo, Haskell mette a disposizione diverse librerie, ciascuna realizzata come un modulo da importare, tra cui citiamo in particolare:
  - `Data.Array` e `Data.Ix` per gli array immutabili.
  - `Data.Bits` per le operazioni sugli interi a livello di bit.
  - `Data.Char` per le operazioni sui caratteri e sulle stringhe.
  - `Data.Complex` per le operazioni sui numeri complessi in forma algebrica o trigonometrica.
  - `Data.List` per le operazioni sulle liste.
  - `System.IO` per le operazioni di input/output incluse quelle sui file. ■fltplf_10

Capitolo 5

Logica Proposizionale

5.1 Sintassi della Logica Proposizionale

- La logica proposizionale è una logica a due valori (vero e falso) basata su un insieme di proposizioni e su un insieme di connettivi attraverso i quali è possibile combinare le proposizioni per creare formule arbitrariamente complesse. Per proposizione intendiamo un'affermazione relativa a un singolo fatto di cui può essere stabilita la verità; di conseguenza sono escluse forme interrogative e dubitative così come affermazioni riguardanti il futuro oppure contenenti parti non completamente specificate.
- I valori di verità sono assimilabili a costanti logiche, le proposizioni sono assimilabili a variabili logiche, i connettivi sono assimilabili a operatori logici e le formule sono assimilabili a espressioni logiche.
- Gli elementi sintattici di base della logica proposizionale sono i seguenti:
  - Un insieme *Prop* di proposizioni, le quali verranno indicate con le lettere p, q, r.
  - I seguenti connettivi logici unari:
    - Negazione: $\neg$.
  - I seguenti connettivi logici binari:
    - Disgiunzione: $\vee$.
    - Congiunzione: $\wedge$.
    - Implicazione: $\rightarrow$.
    - Biimplicazione (o coimplicazione o doppia implicazione): $\leftrightarrow$.
  - I simboli ausiliari (e) usati per rendere non ambigua la lettura di formule che hanno più connettivi.
- I connettivi logici vanno intesi nel seguente modo: $\neg$ corrisponde a "non", $\vee$ corrisponde a "o" (nel senso inclusivo del "vel" latino), $\wedge$ corrisponde a "e", $\rightarrow$ corrisponde a "se ... allora ...", $\leftrightarrow$ corrisponde a "se e solo se".
- L'insieme *FBF* delle formule ben formate della logica proposizionale può essere definito come il più piccolo insieme che include *Prop* ed è chiuso rispetto a $\neg$, $\vee$, $\wedge$, $\rightarrow$, $\leftrightarrow$. Alternativamente, esso può essere definito come il più piccolo linguaggio su $\mathit{Prop} \cup \{\neg, \vee, \wedge, \rightarrow, \leftrightarrow, (,)\}$ tale che:
  - $\mathit{Prop} \subseteq \mathit{FBF}$.
  - $(\neg\phi) \in \mathit{FBF}$ per ogni $\phi \in \mathit{FBF}$.
  - $(\phi_1 \vee \phi_2) \in \mathit{FBF}$ per ogni $\phi_1, \phi_2 \in \mathit{FBF}$.
  - $(\phi_1 \wedge \phi_2) \in \mathit{FBF}$ per ogni $\phi_1, \phi_2 \in \mathit{FBF}$.
  - $(\phi_1 \rightarrow \phi_2) \in \mathit{FBF}$ per ogni $\phi_1, \phi_2 \in \mathit{FBF}$.
  - $(\phi_1 \leftrightarrow \phi_2) \in \mathit{FBF}$ per ogni $\phi_1, \phi_2 \in \mathit{FBF}$.

  Pertanto la sintassi è $\phi ::= p \mid (\neg\phi) \mid (\phi \vee \phi) \mid (\phi \wedge \phi) \mid (\phi \rightarrow \phi) \mid (\phi \leftrightarrow \phi)$.
- Esempio: le formule $(p \wedge \vee q)$ e $\neg r$ non sono ben formate, mentre $(p \wedge q)$ e $(\neg r)$ lo sono.

- Teorema: Sia FBF' un linguaggio su $Prop \cup \{\neg, \vee, \wedge, \rightarrow, \leftrightarrow, (,)\}$ definito ponendo $FBF' = \bigcup_{n\in\mathbb{N}} FBF_n$ dove:

$$FBF_n = \begin{cases} Prop & \text{se } n = 0 \\ FBF_{n-1} \cup \{(\neg\phi) \mid \phi \in FBF_{n-1}\} \cup & \text{se } n > 0 \\ \quad \{(\phi_1 \vee \phi_2) \mid \phi_1, \phi_2 \in FBF_{n-1}\} \cup & \\ \quad \{(\phi_1 \wedge \phi_2) \mid \phi_1, \phi_2 \in FBF_{n-1}\} \cup & \\ \quad \{(\phi_1 \rightarrow \phi_2) \mid \phi_1, \phi_2 \in FBF_{n-1}\} \cup & \\ \quad \{(\phi_1 \leftrightarrow \phi_2) \mid \phi_1, \phi_2 \in FBF_{n-1}\} & \end{cases}$$

  Allora $FBF' = FBF$, cioè FBF ammette un'ulteriore caratterizzazione basata sulla massima profondità di annidamento dei connettivi che compaiono nelle sue formule.

- Dimostrazione: Poiché FBF' include $Prop$ ed è chiuso rispetto a $\neg$, $\vee$, $\wedge$, $\rightarrow$, $\leftrightarrow$ (a tal fine è essenziale che FBF_{n-1} faccia parte di FBF_n come si vede per i connettivi binari), dalla minimalità di FBF come insieme che include $Prop$ ed è chiuso rispetto a quegli operatori ricaviamo che $FBF \subseteq FBF'$.
  Sia $\phi \in FBF'$. Dalla definizione di FBF' segue che $\phi \in FBF_n$ per qualche $n \in \mathbb{N}$. Proviamo che da ciò segue che $\phi \in FBF$ procedendo per induzione su n:

  - Sia $n = 0$. In questo caso $\phi \in Prop$ e quindi $\phi \in FBF$.
  - Dato un arbitrario $n \in \mathbb{N}$, supponiamo che per ogni formula $\phi \in FBF_n$ risulti $\phi \in FBF$ e consideriamo una qualsiasi formula $\phi' \in FBF_{n+1}$:
    * Se $\phi' \in FBF_n$ allora $\phi' \in FBF$ per ipotesi induttiva.
    * Se $\phi' \notin FBF_n$ allora ϕ' è della forma $(\neg\phi'')$ con $\phi'' \in FBF_n$ oppure della forma $(\phi''_1 \vee \phi''_2)$, $(\phi''_1 \wedge \phi''_2)$, $(\phi''_1 \rightarrow \phi''_2)$ o $(\phi''_1 \leftrightarrow \phi''_2)$ con $\phi''_1, \phi''_2 \in FBF_n$. Dall'ipotesi induttiva segue che $\phi'' \in FBF$ nel caso della negazione e $\phi''_1, \phi''_2 \in FBF$ nel caso dei connettivi binari. Dalla chiusura di FBF rispetto a tutti connettivi logici segue che $\phi' \in FBF$.

  Di conseguenza $FBF_n \subseteq FBF$ per ogni $n \in \mathbb{N}$ e quindi $FBF' \subseteq FBF$. In conclusione $FBF' = FBF$.

- Date due formule $\phi, \gamma \in FBF$, diciamo che γ è una sottoformula di ϕ sse γ compare in ϕ. Più precisamente, l'insieme $sf(\phi)$ delle sottoformule di ϕ è definito per induzione sulla struttura sintattica di ϕ come segue:

$$sf(\phi) = \{\phi\} \cup \begin{cases} \emptyset & \text{se } \phi \in Prop \\ sf(\phi') & \text{se } \phi \text{ è della forma } (\neg\phi') \\ sf(\phi'_1) \cup sf(\phi'_2) & \text{se } \phi \text{ è della forma } (\phi'_1 \vee \phi'_2),\ (\phi'_1 \wedge \phi'_2),\ (\phi'_1 \rightarrow \phi'_2) \text{ o } (\phi'_1 \leftrightarrow \phi'_2) \end{cases}$$

- Data una formula $\phi \in FBF$, diciamo che essa è:
  - Atomica sse è una proposizione appartenente a $Prop$.
  - Composta con connettivo primario $\neg$ e sottoformula immediata ϕ' sse è della forma $(\neg\phi')$.
  - Composta con connettivo primario $\vee$ e sottoformule immediate ϕ'_1, ϕ'_2 sse è della forma $(\phi'_1 \vee \phi'_2)$.
  - Composta con connettivo primario $\wedge$ e sottoformule immediate ϕ'_1, ϕ'_2 sse è della forma $(\phi'_1 \wedge \phi'_2)$.
  - Composta con connettivo primario $\rightarrow$ e sottoformule immediate ϕ'_1, ϕ'_2 sse è della forma $(\phi'_1 \rightarrow \phi'_2)$.
  - Composta con connettivo primario $\leftrightarrow$ e sottoformule immediate ϕ'_1, ϕ'_2 sse è della forma $(\phi'_1 \leftrightarrow \phi'_2)$.

- L'individuazione del connettivo primario e delle sue sottoformule immediate è la chiave per ricavare tutte le sottoformule di una formula data.

- Esempio: la formula ben formata ϕ data da $(((\neg p) \wedge (q \vee p)) \rightarrow r)$ ha $\rightarrow$ come connettivo primario, $((\neg p) \wedge (q \vee p))$ ed r come sottoformule immediate e poi $sf(\phi) = \{\phi, ((\neg p) \wedge (q \vee p)), r, (\neg p), (q \vee p), p, q\}$ come insieme di sottoformule.

- Teorema (di leggibilità univoca): Le formule ben formate non sono ambigue, cioè esiste un unico modo di leggere ogni $\phi \in FBF$.
- Dimostrazione: Segue dal fatto che, grazie all'uso sistematico delle parentesi, tutte le formule composte hanno un unico connettivo primario.
- L'uso sistematico delle parentesi può essere evitato introducendo delle regole di precedenza e associatività per i connettivi logici. È consuetudine assumere che $\neg$ abbia precedenza su $\wedge$, che $\wedge$ abbia precedenza su $\vee$, che $\vee$ abbia precedenza su $\rightarrow$, che $\rightarrow$ abbia precedenza su $\leftrightarrow$ e che tutti questi connettivi logici siano associativi da sinistra a eccezione di $\neg$, che è associativo da destra.
- Il principio di induzione per $\mathbb{N}$ può essere riformulato per FBF sulla base della struttura sintattica delle formule considerando i loro connettivi primari e le rispettive sottoformule immediate.
- Teorema (principio di induzione strutturale): Sia $\mathcal{Q}$ una proprietà definita su FBF. Se:
  1. $\mathcal{Q}$ è soddisfatta da ogni formula atomica di FBF;
  2. per ogni $\phi \in FBF$, $\mathcal{Q}$ soddisfatta da ϕ implica $\mathcal{Q}$ soddisfatta da $(\neg\phi)$;
  3. per ogni $\phi_1, \phi_2 \in FBF$, $\mathcal{Q}$ soddisfatta da ϕ_1 e $\mathcal{Q}$ soddisfatta da ϕ_2 implicano $\mathcal{Q}$ soddisfatta da $(\phi_1 \vee \phi_2)$;
  4. per ogni $\phi_1, \phi_2 \in FBF$, $\mathcal{Q}$ soddisfatta da ϕ_1 e $\mathcal{Q}$ soddisfatta da ϕ_2 implicano $\mathcal{Q}$ soddisfatta da $(\phi_1 \wedge \phi_2)$;
  5. per ogni $\phi_1, \phi_2 \in FBF$, $\mathcal{Q}$ soddisfatta da ϕ_1 e $\mathcal{Q}$ soddisfatta da ϕ_2 implicano $\mathcal{Q}$ soddisfatta da $(\phi_1 \rightarrow \phi_2)$;
  6. per ogni $\phi_1, \phi_2 \in FBF$, $\mathcal{Q}$ soddisfatta da ϕ_1 e $\mathcal{Q}$ soddisfatta da ϕ_2 implicano $\mathcal{Q}$ soddisfatta da $(\phi_1 \leftrightarrow \phi_2)$;

  allora $\mathcal{Q}$ è soddisfatta da ogni $\phi \in FBF$.
- Dimostrazione: Sia Φ l'insieme delle formule che soddisfano $\mathcal{Q}$. Ovviamente $\Phi \subseteq FBF$. D'altro canto Φ soddisfa le condizioni della definizione di FBF e quindi $FBF \subseteq \Phi$ per la minimalità di FBF come insieme che include $Prop$ ed è chiuso rispetto a tutti i connettivi. In conclusione $\Phi = FBF$.

5.2 Semantica e Intrattabilità della Logica Proposizionale

- Stabilita la sintassi della logica proposizionale, cioè la forma delle sue formule, è necessario definirne la semantica, cioè il significato delle sue formule, in termini di valori di verità. Osservato che il valore di verità di una formula non è un concetto assoluto ma dipende dai valori di verità delle proposizioni presenti in quella formula, descriviamo informalmente il significato dei connettivi logici come segue:
  - $(\neg\phi)$ è vera se ϕ è falsa, mentre è falsa se ϕ è vera.
  - $(\phi_1 \vee \phi_2)$ è vera se almeno una tra ϕ_1 e ϕ_2 è vera, altrimenti è falsa.
  - $(\phi_1 \wedge \phi_2)$ è vera se ϕ_1 e ϕ_2 sono entrambe vere, altrimenti è falsa.
  - $(\phi_1 \rightarrow \phi_2)$ è vera se ϕ_1 e ϕ_2 sono entrambe vere o entrambe false oppure ϕ_2 è vera e ϕ_1 è falsa, altrimenti è falsa. In altri termini, essa è vera se ϕ_1 è falsa oppure ϕ_2 è vera, altrimenti è falsa.
  - $(\phi_1 \leftrightarrow \phi_2)$ è vera se $(\phi_1 \rightarrow \phi_2)$ e $(\phi_2 \rightarrow \phi_1)$ sono entrambe vere, altrimenti è falsa. In altri termini, essa è vera se ϕ_1 e ϕ_2 sono entrambe vere o entrambe false, altrimenti è falsa.
- Al fine di comprendere meglio il significato del connettivo di implicazione, consideriamo la seguente affermazione condizionale: se domani c'è il sole, allora andremo al mare. Chi la pronuncia è un bugiardo – e quindi l'affermazione complessiva è falsa – solo nel caso in cui domani ci sia il sole e non si vada al mare. Qualora domani non ci sia il sole, la decisione di andare al mare oppure no non inficia la verità dell'intera affermazione (ex falso sequitur quodlibet). Il significato intuitivo delle affermazioni condizionali nel linguaggio naturale ci induce a leggere $(\phi_1 \rightarrow \phi_2)$ come il fatto che da ϕ_1 siamo in grado di concludere ϕ_2 *mediante un qualche ragionamento logico*. Questo non è corretto perché $\rightarrow$ è semplicemente un simbolo appartenente al linguaggio della logica proposizionale, mentre i meccanismi di inferenza appartengono ai sistemi deduttivi e quindi si collocano a un livello metalinguistico.

- Per attribuire un valore di verità a ogni proposizione, è necessaria una funzione da *Prop* a $\{vero, falso\}$. Equivalentemente, chiamiamo assegnamento di verità ciascun elemento A di 2^{Prop}, intendendo che una proposizione p è vera sse $p \in A$. Il valore di verità di una formula ben formata può quindi variare a seconda di quanto stabilito dallo specifico assegnamento di verità utilizzato per le proposizioni presenti nella formula stessa.
- La semantica della logica proposizionale viene formalizzata attraverso la relazione di soddisfacimento (o relazione di verità) $\models$ definita come il più piccolo sottoinsieme di $2^{Prop} \times FBF$ tale che:
  - $A \models p$ se $p \in A$.
  - $A \models (\neg\phi)$ se $A \not\models \phi$.
  - $A \models (\phi_1 \vee \phi_2)$ se $A \models \phi_1$ o $A \models \phi_2$.
  - $A \models (\phi_1 \wedge \phi_2)$ se $A \models \phi_1$ e $A \models \phi_2$.
  - $A \models (\phi_1 \rightarrow \phi_2)$ se $A \not\models \phi_1$ oppure $A \models \phi_2$.
  - $A \models (\phi_1 \leftrightarrow \phi_2)$ se $A \models \phi_1$ e $A \models \phi_2$, oppure $A \not\models \phi_1$ e $A \not\models \phi_2$.

  Quando $A \models \phi$ diciamo che A soddisfa ϕ oppure che A è un modello di ϕ.
- Esempi:
  - $\{p\} \models (p \vee q)$.
  - $\{p\} \not\models (p \wedge q)$.
  - $\{p, q\} \not\models ((p \rightarrow q) \rightarrow r)$.
  - $\emptyset \models (p \leftrightarrow q)$.
- Un modo alternativo di definire la semantica della logica proposizionale si basa sulle tabelle di verità. Potremmo chiamarlo la visione dell'ingegnere elettronico in quanto ogni formula ben formata viene vista come un circuito in cui le proposizioni sono i fili in ingresso, i connettivi logici sono le porte e gli assegnamenti di verità determinano il voltaggio lungo ogni filo di ingresso. Questa visione è interessante perché consente di evidenziare delle analogie tra connettivi logici e operazioni e relazioni aritmetiche.
- Indicato con 1 il valore di verità vero e con 0 il valore di verità falso, dato un assegnamento di verità $A \in 2^{Prop}$ definiamo una funzione di interpretazione $\mathcal{I}_A : FBF \rightarrow \{0, 1\}$ coerente con A – cioè tale che, per ogni $p \in Prop$, $\mathcal{I}_A(p) = 1$ sse $p \in A$ – per induzione sulla struttura sintattica delle formule ben formate attraverso la seguente tabella di verità del connettivo logico $\neg$:

$\mathcal{I}_A(\phi)$	$\mathcal{I}_A(\neg\phi)$
1	0
0	1

  e le seguenti tabelle di verità dei connettivi logici binari:

$\mathcal{I}_A(\phi_1)$	$\mathcal{I}_A(\phi_2)$	$\mathcal{I}_A(\phi_1 \vee \phi_2)$	$\mathcal{I}_A(\phi_1 \wedge \phi_2)$	$\mathcal{I}_A(\phi_1 \rightarrow \phi_2)$	$\mathcal{I}_A(\phi_1 \leftrightarrow \phi_2)$
1	1	1	1	1	1
1	0	1	0	0	0
0	1	1	0	1	0
0	0	0	0	1	1

- Osserviamo che per ogni $\phi, \phi_1, \phi_2 \in FBF$ vale che:
  - $\mathcal{I}_A(\neg\phi) = 1 - \mathcal{I}_A(\phi)$.
  - $\mathcal{I}_A(\phi_1 \vee \phi_2) = \max(\mathcal{I}_A(\phi_1), \mathcal{I}_A(\phi_2)) = \min(\mathcal{I}_A(\phi_1) + \mathcal{I}_A(\phi_2), 1)$.
  - $\mathcal{I}_A(\phi_1 \wedge \phi_2) = \min(\mathcal{I}_A(\phi_1), \mathcal{I}_A(\phi_2)) = \mathcal{I}_A(\phi_1) \cdot \mathcal{I}_A(\phi_2)$.
  - $\mathcal{I}_A(\phi_1 \rightarrow \phi_2) = 1$ sse $\mathcal{I}_A(\phi_1) \leq \mathcal{I}_A(\phi_2)$.
  - $\mathcal{I}_A(\phi_1 \leftrightarrow \phi_2) = 1$ sse $\mathcal{I}_A(\phi_1) = \mathcal{I}_A(\phi_2)$.

- Teorema: Per ogni $\phi \in FBF$ e $A \in 2^{Prop}$ risulta $\mathcal{I}_A(\phi) = 1$ sse $A \models \phi$.
- Dimostrazione: Procediamo per induzione sulla struttura sintattica di ϕ:
  - Se $\phi \in Prop$ allora $\mathcal{I}_A(\phi) = 1$ sse $\phi \in A$ sfruttando la coerenza di $\mathcal{I}_A$ con A, cioè sse $A \models \phi$.
  - Sia ϕ della forma $(\neg\phi')$ e supponiamo che $\mathcal{I}_A(\phi')=1$ sse $A \models \phi'$. Allora $\mathcal{I}_A(\phi)=1$ sse $1-\mathcal{I}_A(\phi')=1$ sse $\mathcal{I}_A(\phi')=0$, cioè sse $A \not\models \phi'$ sfruttando l'ipotesi induttiva. Di conseguenza $\mathcal{I}_A(\phi)=1$ sse $A \models \phi$.
  - Sia ϕ della forma $(\phi'_1 \vee \phi'_2)$ e supponiamo che $\mathcal{I}_A(\phi'_i) = 1$ sse $A \models \phi'_i$ per $i \in \{1,2\}$. Allora $\mathcal{I}_A(\phi) = 1$ sse $\mathcal{I}_A(\phi'_1) = 1$ o $\mathcal{I}_A(\phi'_2) = 1$, cioè sse $A \models \phi'_1$ o $A \models \phi'_2$ sfruttando l'ipotesi induttiva. Di conseguenza $\mathcal{I}_A(\phi) = 1$ sse $A \models \phi$.
  - Sia ϕ della forma $(\phi'_1 \wedge \phi'_2)$ e supponiamo che $\mathcal{I}_A(\phi'_i) = 1$ sse $A \models \phi'_i$ per $i \in \{1,2\}$. Allora $\mathcal{I}_A(\phi) = 1$ sse $\mathcal{I}_A(\phi'_1) = 1$ e $\mathcal{I}_A(\phi'_2) = 1$, cioè sse $A \models \phi'_1$ e $A \models \phi'_2$ sfruttando l'ipotesi induttiva. Di conseguenza $\mathcal{I}_A(\phi) = 1$ sse $A \models \phi$.
  - Sia ϕ della forma $(\phi'_1 \rightarrow \phi'_2)$ e supponiamo che $\mathcal{I}_A(\phi'_i) = 1$ sse $A \models \phi'_i$ per $i \in \{1,2\}$. Allora $\mathcal{I}_A(\phi) = 1$ sse $\mathcal{I}_A(\phi'_1) = 0$ oppure $\mathcal{I}_A(\phi'_2) = 1$, cioè sse $A \not\models \phi'_1$ oppure $A \models \phi'_2$ sfruttando l'ipotesi induttiva. Di conseguenza $\mathcal{I}_A(\phi) = 1$ sse $A \models \phi$.
  - Sia ϕ della forma $(\phi'_1 \leftrightarrow \phi'_2)$ e supponiamo che $\mathcal{I}_A(\phi'_i) = 1$ sse $A \models \phi'_i$ per $i \in \{1,2\}$. Allora $\mathcal{I}_A(\phi) = 1$ sse $\mathcal{I}_A(\phi'_1) = 1$ e $\mathcal{I}_A(\phi'_2) = 1$ oppure $\mathcal{I}_A(\phi'_1) = 0$ e $\mathcal{I}_A(\phi'_2) = 0$, cioè sse $A \models \phi'_1$ e $A \models \phi'_2$ oppure $A \not\models \phi'_1$ e $A \not\models \phi'_2$ sfruttando l'ipotesi induttiva. Di conseguenza $\mathcal{I}_A(\phi)=1$ sse $A \models \phi$.
- Un altro modo alternativo di definire la semantica della logica proposizionale si basa sull'individuazione di tutti i possibili modelli per ogni formula ben formata. Potremmo chiamarlo la visione dell'ingegnere del software in quanto questi è interessato a descrivere tutti i possibili scenari in cui il sistema software che ha sviluppato funziona correttamente. Questa visione è interessante perché consente di evidenziare delle analogie tra connettivi logici e operazioni insiemistiche.
- La funzione $modelli : FBF \rightarrow 2^{2^{Prop}}$ è definita per induzione sulla struttura sintattica delle formule ben formate come segue (dove il complementare è preso rispetto a 2^{Prop}):
  - $modelli(p) = \{A \in 2^{Prop} \mid p \in A\}$.
  - $modelli(\neg\phi) = \complement(modelli(\phi))$.
  - $modelli(\phi_1 \vee \phi_2) = modelli(\phi_1) \cup modelli(\phi_2)$.
  - $modelli(\phi_1 \wedge \phi_2) = modelli(\phi_1) \cap modelli(\phi_2)$.
  - $modelli(\phi_1 \rightarrow \phi_2) = \complement(modelli(\phi_1)) \cup modelli(\phi_2)$.
  - $modelli(\phi_1 \leftrightarrow \phi_2) = (modelli(\phi_1) \cap modelli(\phi_2)) \cup (\complement(modelli(\phi_1)) \cap \complement(modelli(\phi_2)))$.
- Teorema: Per ogni $\phi \in FBF$ e $A \in 2^{Prop}$ risulta $A \in modelli(\phi)$ sse $A \models \phi$.
- Dimostrazione: Procediamo per induzione sulla struttura sintattica di ϕ:
  - Se $\phi \in Prop$ allora $A \in modelli(\phi)$ sse $\phi \in A$, cioè sse $A \models \phi$.
  - Sia ϕ della forma $(\neg\phi')$ e supponiamo che $A \in modelli(\phi')$ sse $A \models \phi'$. Allora $A \in modelli(\phi) = 2^{Prop} \setminus modelli(\phi')$ sse $A \not\models \phi'$ sfruttando l'ipotesi induttiva. Di conseguenza $A \in modelli(\phi)$ sse $A \models \phi$.
  - Sia ϕ della forma $(\phi'_1 \vee \phi'_2)$ e supponiamo che $A \in modelli(\phi'_i)$ sse $A \models \phi'_i$ per $i \in \{1,2\}$. Allora $A \in modelli(\phi)$ sse $A \in modelli(\phi'_1)$ o $A \in modelli(\phi'_2)$, cioè sse $A \models \phi'_1$ o $A \models \phi'_2$ sfruttando l'ipotesi induttiva. Di conseguenza $A \in modelli(\phi)$ sse $A \models \phi$.
  - Sia ϕ della forma $(\phi'_1 \wedge \phi'_2)$ e supponiamo che $A \in modelli(\phi'_i)$ sse $A \models \phi'_i$ per $i \in \{1,2\}$. Allora $A \in modelli(\phi)$ sse $A \in modelli(\phi'_1)$ e $A \in modelli(\phi'_2)$, cioè sse $A \models \phi'_1$ e $A \models \phi'_2$ sfruttando l'ipotesi induttiva. Di conseguenza $A \in modelli(\phi)$ sse $A \models \phi$.
  - Sia ϕ della forma $(\phi'_1 \rightarrow \phi'_2)$ e supponiamo che $A \in modelli(\phi'_i)$ sse $A \models \phi'_i$ per $i \in \{1,2\}$. Allora $A \in modelli(\phi)$ sse $A \in 2^{Prop} \setminus modelli(\phi'_1)$ oppure $A \in modelli(\phi'_2)$, cioè sse $A \not\models \phi'_1$ oppure $A \models \phi'_2$ sfruttando l'ipotesi induttiva. Di conseguenza $A \in modelli(\phi)$ sse $A \models \phi$.
  - Sia ϕ della forma $(\phi'_1 \leftrightarrow \phi'_2)$ e supponiamo che $A \in modelli(\phi'_i)$ sse $A \models \phi'_i$ per $i \in \{1,2\}$. Allora $A \in modelli(\phi)$ sse $A \in modelli(\phi'_1)$ e $A \in modelli(\phi'_2)$ oppure $A \in 2^{Prop} \setminus modelli(\phi'_1)$ e $A \in 2^{Prop} \setminus modelli(\phi'_2)$, cioè sse $A \models \phi'_1$ e $A \models \phi'_2$ oppure $A \not\models \phi'_1$ e $A \not\models \phi'_2$ sfruttando l'ipotesi induttiva. Di conseguenza $A \in modelli(\phi)$ sse $A \models \phi$.

- Data una formula $\phi \in FBF$, il suo valore di verità non dipende da ciò che non sta nell'insieme $sfa(\phi)$ delle sue sottoformule atomiche, il quale è definito per induzione sulla struttura sintattica di ϕ come segue:
$$sfa(\phi) = \begin{cases} \{\phi\} & \text{se } \phi \in Prop \\ sfa(\phi') & \text{se } \phi \text{ è della forma } (\neg\phi') \\ sfa(\phi'_1) \cup sfa(\phi'_2) & \text{se } \phi \text{ è della forma } (\phi'_1 \vee \phi'_2),\ (\phi'_1 \wedge \phi'_2),\ (\phi'_1 \rightarrow \phi'_2) \text{ o } (\phi'_1 \leftrightarrow \phi'_2) \end{cases}$$

- Teorema: Siano $\phi \in FBF$ e $A_1, A_2 \in 2^{Prop}$. Se per ogni $p \in sfa(\phi)$ risulta che $p \in A_1$ sse $p \in A_2$, allora $A_1 \models \phi$ sse $A_2 \models \phi$.

- Dimostrazione: Procediamo per induzione sulla struttura sintattica di ϕ:
  - Se $\phi \in Prop$ allora $\phi \in sfa(\phi)$ e quindi $\phi \in A_1$ sse $\phi \in A_2$. Perciò $A_1 \models \phi$ sse $A_2 \models \phi$.
  - Sia ϕ della forma $(\neg\phi')$ e supponiamo che $A_1 \models \phi'$ sse $A_2 \models \phi'$. Allora $A_1 \models \phi$ sse $A_1 \not\models \phi'$, cioè sse $A_2 \not\models \phi'$ sfruttando l'ipotesi induttiva, cioè sse $A_2 \models \phi$.
  - Sia ϕ della forma $(\phi'_1 \vee \phi'_2)$ e supponiamo che $A_1 \models \phi'_i$ sse $A_2 \models \phi'_i$ per $i \in \{1, 2\}$. Allora $A_1 \models \phi$ sse $A_1 \models \phi'_1$ o $A_1 \models \phi'_2$, cioè sse $A_2 \models \phi'_1$ o $A_2 \models \phi'_2$ sfruttando l'ipotesi induttiva, cioè sse $A_2 \models \phi$.
  - Sia ϕ della forma $(\phi'_1 \wedge \phi'_2)$ e supponiamo che $A_1 \models \phi'_i$ sse $A_2 \models \phi'_i$ per $i \in \{1, 2\}$. Allora $A_1 \models \phi$ sse $A_1 \models \phi'_1$ e $A_1 \models \phi'_2$, cioè sse $A_2 \models \phi'_1$ e $A_2 \models \phi'_2$ sfruttando l'ipotesi induttiva, cioè sse $A_2 \models \phi$.
  - Sia ϕ della forma $(\phi'_1 \rightarrow \phi'_2)$ e supponiamo che $A_1 \models \phi'_i$ sse $A_2 \models \phi'_i$ per $i \in \{1, 2\}$. Allora $A_1 \models \phi$ sse $A_1 \not\models \phi'_1$ oppure $A_1 \models \phi'_2$, cioè sse $A_2 \not\models \phi'_1$ e $A_2 \models \phi'_2$ sfruttando l'ipotesi induttiva, cioè sse $A_2 \models \phi$.
  - Sia ϕ della forma $(\phi'_1 \leftrightarrow \phi'_2)$ e supponiamo che $A_1 \models \phi'_i$ sse $A_2 \models \phi'_i$ per $i \in \{1, 2\}$. Allora $A_1 \models \phi$ sse $A_1 \models \phi'_1$ e $A_1 \models \phi'_2$ oppure $A_1 \not\models \phi'_1$ e $A_1 \not\models \phi'_2$, cioè sse $A_2 \models \phi'_1$ e $A_2 \models \phi'_2$ oppure $A_2 \not\models \phi'_1$ e $A_2 \not\models \phi'_2$ sfruttando l'ipotesi induttiva, cioè sse $A_2 \models \phi$.

- Data una formula $\phi \in FBF$, diciamo che essa è:
  - Soddisfacibile sse esiste $A \in 2^{Prop}$ tale che $A \models \phi$.
  - Valida, ovvero una tautologia, sse $A \models \phi$ per ogni $A \in 2^{Prop}$.
  - Insoddisfacibile, ovvero una contraddizione, sse $A \not\models \phi$ per ogni $A \in 2^{Prop}$.

- Teorema: Sia $\phi \in FBF$. Allora:
  - ϕ è una tautologia sse $(\neg\phi)$ è una contraddizione.
  - ϕ non è una tautologia sse $(\neg\phi)$ è soddisfacibile.
  - ϕ è soddisfacibile sse ϕ non è una contraddizione.

- Dimostrazione:
  - ϕ è una tautologia sse $A \models \phi$ per ogni $A \in 2^{Prop}$, cioè sse $A \not\models (\neg\phi)$ per ogni $A \in 2^{Prop}$, cioè sse $(\neg\phi)$ è una contraddizione.
  - ϕ non è una tautologia sse non è vero che $A \models \phi$ per ogni $A \in 2^{Prop}$, cioè sse esiste $A \in 2^{Prop}$ tale che $A \not\models \phi$, cioè sse esiste $A \in 2^{Prop}$ tale che $A \models (\neg\phi)$, cioè sse $(\neg\phi)$ è soddisfacibile.
  - ϕ è soddisfacibile sse esiste $A \in 2^{Prop}$ tale che $A \models \phi$, cioè sse non è vero che $A \not\models \phi$ per ogni $A \in 2^{Prop}$, cioè sse ϕ non è una contraddizione.

- Esempi di tautologie e contraddizioni (per semplicità omettiamo la maggior parte delle parentesi):

p	q	$\neg p$	$\neg q$	$p \rightarrow q$ $\neg p \vee q$ $\neg q \rightarrow \neg p$	$p \vee \neg p$	$q \wedge \neg q$	$(p \rightarrow q) \leftrightarrow (\neg p \vee q)$	$(p \rightarrow q) \leftrightarrow (\neg q \rightarrow \neg p)$
1	1	0	0	1	1	0	1	1
1	0	0	1	0	1	0	1	1
0	1	1	0	1	1	0	1	1
0	0	1	1	1	1	0	1	1

- Dati $\phi \in FBF$ e $A \in 2^{Prop}$, è possibile verificare se $A \models \phi$ impiegando un tempo che cresce linearmente con il numero di occorrenze di proposizioni e connettivi logici in ϕ. L'algoritmo consiste nell'applicare la definizione di $\models$ individuando a ogni passo il connettivo primario e le sue sottoformule immediate.
- Data $\phi \in FBF$, un algoritmo per decidere se ϕ è soddisfacibile, una tautologia o una contraddizione consiste nel costruire la corrispondente tabella di verità e nel verificare se la sua ultima colonna contiene almeno un 1, tutti 1 o tutti 0, rispettivamente. Il tempo di esecuzione di questo algoritmo cresce esponenzialmente al crescere di $|sfa(\phi)|$ perché la tabella di verità ha $2^{|sfa(\phi)|}$ righe, che sono tante quante le interpretazioni coerenti con le classi di assegnamenti di verità che coincidono sull'insieme $sfa(\phi)$. Poiché molti problemi possono essere ricondotti a quello della soddisfacibilità, una delle più grandi questioni aperte nella teoria della complessità computazionale è quella di stabilire se esiste un algoritmo che, diversamente dal precedente, sia in grado di risolvere il problema della soddisfacibilità in tempo polinomiale, così da rendere questo e gli altri problemi computazionalmente trattabili. ■fltplf_11

5.3 Conseguenza ed Equivalenza nella Logica Proposizionale

- Sulla base della semantica della logica proposizionale, possiamo introdurre una nozione d'ordine e una nozione d'equivalenza sull'insieme FBF. La seconda nozione è particolarmente utile per stabilire quando due formule sintatticamente diverse hanno lo stesso valore di verità.
- Dato un insieme di formule $\Phi \in 2^{FBF}$, denotiamo con $modelli(\Phi) = \bigcap_{\phi\in\Phi} modelli(\phi)$ l'insieme degli assegnamenti di verità che soddisfano tutte le formule di Φ. Per $\Phi = \emptyset$, poniamo $modelli(\emptyset) = 2^{Prop}$.
- La relazione di soddisfacimento $\models$ può essere trasformata in una relazione su 2^{FBF} detta relazione di conseguenza logica ponendo $\Phi \models \Gamma$ (o $\Phi \Rightarrow \Gamma$) sse $modelli(\Phi) \subseteq modelli(\Gamma)$ per ogni $\Phi, \Gamma \in 2^{FBF}$. In altri termini, Γ è una conseguenza logica di Φ sse ogni assegnamento di verità che soddisfa tutte le formule di Φ soddisfa anche tutte le formule di Γ.
- Il fatto che Γ sia una conseguenza logica di Φ significa che, qualora ci si trovi in una condizione che soddisfa tutte le formule di Φ, allora anche tutte le formule di Γ sono soddisfatte in quella condizione senza doverlo verificare. In particolare, $\models \Gamma$ significa che tutte le formule di Γ sono delle tautologie perché $modelli(\emptyset) = 2^{Prop} \subseteq modelli(\Gamma)$.
- La relazione di conseguenza logica è riflessiva e transitiva – perché tale è la relazione di inclusione insiemistica – ma non è una relazione d'ordine parziale. Ad esempio $\{(p \to q)\} \neq \{((\neg p) \vee q)\}$ ma $modelli(\{(p \to q)\}) = \{A \in 2^{Prop} \mid p \notin A \text{ oppure } q \in A\} = modelli(\{((\neg p) \vee q)\})$. Inoltre la relazione $\models$ (o $\Rightarrow$) è strettamente legata al connettivo $\to$.
- Teorema (di conseguenza logica): Dato $n \in \mathbb{N}$, siano $\Phi = \{\phi_1, \ldots, \phi_n\} \in 2^{FBF}$ e $\gamma \in FBF$. Allora $\Phi \models \{\gamma\}$ sse $((\bigwedge_{i=1}^{n} \phi_i) \to \gamma)$ è una tautologia, ovvero sse $((\bigwedge_{i=1}^{n} \phi_i) \wedge (\neg\gamma))$ è una contraddizione.
- Dimostrazione: In primo luogo proviamo che da $\Phi \models \{\gamma\}$ segue che $((\bigwedge_{i=1}^{n} \phi_i) \to \gamma)$ è una tautologia. Supponiamo che esista $A \in 2^{Prop}$ tale che $A \not\models ((\bigwedge_{i=1}^{n} \phi_i) \to \gamma)$. Ciò significa che $A \models (\bigwedge_{i=1}^{n} \phi_i)$ e $A \not\models \gamma$. Dunque esiste $A \in 2^{Prop}$ tale che $A \in modelli(\Phi)$ e $A \notin modelli(\gamma)$, che contraddice l'ipotesi $\Phi \models \{\gamma\}$. In secondo luogo proviamo che dal fatto che $((\bigwedge_{i=1}^{n} \phi_i) \to \gamma)$ sia una tautologia segue che $((\bigwedge_{i=1}^{n} \phi_i) \wedge (\neg\gamma))$ è una contraddizione. Supponiamo che esista $A \in 2^{Prop}$ tale che $A \models ((\bigwedge_{i=1}^{n} \phi_i) \wedge (\neg\gamma))$. Ciò significa che $A \models (\bigwedge_{i=1}^{n} \phi_i)$ e $A \not\models \gamma$. Dunque esiste $A \in 2^{Prop}$ tale che $A \not\models ((\bigwedge_{i=1}^{n} \phi_i) \to \gamma)$, che contraddice l'ipotesi secondo cui $((\bigwedge_{i=1}^{n} \phi_i) \to \gamma)$ è una tautologia.

  In terzo luogo proviamo che dal fatto che $((\bigwedge_{i=1}^{n} \phi_i) \wedge (\neg\gamma))$ sia una contraddizione segue che $\Phi \models \{\gamma\}$. Supponiamo che esista $A \in 2^{Prop}$ tale che $A \in modelli(\Phi)$ e $A \notin modelli(\gamma)$. Ciò significa che $A \models (\bigwedge_{i=1}^{n} \phi_i)$ e $A \not\models \gamma$. Dunque esiste $A \in 2^{Prop}$ tale che $A \models ((\bigwedge_{i=1}^{n} \phi_i) \wedge (\neg\gamma))$, che contraddice l'ipotesi secondo cui $((\bigwedge_{i=1}^{n} \phi_i) \wedge (\neg\gamma))$ è una contraddizione.
- Corollario: Siano $\phi, \gamma \in FBF$. Allora $\{\phi\} \models \{\gamma\}$ sse $\models \{(\phi \to \gamma)\}$.

- La relazione di equivalenza logica è una relazione su 2^{FBF} definita ponendo $\Phi \equiv \Gamma$ (o $\Phi \Leftrightarrow \Gamma$) sse $\Phi \models \Gamma$ e $\Gamma \models \Phi$. In altri termini, Φ e Γ sono logicamente equivalenti sse $modelli(\Phi) = modelli(\Gamma)$. Questa relazione è una relazione d'equivalenza, cioè è riflessiva, simmetrica e transitiva. Inoltre la relazione $\equiv$ (o $\Leftrightarrow$) è strettamente legata al connettivo $\leftrightarrow$.
- Teorema (di equivalenza logica): Dato $n \in \mathbb{N}$, siano $\Phi = \{\phi_1, \ldots, \phi_n\} \in 2^{FBF}$ e $\gamma \in FBF$. Allora $\Phi \equiv \{\gamma\}$ sse $((\bigwedge_{i=1}^{n} \phi_i) \leftrightarrow \gamma)$ è una tautologia.
- Dimostrazione: $\Phi \equiv \{\gamma\}$ sse $modelli(\Phi) = modelli(\{\gamma\})$, cioè sse per ogni $A \in 2^{Prop}$ risulta $A \models (\bigwedge_{i=1}^{n} \phi_i)$ e $A \models \gamma$ oppure $A \not\models (\bigwedge_{i=1}^{n} \phi_i)$ e $A \not\models \gamma$, cioè sse per ogni $A \in 2^{Prop}$ risulta $A \models ((\bigwedge_{i=1}^{n} \phi_i) \leftrightarrow \gamma)$, cioè sse $((\bigwedge_{i=1}^{n} \phi_i) \leftrightarrow \gamma)$ è una tautologia.
- Corollario: Siano $\phi, \gamma \in FBF$. Allora $\{\phi\} \equiv \{\gamma\}$ sse $\models \{(\phi \leftrightarrow \gamma)\}$.
- Il corollario al teorema di equivalenza logica fornisce una caratterizzazione alternativa di $\{\phi\} \equiv \{\gamma\}$ che è utile dal punto di vista computazionale, nel senso che invece di costruire $modelli(\{\phi\})$ e $modelli(\{\gamma\})$, che potrebbero essere insiemi infiniti, basta verificare che $(\phi \leftrightarrow \gamma)$ sia una tautologia attraverso la costruzione della sua tabella di verità.
- La relazione di equivalenza logica ristretta a FBF, cioè a insiemi singoletto (che verranno da ora in poi denotati senza parentesi graffe), risulta essere una congruenza rispetto a tutti i connettivi logici. Questo è molto importante perché consente di manipolare composizionalmente le formule. In altri termini, data una formula ben formata, qualora una sua sottoformula venga sostituita con un'altra formula a essa logicamente equivalente, la formula ben formata risultante è logicamente equivalente a quella originaria. Dunque la sostituzione di sottoformule con formule logicamente equivalenti a esse non altera il valore di verità della formula ben formata originaria.
- Teorema (di sostituzione): Siano $\phi_1, \phi_2 \in FBF$. Se $\phi_1 \equiv \phi_2$ allora:
  - $(\neg\phi_1) \equiv (\neg\phi_2)$.
  - $(\phi_1 \vee \gamma) \equiv (\phi_2 \vee \gamma)$ e $(\gamma \vee \phi_1) \equiv (\gamma \vee \phi_2)$ per ogni $\gamma \in FBF$.
  - $(\phi_1 \wedge \gamma) \equiv (\phi_2 \wedge \gamma)$ e $(\gamma \wedge \phi_1) \equiv (\gamma \wedge \phi_2)$ per ogni $\gamma \in FBF$.
  - $(\phi_1 \rightarrow \gamma) \equiv (\phi_2 \rightarrow \gamma)$ e $(\gamma \rightarrow \phi_1) \equiv (\gamma \rightarrow \phi_2)$ per ogni $\gamma \in FBF$.
  - $(\phi_1 \leftrightarrow \gamma) \equiv (\phi_2 \leftrightarrow \gamma)$ e $(\gamma \leftrightarrow \phi_1) \equiv (\gamma \leftrightarrow \phi_2)$ per ogni $\gamma \in FBF$.
- Dimostrazione: Siano $\phi_1, \phi_2 \in FBF$ tali che $\phi_1 \equiv \phi_2$ e quindi $modelli(\phi_1) = modelli(\phi_2)$:
  - $(\neg\phi_1) \equiv (\neg\phi_2)$ segue da $modelli(\neg\phi_1)) = 2^{Prop} \setminus modelli(\phi_1) = 2^{Prop} \setminus modelli(\phi_2) = modelli(\neg\phi_2))$.
  - $(\phi_1 \vee \gamma) \equiv (\phi_2 \vee \gamma)$ segue da $modelli(\phi_1 \vee \gamma) = modelli(\phi_1) \cup modelli(\gamma) = modelli(\phi_2) \cup modelli(\gamma) = modelli(\phi_2 \vee \gamma)$. La dimostrazione di $(\gamma \vee \phi_1) \equiv (\gamma \vee \phi_2)$ è simile.
  - $(\phi_1 \wedge \gamma) \equiv (\phi_2 \wedge \gamma)$ segue da $modelli(\phi_1 \wedge \gamma) = modelli(\phi_1) \cap modelli(\gamma) = modelli(\phi_2) \cap modelli(\gamma) = modelli(\phi_2 \wedge \gamma)$. La dimostrazione di $(\gamma \wedge \phi_1) \equiv (\gamma \wedge \phi_2)$ è simile.
  - $(\phi_1 \rightarrow \gamma) \equiv (\phi_2 \rightarrow \gamma)$ segue da $modelli(\phi_1 \rightarrow \gamma) = (2^{Prop} \setminus modelli(\phi_1)) \cup modelli(\gamma) = (2^{Prop} \setminus modelli(\phi_2)) \cup modelli(\gamma) = modelli(\phi_2 \rightarrow \gamma)$. La dimostrazione di $(\gamma \rightarrow \phi_1) \equiv (\gamma \rightarrow \phi_2)$ è simile.
  - $(\phi_1 \leftrightarrow \gamma) \equiv (\phi_2 \leftrightarrow \gamma)$ segue da $modelli(\phi_1 \leftrightarrow \gamma) = (modelli(\phi_1) \cap modelli(\gamma)) \cup ((2^{Prop} \setminus modelli(\phi_1)) \cap (2^{Prop} \setminus modelli(\gamma))) = (modelli(\phi_2) \cap modelli(\gamma)) \cup ((2^{Prop} \setminus modelli(\phi_2)) \cap (2^{Prop} \setminus modelli(\gamma))) = modelli(\phi_2 \leftrightarrow \gamma)$. La dimostrazione di $(\gamma \leftrightarrow \phi_1) \equiv (\gamma \leftrightarrow \phi_2)$ è simile.
- L'equivalenza logica $\equiv$ è un concetto semantico che non va confuso con l'uguaglianza sintattica $=$. Formule ben formate sintatticamente diverse possono essere logicamente equivalenti, come ad esempio $(p \rightarrow q)$, $(\neg p \vee q)$, $(\neg q \rightarrow \neg p)$.

5.4 Proprietà Algebriche dei Connettivi Logici

- Teorema: Siano $\mathbb{O}$ una qualsiasi contraddizione e $\mathbb{I}$ una qualsiasi tautologia. La struttura algebrica $(FBF, \vee, \wedge, \neg, \mathbb{O}, \mathbb{I})$ è un reticolo booleano, cioè soddisfa le seguenti leggi:
  - Commutatività:
    - $(\phi \vee \gamma) \equiv (\gamma \vee \phi)$.
    - $(\phi \wedge \gamma) \equiv (\gamma \wedge \phi)$.
  - Associatività:
    - $((\phi \vee \gamma) \vee \delta) \equiv (\phi \vee (\gamma \vee \delta))$.
    - $((\phi \wedge \gamma) \wedge \delta) \equiv (\phi \wedge (\gamma \wedge \delta))$.
  - Assorbimento:
    - $(\phi \vee (\phi \wedge \gamma)) \equiv \phi$.
    - $(\phi \wedge (\phi \vee \gamma)) \equiv \phi$.
  - Distributività:
    - $(\phi \vee (\gamma \wedge \delta)) \equiv ((\phi \vee \gamma) \wedge (\phi \vee \delta))$.
    - $(\phi \wedge (\gamma \vee \delta)) \equiv ((\phi \wedge \gamma) \vee (\phi \wedge \delta))$.
  - Elemento neutro:
    - $(\phi \vee \mathbb{O}) \equiv \phi$.
    - $(\phi \wedge \mathbb{I}) \equiv \phi$.
  - Elemento inverso:
    - $(\phi \vee (\neg\phi)) \equiv \mathbb{I}$.
    - $(\phi \wedge (\neg\phi)) \equiv \mathbb{O}$.
- Dimostrazione: In virtù del corollario al teorema di equivalenza logica, per ogni legge è sufficiente verificare che la biimplicazione tra la formula di sinistra e la formula di destra è una tautologia costruendo la corrispondente tabella di verità e controllando che la sua ultima colonna contiene tutti 1.
- Teorema: La struttura algebrica $(FBF, \vee, \wedge, \neg, \mathbb{O}, \mathbb{I})$ possiede inoltre le seguenti proprietà derivate:
  - Elemento assorbente:
    - $(\phi \vee \mathbb{I}) \equiv \mathbb{I}$.
    - $(\phi \wedge \mathbb{O}) \equiv \mathbb{O}$.
  - Idempotenza:
    - $(\phi \vee \phi) \equiv \phi$.
    - $(\phi \wedge \phi) \equiv \phi$.
  - Doppia inversione:
    - $(\neg(\neg\phi)) \equiv \phi$.
  - Leggi di De Morgan:
    - $(\neg(\phi \vee \gamma)) \equiv ((\neg\phi) \wedge (\neg\gamma))$.
    - $(\neg(\phi \wedge \gamma)) \equiv ((\neg\phi) \vee (\neg\gamma))$.
- Dimostrazione: In virtù del corollario al teorema di equivalenza logica, per ogni proprietà è sufficiente verificare che la biimplicazione tra la formula di sinistra e la formula di destra è una tautologia costruendo la corrispondente tabella di verità e controllando che la sua ultima colonna contiene tutti 1.

- Poiché $(\phi_1 \vee \phi_2)$ è falsa sse ϕ_1 e ϕ_2 sono entrambe false mentre $(\phi_1 \wedge \phi_2)$ è vera sse ϕ_1 e ϕ_2 sono entrambe vere, i due connettivi logici binari del reticolo booleano sono operazioni duali, cioè sono derivabili l'uno dall'altro scambiando i ruoli tra vero e falso. Peraltro, applicando la negazione ad ambo i membri delle leggi di De Morgan e sfruttando il teorema di sostituzione, ricaviamo che $(\phi \vee \gamma) \equiv (\neg((\neg\phi) \wedge (\neg\gamma)))$ e $(\phi \wedge \gamma) \equiv (\neg((\neg\phi) \vee (\neg\gamma)))$ grazie alla doppia inversione applicata al membro sulla sinistra.
- In generale, dati un insieme C di connettivi logici e un connettivo logico $\odot \notin C$, diciamo che $\odot$ è semanticamente derivabile da C sse ogni formula ben formata di cui $\odot$ è il connettivo primario è equivalente a una formula ben formata i cui connettivi logici appartengono tutti a C.
- Un insieme C di connettivi logici è funzionalmente completo sse ogni altro possibile connettivo logico è semanticamente derivabile da C. Oltre ai quattro connettivi logici binari, è possibile introdurne altri, stabilire delle leggi di derivabilità e individuare degli insiemi funzionalmente completi.
- Presentiamo le tabelle di verità di tre ulteriori connettivi logici binari chiamati xor $\veebar$ ("o ... o ..." come l'"aut" latino), nor $\downarrow$ ("né ... né ...") e nand $\uparrow$ ("... o ... ma non entrambi"):

$\mathcal{I}_A(\phi_1)$	$\mathcal{I}_A(\phi_2)$	$\mathcal{I}_A(\phi_1 \veebar \phi_2)$	$\mathcal{I}_A(\phi_1 \downarrow \phi_2)$	$\mathcal{I}_A(\phi_1 \uparrow \phi_2)$
1	1	0	0	0
1	0	1	0	1
0	1	1	0	1
0	0	0	1	1

- Teorema: Valgono le seguenti leggi di derivabilità semantica per i connettivi logici diversi da $\vee$, $\wedge$, $\neg$:
  - $(\phi \to \gamma) \equiv ((\neg\phi) \vee \gamma)$.
  - $(\phi \leftrightarrow \gamma) \equiv ((\phi \to \gamma) \wedge (\gamma \to \phi))$.
  - $(\phi \leftrightarrow \gamma) \equiv ((\phi \wedge \gamma) \vee ((\neg\phi) \wedge (\neg\gamma)))$.
  - $(\phi \leftrightarrow \gamma) \equiv (\neg(\phi \veebar \gamma))$.
  - $(\phi \veebar \gamma) \equiv (\neg(\phi \leftrightarrow \gamma))$.
  - $(\phi \veebar \gamma) \equiv ((\phi \wedge (\neg\gamma)) \vee ((\neg\phi) \wedge \gamma))$.
  - $(\phi \downarrow \gamma) \equiv (\neg(\phi \vee \gamma))$.
  - $(\phi \downarrow \gamma) \equiv ((\neg\phi) \wedge (\neg\gamma))$.
  - $(\phi \uparrow \gamma) \equiv (\neg(\phi \wedge \gamma))$.
  - $(\phi \uparrow \gamma) \equiv ((\neg\phi) \vee (\neg\gamma))$.

  Inoltre valgono le seguenti leggi di derivabilità semantica per i connettivi logici $\vee$, $\wedge$, $\neg$:
  - $(\phi \vee \gamma) \equiv (\neg((\neg\phi) \wedge (\neg\gamma)))$.
  - $(\phi \vee \gamma) \equiv ((\neg\phi) \to \gamma)$.
  - $(\phi \vee \gamma) \equiv (\neg(\phi \downarrow \gamma)) \equiv ((\phi \downarrow \gamma) \downarrow (\phi \downarrow \gamma))$.
  - $(\phi \vee \gamma) \equiv ((\neg\phi) \uparrow (\neg\gamma)) \equiv ((\phi \uparrow \phi) \uparrow (\gamma \uparrow \gamma))$.
  - $(\phi \wedge \gamma) \equiv (\neg((\neg\phi) \vee (\neg\gamma)))$.
  - $(\phi \wedge \gamma) \equiv ((((\phi \to \mathbb{O}) \to \mathbb{O}) \to (\gamma \to \mathbb{O})) \to \mathbb{O})$.
  - $(\phi \wedge \gamma) \equiv ((\neg\phi) \downarrow (\neg\gamma)) \equiv ((\phi \downarrow \phi) \downarrow (\gamma \downarrow \gamma))$.
  - $(\phi \wedge \gamma) \equiv (\neg(\phi \uparrow \gamma)) \equiv ((\phi \uparrow \gamma) \uparrow (\phi \uparrow \gamma))$.
  - $(\neg\phi) \equiv (\phi \to \mathbb{O})$.
  - $(\neg\phi) \equiv (\phi \downarrow \phi)$.
  - $(\neg\phi) \equiv (\phi \uparrow \phi)$.
- Dimostrazione: In virtù del corollario al teorema di equivalenza logica, per ogni legge è sufficiente verificare che la biimplicazione tra la formula di sinistra e la formula di destra è una tautologia costruendo la corrispondente tabella di verità e controllando che la sua ultima colonna contiene tutti 1.

- Corollario: I seguenti insiemi di connettivi logici sono funzionalmente completi:
  - $\{\vee, \wedge, \neg\}$.
  - $\{\vee, \neg\}$.
  - $\{\wedge, \neg\}$.
  - $\{\downarrow\}$.
  - $\{\uparrow\}$.

5.5 Sistemi Deduttivi per la Logica Proposizionale

- La logica matematica si fonda su un livello linguistico e un livello metalinguistico. Fissata la sintassi, il primo livello è formato da una semantica attraverso cui viene attribuito un valore di verità a ogni formula. Il secondo livello comprende invece meccanismi di ragionamento per stabilire la verità di una formula deducendola da altre formule che sono state poste essere vere o dedotte essere vere, *attraverso pure manipolazioni simboliche*. Qual è il legame tra i due livelli?
- I connettivi logici che danno luogo alla sintassi della logica proposizionale sono funzionalmente completi e concorrono a formare gli enunciati dei teoremi della matematica. Tali teoremi sono di solito espressi nella forma $(\phi \to \gamma)$ oppure $(\phi \leftrightarrow \gamma)$ e rappresentano delle tautologie, cioè $\phi \Rightarrow \gamma$ oppure $\phi \Leftrightarrow \gamma$.
- Dato un teorema della forma $(\phi \to \gamma)$, diciamo che:
  - ϕ è condizione sufficiente per γ, nel senso che affinché γ sia vera è sufficiente che ϕ sia vera; se ϕ è falsa non possiamo concludere nulla su γ.
  - γ è condizione necessaria per ϕ, nel senso che – sfruttando la proprietà $(\phi \to \gamma) \equiv ((\neg\gamma) \to (\neg\phi))$ – affinché ϕ sia vera è necessario che γ sia vera; se γ è falsa sicuramente anche ϕ è falsa altrimenti il teorema non vale.
- Dato un teorema della forma $(\phi \leftrightarrow \gamma)$, diciamo che γ è condizione necessaria e sufficiente per ϕ, perché basta che una delle due formule sia vera (rispettivamente falsa) per concludere che anche l'altra lo è.
- Per dimostrare un teorema della matematica di solito non si ragiona in termini semantici mediante funzioni di interpretazione basate su tabelle di verità o insiemi di modelli (teoria dei modelli), ma si procede per deduzione mediante l'applicazione di regole di inferenza a risultati che sono già noti (teoria della dimostrazione).
- Un teorema della forma $(\phi \to \gamma)$ può essere dimostrato:
  - In modo diretto, cioè assumendo che ϕ sia vera e cercando di derivare che anche γ è vera; se ϕ fosse falsa allora $(\phi \to \gamma)$ sarebbe banalmente vera.
  - In modo indiretto attraverso l'equivalente enunciato contronominale della forma $((\neg\gamma) \to (\neg\phi))$, cioè assumendo che $(\neg\gamma)$ sia vera e cercando di derivare che anche $(\neg\phi)$ è vera.
  - Per assurdo, cioè assumendo che ϕ e $(\neg\gamma)$ siano entrambe vere e cercando di derivare una contraddizione; si noti che $(\phi \wedge (\neg\gamma)) \equiv (\neg((\neg\phi) \vee \gamma)) \equiv (\neg(\phi \to \gamma))$.
- Un teorema della forma $(\phi \leftrightarrow \gamma)$ viene dimostrato derivando che $(\phi \to \gamma)$ e $(\gamma \to \phi)$ sono vere, dato che $(\phi \leftrightarrow \gamma) \equiv ((\phi \to \gamma) \wedge (\gamma \to \phi))$.
- Esempio: Per dimostrare "se $(p \to q)$ e $(p \to (q \to r))$ allora $(p \to r)$" si può costruire la tabella di verità di $(((p \to q) \wedge (p \to (q \to r))) \to (p \to r))$ e verificare che essa sia una tautologia. Tuttavia, in un testo di matematica troveremmo una dimostrazione come la seguente. Supponiamo che $(p \to q)$ e $(p \to (q \to r))$ siano vere. Concentrandoci su $(p \to r)$, assumiamo che p sia vera. Poiché p è vera e $(p \to q)$ è vera, anche q deve essere vera. Poiché p e q sono vere e $(p \to (q \to r))$ è vera, anche r deve essere vera. Dunque $(p \to r)$ è vera. Di conseguenza $(((p \to q) \wedge (p \to (q \to r))) \to (p \to r))$ è vera.
- Quando si esamina la validità di una formula ben formata, invece di considerare la relazione semantica di conseguenza logica $\models$ propria della teoria dei modelli per stabilire se la formula è una tautologia, si può dunque considerare una relazione sintattica di derivabilità logica $\vdash$ propria della teoria della dimostrazione per stabilire se la formula è un teorema.

- Un sistema deduttivo (o sistema di dimostrazione o sistema formale) è un insieme $\Delta \subseteq FBF^*$ di sequenze non vuote di lunghezza finita di formule ben formate, in cui:
  - Ogni sequenza $\phi \in \Delta$ di lunghezza uno è detta assioma ed è rappresentata come segue:
    $$\frac{}{\phi}$$
  - Ogni sequenza $\phi_1 \dots \phi_k\phi \in \Delta$ con $k \geq 1$ è detta regola di inferenza ed è rappresentata come segue:
    $$\frac{\phi_1 \dots \phi_k}{\phi}$$
    dove $\phi_1, \dots, \phi_k$ sono le premesse mentre ϕ è la conclusione ottenuta per deduzione dalle premesse.
- Sia $\Delta \subseteq FBF^*$ un sistema deduttivo. Diciamo che $\phi \in FBF$ è un teorema in Δ, scritto $\vdash_\Delta \phi$, sse esiste una sequenza non vuota di lunghezza finita di formule ben formate $\gamma_1 \dots \gamma_n \in FBF^*$ tale che:
  - Ogni formula γ_i della sequenza:
    - o è un assioma di Δ, cioè $\gamma_i \in \Delta$;
    - o è la conclusione di una regola di inferenza di Δ applicata a premesse costituite da alcune delle formule che appartengono a $\{\gamma_1, \dots, \gamma_{i-1}\}$.
  - L'ultima formula γ_n della sequenza è ϕ.

  In questo caso la sequenza è detta essere una dimostrazione (o prova) di ϕ in Δ. In generale, dato $\Phi \in 2^{FBF}$, scriviamo $\Phi \vdash_\Delta \phi$ per indicare che le formule della sequenza possono appartenere a Φ; diciamo che Φ è una teoria in Δ sse $\Phi \vdash_\Delta \phi$ implica $\phi \in \Phi$ per ogni $\phi \in FBF$.
- Dato un sistema deduttivo $\Delta \subseteq FBF^*$, diciamo che esso è:
  - Corretto rispetto alla logica considerata sse per ogni $\phi \in FBF$ da $\vdash_\Delta \phi$ segue che $\models \phi$, cioè sono derivabili come teoremi in Δ solo le tautologie della logica considerata.
  - Completo rispetto alla logica considerata sse per ogni $\phi \in FBF$ da $\models \phi$ segue che $\vdash_\Delta \phi$, cioè sono derivabili come teoremi in Δ tutte le tautologie della logica considerata.
  - Decidibile sse è possibile stabilire in tempo finito se un'arbitraria formula ben formata della logica considerata è un teorema in Δ oppure no.
- Il sistema deduttivo sviluppato da Gentzen sotto il nome di deduzione naturale si basa sui seguenti tre gruppi di regole di inferenza il cui significato è molto intuitivo:
  - Regole elementari che introducono connettivi logici rispetto alle premesse:
    $$\frac{p}{(p \vee q)} \qquad \frac{q}{(p \vee q)} \qquad \frac{p \quad q}{(p \wedge q)}$$
  - Regole elementari che eliminano connettivi logici rispetto alle premesse:
    $$\frac{(p \wedge q)}{p} \qquad \frac{(p \wedge q)}{q} \qquad \frac{p \quad (p \to q)}{q} \qquad \frac{(\neg q) \quad (p \to q)}{(\neg p)} \qquad \frac{p \quad (\neg p)}{\mathbb{O}} \qquad \frac{\mathbb{O}}{p}$$
    La terza regola è detta modus ponens, la quarta regola è detta modus tollens e la sesta regola formalizza il principio secondo cui ex falso sequitur quodlibet.
  - Regole condizionali in cui alcune premesse dipendono da ipotesi racchiuse tra parentesi quadre:
    $$\frac{[p]r \quad [q]r \quad (p \vee q)}{r} \qquad \frac{[p]q}{(p \to q)} \qquad \frac{[p]\mathbb{O}}{(\neg p)} \qquad \frac{[(\neg p)]\mathbb{O}}{p}$$
    La prima regola esprime il fatto che da $(p \vee q)$ non si possa inferire né p né q singolarmente ma ogni proposizione r valida tanto sotto p quanto sotto q, mentre la quarta regola è detta reductio ad absurdum.

- Graficamente parlando, le dimostrazioni prendono la forma di alberi. Il motivo è che, ogni volta che la conclusione di una regola coincide con la premessa di un'altra, le due regole possono essere composte. Ciò deriva dal principio, detta regola di taglio, secondo cui se da un insieme di ipotesi Φ si deriva p e se da $\Phi \cup \{p\}$ si deriva q, allora la composizione delle due dimostrazioni è una dimostrazione che da Φ si deriva q. In altri termini, per dimostrare un teorema complesso (q) a partire da una serie di ipotesi (Φ), si può cominciare derivando dei lemmi più semplici (p) da quelle stesse ipotesi.

- Esempio: Il seguente albero rappresenta una dimostrazione di $\{(p \wedge q), ((q \wedge p) \to r)\} \vdash_{\text{DNG}} (r \vee s)$:

$$\dfrac{\dfrac{\dfrac{(p \wedge q)}{q} \quad \dfrac{(p \wedge q)}{p}}{(q \wedge p)} \quad ((q \wedge p) \to r)}{\dfrac{r}{(r \vee s)}}$$

- Teorema: Il sistema di deduzione naturale di Gentzen è corretto e completo rispetto alla logica proposizionale.

- I sistemi deduttivi alla Hilbert accettano come unica regola di inferenza il modus ponens (essendo quella più utile per dimostrare i teoremi della matematica):

$$\dfrac{p \quad (p \to q)}{q}$$

  e sostituiscono le altre regole della deduzione naturale con oppurtuni assiomi che contengono l'implicazione come unico connettivo binario (coerentemente con l'adozione del modus ponens).

- Osservato che la seconda regola condizionale della deduzione naturale è derivabile dai seguenti due assiomi:

$$\dfrac{}{(p \to (q \to p))} \qquad \dfrac{}{((p \to (q \to r)) \to ((p \to q) \to (p \to r)))}$$

  tutte le altre regole di inferenza della deduzione naturale possono essere derivate dai seguenti assiomi in cui le premesse delle regole originarie sono codificate attraverso il connettivo di implicazione:

$$\dfrac{}{(p \to (p \vee q))} \qquad \dfrac{}{(q \to (p \vee q))} \qquad \dfrac{}{(p \to (q \to (p \wedge q)))}$$

$$\dfrac{}{((p \wedge q) \to p)} \qquad \dfrac{}{((p \wedge q) \to q)} \qquad \dfrac{}{(p \to ((\neg p) \to \mathbb{O}))} \qquad \dfrac{}{(\mathbb{O} \to p)}$$

$$\dfrac{}{((p \to r) \to ((q \to r) \to ((p \vee q) \to r)))} \qquad \dfrac{}{((p \to \mathbb{O}) \to (\neg p))} \qquad \dfrac{}{(((\neg p) \to \mathbb{O}) \to p)}$$

- Esempio: Il seguente albero rappresenta una dimostrazione di $\vdash_{\text{H}} (p \to p)$ in cui utilizziamo soltanto i primi due assiomi riportati sopra, dove il secondo assioma viene usato una sola volta con q sostituito da $(p \to p)$ ed r sostituito da p mentre il primo assioma viene usato due volte con analoghe sostituzioni:

$$\dfrac{\dfrac{((p \to ((p \to p) \to p)) \to ((p \to (p \to p)) \to (p \to p))) \quad (p \to ((p \to p) \to p))}{((p \to (p \to p)) \to (p \to p))} \quad (p \to (p \to p))}{(p \to p)}$$

- Teorema: Il sistema deduttivo alla Hilbert formato dal modus ponens e dai seguenti tre assiomi:

$$\frac{}{(p \to (q \to p))} \qquad \frac{}{((p \to (q \to r)) \to ((p \to q) \to (p \to r)))} \qquad \frac{}{(((\neg p) \to (\neg q)) \to (q \to p))}$$

  è corretto e completo rispetto alla logica proposizionale, nonché decidibile.

- Notare che i primi due assiomi corrispondono ai tipi dei combinatori K ed S della logica combinatoria (v. Sez. 3.5).

- Corollario: L'insieme di connettivi logici $\{\neg, \to\}$ è funzionalmente completo per le tautologie.

- La relazione $\vdash_{\mathrm{H}}$ di derivabilità logica nei sistemi deduttivi alla Hilbert è strettamente legata al connettivo $\to$.

- Teorema (di deduzione): Siano $\Phi \in 2^{FBF}$ finito e $\phi, \gamma \in FBF$. Se $\Phi \cup \{\phi\} \vdash_{\mathrm{H}} \gamma$, allora $\Phi \vdash_{\mathrm{H}} (\phi \to \gamma)$.

- Nei sistemi deduttivi alla Hilbert è interessante ridurre il numero di assiomi preservando il potere deduttivo. Il minimo sistema deduttivo alla Hilbert è formato dal modus ponens e dal seguente assioma detto assioma di Meredith:

$$\frac{}{(((((p \to q) \to ((\neg r) \to (\neg s))) \to r) \to t) \to ((t \to p) \to (s \to p)))}$$

- A essere precisi, nei sistema deduttivi alla Hilbert gli assiomi sono in realtà schemi di assiomi, nel senso che da ognuno di essi è possibile ottenere un assioma equivalente a quello originario sostituendo tutte le occorrenze di almeno una proposizione con la stessa formula ben formata (come fatto nell'ultimo esempio relativo a $\vdash_{\mathrm{H}}$). Il motivo per cui il nuovo assioma è equivalente a quello originario discende dal teorema di sostituzione. ■fltplf_12

Capitolo 6

Logica dei Predicati

6.1 Sintassi della Logica dei Predicati

- La logica proposizionale non è sufficientemente espressiva per rappresentare ragionamenti relativi a una moltitudine di oggetti, come "tutti gli oggetti godono di una certa proprietà", "esiste almeno un oggetto che gode di una certa proprietà" o "per ogni oggetto di un certo tipo esiste un oggetto di un altro tipo tale che i due oggetti sono compatibili tra loro". La logica proposizionale si concentra infatti sui connettivi logici e su come il valore di verità di una formula ben formata dipenda dal valore di verità delle sue sottoformule immediate, senza prevedere alcuna strutturazione interna per le proposizioni.
- La logica dei predicati ovvia a questi inconvenienti includendo degli ulteriori operatori detti quantificatori in aggiunta ai connettivi logici della logica proposizionale e sostituendo le proposizioni con predicati applicati a termini. I termini, che individuano gli oggetti di interesse, sono espressi tramite costanti, variabili e funzioni definite sui termini. I predicati, che specificano le proprietà di interesse per gli oggetti precedentemente individuati, sono espressi tramite relazioni su insiemi di termini. Mentre le proposizioni sono variabili logiche, i predicati sono assimilabili a funzioni logiche.
- La logica dei predicati del primo ordine ha un quantificatore universale denotato con $\forall$ ("per ogni") e un quantificatore esistenziale denotato con $\exists$ ("esiste"), i quali possono fare riferimento solo a variabili che compaiono come argomenti di funzioni o predicati. Da ora in poi per logica dei predicati intenderemo sempre quella del primo ordine, escludendo quindi variabili funzionali e variabili predicative assieme alle relative quantificazioni.
- Gli elementi sintattici di base della logica dei predicati sono i seguenti:
  - Un insieme *Cos* di simboli di costante, i quali verranno indicati con le lettere a, b, c.
  - Un insieme numerabile *Var* di simboli di variabile, i quali verranno indicati con le lettere x, y, z.
  - Un insieme *Fun* di simboli di funzione, i quali verranno indicati con le lettere f, g, h.
  - Un insieme *Pred* di simboli di predicato, i quali verranno indicati con le lettere P, Q, R.
  - I connettivi logici $\neg$, $\vee$, $\wedge$, $\rightarrow$, $\leftrightarrow$ della logica proposizionale.
  - I quantificatori $\forall$, $\exists$ su *Var*.
  - I simboli ausiliari (e).
- L'insieme *Ter* dei termini è il più piccolo linguaggio su $Cos \cup Var \cup Fun \cup \{(,)\}$ tale che:
  - $Cos \subseteq Ter$.
  - $Var \subseteq Ter$.
  - $f(t_1, \ldots, t_n) \in Ter$ per ogni $f \in Fun$ con $n \geq 1$ argomenti e $t_1, \ldots, t_n \in Ter$.

  Pertanto la sintassi è $t ::= a \mid x \mid f(t, \ldots, t)$.
  Gli elementi di *Ter* non hanno valori di verità ad essi associati in quanto sono oggetti *extra logici* introdotti nella sintassi soltanto per arricchire l'espressività rispetto alla logica proposizionale.

- L'insieme FBF delle formule ben formate della logica dei predicati è il più piccolo linguaggio su $Ter \cup Pred \cup \{\neg, \vee, \wedge, \rightarrow, \leftrightarrow, \forall, \exists, (,)\}$ tale che:
  - $P(t_1, \ldots, t_n) \in FBF$ per ogni $P \in Pred$ con $n \geq 1$ argomenti e $t_1, \ldots, t_n \in Ter$.
  - $(\neg\phi) \in FBF$ per ogni $\phi \in FBF$.
  - $(\phi_1 \vee \phi_2) \in FBF$ per ogni $\phi_1, \phi_2 \in FBF$.
  - $(\phi_1 \wedge \phi_2) \in FBF$ per ogni $\phi_1, \phi_2 \in FBF$.
  - $(\phi_1 \rightarrow \phi_2) \in FBF$ per ogni $\phi_1, \phi_2 \in FBF$.
  - $(\phi_1 \leftrightarrow \phi_2) \in FBF$ per ogni $\phi_1, \phi_2 \in FBF$.
  - $(\forall x)(\phi) \in FBF$ per ogni $x \in Var$ e $\phi \in FBF$.
  - $(\exists x)(\phi) \in FBF$ per ogni $x \in Var$ e $\phi \in FBF$.

  Pertanto la sintassi è $\phi ::= P(t, \ldots, t) \mid (\neg\phi) \mid (\phi \vee \phi) \mid (\phi \wedge \phi) \mid (\phi \rightarrow \phi) \mid (\phi \leftrightarrow \phi) \mid (\forall x)(\phi) \mid (\exists x)(\phi)$.

- Esempio: "tutte le persone sono mortali" si esprime come $(\forall x)(Persona(x) \rightarrow Mortale(x))$.

- Teorema: Sia FBF' un linguaggio su $Ter \cup Pred \cup \{\neg, \vee, \wedge, \rightarrow, \leftrightarrow, \forall, \exists, (,)\}$ definito ponendo $FBF' = \bigcup_{n \in \mathbb{N}} FBF_n$ dove:

$$FBF_n = \begin{cases} \{P(t_1, \ldots, t_m) \mid P \in Pred \text{ con } m \geq 1 \text{ argomenti e } t_1, \ldots, t_m \in Ter\} & \text{se } n = 0 \\ FBF_{n-1} \cup \{(\neg\phi) \mid \phi \in FBF_{n-1}\} \cup & \text{se } n > 0 \\ \qquad \{(\phi_1 \vee \phi_2) \mid \phi_1, \phi_2 \in FBF_{n-1}\} \cup & \\ \qquad \{(\phi_1 \wedge \phi_2) \mid \phi_1, \phi_2 \in FBF_{n-1}\} \cup & \\ \qquad \{(\phi_1 \rightarrow \phi_2) \mid \phi_1, \phi_2 \in FBF_{n-1}\} \cup & \\ \qquad \{(\phi_1 \leftrightarrow \phi_2) \mid \phi_1, \phi_2 \in FBF_{n-1}\} \cup & \\ \qquad \{(\forall x)(\phi) \mid x \in Var \text{ e } \phi \in FBF_{n-1}\} \cup & \\ \qquad \{(\exists x)(\phi) \mid x \in Var \text{ e } \phi \in FBF_{n-1}\} & \end{cases}$$

  Allora $FBF' = FBF$, cioè FBF ammette una caratterizzazione alternativa basata sulla massima profondità di annidamento dei connettivi e dei quantificatori che compaiono nelle sue formule.

- L'insieme $sf(\phi)$ delle sottoformule di $\phi \in FBF$ è definito per induzione sulla struttura sintattica di ϕ come segue:

$$sf(\phi) = \{\phi\} \cup \begin{cases} \emptyset & \text{se } \phi \text{ è della forma } P(t_1, \ldots, t_n) \text{ con } P \in Pred \text{ e } t_1, \ldots, t_n \in Ter \\ sf(\phi') & \text{se } \phi \text{ è della forma } (\neg\phi'), (\forall x)(\phi') \text{ o } (\exists x)(\phi') \\ sf(\phi'_1) \cup sf(\phi'_2) & \text{se } \phi \text{ è della forma } (\phi'_1 \vee \phi'_2), (\phi'_1 \wedge \phi'_2), (\phi'_1 \rightarrow \phi'_2) \text{ o } (\phi'_1 \leftrightarrow \phi'_2) \end{cases}$$

- Data una formula $\phi \in FBF$, diciamo che essa è:
  - Atomica sse è della forma $P(t_1, \ldots, t_n)$ con $P \in Pred$ e $t_1, \ldots, t_n \in Ter$.
  - Composta con connettivo primario $\neg$ e sottoformula immediata ϕ' sse è della forma $(\neg\phi')$.
  - Composta con connettivo primario $\vee$ e sottoformule immediate ϕ'_1, ϕ'_2 sse è della forma $(\phi'_1 \vee \phi'_2)$.
  - Composta con connettivo primario $\wedge$ e sottoformule immediate ϕ'_1, ϕ'_2 sse è della forma $(\phi'_1 \wedge \phi'_2)$.
  - Composta con connettivo primario $\rightarrow$ e sottoformule immediate ϕ'_1, ϕ'_2 sse è della forma $(\phi'_1 \rightarrow \phi'_2)$.
  - Composta con connettivo primario $\leftrightarrow$ e sottoformule immediate ϕ'_1, ϕ'_2 sse è della forma $(\phi'_1 \leftrightarrow \phi'_2)$.
  - Quantificata universalmente rispetto a x con sottoformula immediata ϕ' sse è della forma $(\forall x)(\phi')$.
  - Quantificata esistenzialmente rispetto a x con sottoformula immediata ϕ' sse è della forma $(\exists x)(\phi')$.

- Teorema (di leggibilità univoca): Le formule ben formate non sono ambigue, cioè esiste un unico modo di leggere ogni $\phi \in FBF$.

- L'uso sistematico delle parentesi può essere evitato introducendo delle regole di precedenza e associatività per i connettivi logici e i quantificatori. È consuetudine assumere che $\forall$, $\exists$ e $\neg$ abbiano precedenza su $\wedge$, che $\wedge$ abbia precedenza su $\vee$, che $\vee$ abbia precedenza su $\rightarrow$, che $\rightarrow$ abbia precedenza su $\leftrightarrow$ e che tutti questi operatori logici siano associativi da sinistra tranne $\forall$, $\exists$ e $\neg$ che lo sono da destra. Inoltre quantificatori consecutivi verranno racchiusi entro un'unica coppia di parentesi tonde, p.e. $(\forall x \exists y)(\phi)$.

- Teorema (principio di induzione strutturale per i termini): Sia $\mathcal{Q}$ una proprietà definita su Ter. Se:

  1. $\mathcal{Q}$ è soddisfatta da ogni costante di Cos;
  2. $\mathcal{Q}$ è soddisfatta da ogni variabile di Var;
  3. per ogni $t_1, \ldots, t_n \in Ter$ con $n \geq 1$, $\mathcal{Q}$ soddisfatta da ogni t_i per $i = 1, \ldots, n$ implica $\mathcal{Q}$ soddisfatta da $f(t_1, \ldots, t_n)$ per ogni $f \in Fun$ con $n \geq 1$ argomenti;

  allora $\mathcal{Q}$ è soddisfatta da ogni $t \in Ter$.

- Teorema (principio di induzione strutturale per le formule ben formate): Sia $\mathcal{Q}$ una proprietà definita su FBF. Se:

  1. $\mathcal{Q}$ è soddisfatta da ogni formula atomica di FBF;
  2. per ogni $\phi \in FBF$, $\mathcal{Q}$ soddisfatta da ϕ implica $\mathcal{Q}$ soddisfatta da $(\neg\phi)$;
  3. per ogni $\phi_1, \phi_2 \in FBF$, $\mathcal{Q}$ soddisfatta da ϕ_1 e $\mathcal{Q}$ soddisfatta da ϕ_2 implicano $\mathcal{Q}$ soddisfatta da $(\phi_1 \vee \phi_2)$;
  4. per ogni $\phi_1, \phi_2 \in FBF$, $\mathcal{Q}$ soddisfatta da ϕ_1 e $\mathcal{Q}$ soddisfatta da ϕ_2 implicano $\mathcal{Q}$ soddisfatta da $(\phi_1 \wedge \phi_2)$;
  5. per ogni $\phi_1, \phi_2 \in FBF$, $\mathcal{Q}$ soddisfatta da ϕ_1 e $\mathcal{Q}$ soddisfatta da ϕ_2 implicano $\mathcal{Q}$ soddisfatta da $(\phi_1 \rightarrow \phi_2)$;
  6. per ogni $\phi_1, \phi_2 \in FBF$, $\mathcal{Q}$ soddisfatta da ϕ_1 e $\mathcal{Q}$ soddisfatta da ϕ_2 implicano $\mathcal{Q}$ soddisfatta da $(\phi_1 \leftrightarrow \phi_2)$;
  7. per ogni $\phi \in FBF$, $\mathcal{Q}$ soddisfatta da ϕ implica $\mathcal{Q}$ soddisfatta da $(\forall x)(\phi)$ per ogni $x \in Var$;
  8. per ogni $\phi \in FBF$, $\mathcal{Q}$ soddisfatta da ϕ implica $\mathcal{Q}$ soddisfatta da $(\exists x)(\phi)$ per ogni $x \in Var$;

  allora $\mathcal{Q}$ è soddisfatta da ogni $\phi \in FBF$.

- Un quantificatore non è solo un operatore logico unario, ma anche un legatore per la sua variabile che ha come campo d'azione la sottoformula immediata a cui è applicato. Siano $\phi \in FBF$ e $x \in Var$:

  – x occorre in ϕ sse x compare in una sottoformula di ϕ priva di quantificatori.
  – Un'occorrenza di x in ϕ è legata (a un quantificatore) se compare in una sottoformula di ϕ della forma $(\forall x)(\phi')$ o $(\exists x)(\phi')$ – nel qual caso ϕ' è il campo d'azione di quel quantificatore – altrimenti è libera.
  – ϕ è chiusa se ogni occorrenza di ogni variabile che compare in ϕ è legata, altrimenti è aperta.

- Esempi:

  – La variabile x non occorre nella formula $(\forall x)(P(a))$ perché la sola sottoformula senza quantificatori è $P(a)$.
  – La formula chiusa $((\forall x)(P(x)) \wedge (\forall x)(Q(x)))$ mostra che due occorrenze di una stessa variabile possono ricadere nel campo d'azione di due quantificatori che compaiono in due sottoformule indipendenti della formula originaria.
  – La formula chiusa $(\forall x)(P(x) \rightarrow (\forall x)(Q(x)))$ mostra che due occorrenze di una stessa variabile possono ricadere nel campo d'azione di due quantificatori che compaiono in due sottoformule tali che una è sottoformula dell'altra. In tal caso, ogni occorrenza della variabile è legata al quantificatore più interno tra quelli nel cui campo d'azione essa si trova; quindi la x di $P(x)$ è legata al quantificatore che sta a sinistra mentre la x di $Q(x)$ è legata al quantificatore che sta a destra.

- Sia $t \in Ter$. L'insieme $var(t)$ delle variabili che occorrono in t è definito per induzione sulla struttura sintattica di t come segue:

$$var(t) = \begin{cases} \emptyset & \text{se } t \in Cos \\ \{t\} & \text{se } t \in Var \\ var(t_1) \cup \cdots \cup var(t_n) & \text{se } t \text{ è della forma } f(t_1, \ldots, t_n) \text{ con } f \in Fun \text{ e } t_1, \ldots, t_n \in Ter \end{cases}$$

- Sia $\phi \in FBF$:
  - L'insieme $var(\phi)$ delle variabili che occorrono in ϕ è definito per induzione sulla struttura sintattica di ϕ come segue:
$$var(\phi) = \begin{cases} var(t_1) \cup \cdots \cup var(t_n) & \text{se } \phi \text{ è della forma } P(t_1, \ldots, t_n) \text{ con } P \in Pred \text{ e } t_1, \ldots, t_n \in Ter \\ var(\phi') & \text{se } \phi \text{ è della forma } (\neg\phi'), (\forall x)(\phi') \text{ o } (\exists x)(\phi') \\ var(\phi'_1) \cup var(\phi'_2) & \text{se } \phi \text{ è della forma } (\phi'_1 \vee \phi'_2), (\phi'_1 \wedge \phi'_2), (\phi'_1 \to \phi'_2) \text{ o } (\phi'_1 \leftrightarrow \phi'_2) \end{cases}$$
  - L'insieme $varleg(\phi)$ delle variabili che occorrono legate in ϕ è definito per induzione sulla struttura sintattica di ϕ come segue:
$$varleg(\phi) = \begin{cases} \emptyset & \text{se } \phi \text{ è della forma } P(t_1, \ldots, t_n) \text{ con } P \in Pred \text{ e } t_1, \ldots, t_n \in Ter \\ varleg(\phi') & \text{se } \phi \text{ è della forma } (\neg\phi') \\ varleg(\phi'_1) \cup varleg(\phi'_2) & \text{se } \phi \text{ è della forma } (\phi'_1 \vee \phi'_2), (\phi'_1 \wedge \phi'_2), (\phi'_1 \to \phi'_2) \text{ o } (\phi'_1 \leftrightarrow \phi'_2) \\ varleg(\phi') & \text{se } \phi \text{ è della forma } (\forall x)(\phi') \text{ o } (\exists x)(\phi') \text{ e } x \notin var(\phi') \\ varleg(\phi') \cup \{x\} & \text{se } \phi \text{ è della forma } (\forall x)(\phi') \text{ o } (\exists x)(\phi') \text{ e } x \in var(\phi') \end{cases}$$
  - L'insieme $varlib(\phi)$ delle variabili che occorrono libere in ϕ è definito per induzione sulla struttura sintattica di ϕ come segue (ϕ è chiusa quando $varlib(\phi) = \emptyset$, aperta quando $varlib(\phi) \neq \emptyset$):
$$varlib(\phi) = \begin{cases} var(t_1) \cup \cdots \cup var(t_n) & \text{se } \phi \text{ è della forma } P(t_1, \ldots, t_n) \text{ con } P \in Pred \text{ e } t_1, \ldots, t_n \in Ter \\ varlib(\phi') & \text{se } \phi \text{ è della forma } (\neg\phi') \\ varlib(\phi'_1) \cup varlib(\phi'_2) & \text{se } \phi \text{ è della forma } (\phi'_1 \vee \phi'_2), (\phi'_1 \wedge \phi'_2), (\phi'_1 \to \phi'_2) \text{ o } (\phi'_1 \leftrightarrow \phi'_2) \\ varlib(\phi') \setminus \{x\} & \text{se } \phi \text{ è della forma } (\forall x)(\phi') \text{ o } (\exists x)(\phi') \end{cases}$$

- Esempio:
  - La formula aperta $((\exists x)(R(a, b, x)) \wedge (\neg Q(x)))$ mostra che la stessa variabile può occorrere sia legata che libera nella medesima formula; infatti la x di $R(a, b, x)$ è legata mentra la x di $Q(x)$ è libera. Ciò significa che in generale $varleg(\phi) \cap varlib(\phi) \neq \emptyset$ quando $\phi \in FBF$ è aperta. In tal caso, per evitare confusione è conveniente sostituire le occorrenze libere oppure le occorrenze legate e relativi quantificatori di quella variabile con un nuovo simbolo di variabile; ad esempio, la formula data può diventare $((\exists x)(R(a, b, x)) \wedge (\neg Q(y)))$ oppure $((\exists z)(R(a, b, z)) \wedge (\neg Q(x)))$.

- Siano $t, t' \in Ter$ e $y \in Var$. Il termine ottenuto da t sostituendo con t' ogni occorrenza di y in t è definito per induzione sulla struttura sintattica di t come segue:
$$t[^{t'}/_{y}] = \begin{cases} t & \text{se } t \in Cos \text{ oppure } t \in Var \text{ e } t \neq y \\ t' & \text{se } t \in Var \text{ e } t = y \\ f(t_1[^{t'}/_{y}], \ldots, t_n[^{t'}/_{y}]) & \text{se } t \text{ è della forma } f(t_1, \ldots, t_n) \text{ con } f \in Fun \text{ e } t_1, \ldots, t_n \in Ter \end{cases}$$

- Siano $\phi \in FBF$, $t' \in Ter$ e $y \in Var$. La formula ottenuta da ϕ sostituendo con t' ogni occorrenza di y libera in ϕ è definita per induzione sulla struttura sintattica di ϕ come segue:
$$\phi[^{t'}/_{y}] = \begin{cases} P(t_1[^{t'}/_{y}], \ldots, t_n[^{t'}/_{y}]) & \text{se } \phi \text{ è della forma } P(t_1, \ldots, t_n) \text{ con } P \in Pred \text{ e } t_1, \ldots, t_n \in Ter \\ (\neg\phi'[^{t'}/_{y}]) & \text{se } \phi \text{ è della forma } (\neg\phi') \\ (\phi'_1[^{t'}/_{y}] \vee \phi'_2[^{t'}/_{y}]) & \text{se } \phi \text{ è della forma } (\phi'_1 \vee \phi'_2) \\ (\phi'_1[^{t'}/_{y}] \wedge \phi'_2[^{t'}/_{y}]) & \text{se } \phi \text{ è della forma } (\phi'_1 \wedge \phi'_2) \\ (\phi'_1[^{t'}/_{y}] \to \phi'_2[^{t'}/_{y}]) & \text{se } \phi \text{ è della forma } (\phi'_1 \to \phi'_2) \\ (\phi'_1[^{t'}/_{y}] \leftrightarrow \phi'_2[^{t'}/_{y}]) & \text{se } \phi \text{ è della forma } (\phi'_1 \leftrightarrow \phi'_2) \\ \phi & \text{se } \phi \text{ è della forma } (\forall y)(\phi') \text{ o } (\exists y)(\phi') \\ (\forall x)(\phi'[^{t'}/_{y}]) & \text{se } \phi \text{ è della forma } (\forall x)(\phi') \text{ e } x \neq y \text{ e } x \notin var(t') \\ (\forall z)(\phi'[^{z}/_{x}][^{t'}/_{y}]) & \text{se } \phi \text{ è della forma } (\forall x)(\phi') \text{ e } x \neq y \text{ e } x \in var(t') \text{ con } z \notin \{y\} \cup var(t') \cup var(\phi') \\ (\exists x)(\phi'[^{t'}/_{y}]) & \text{se } \phi \text{ è della forma } (\exists x)(\phi') \text{ e } x \neq y \text{ e } x \notin var(t') \\ (\exists z)(\phi'[^{z}/_{x}][^{t'}/_{y}]) & \text{se } \phi \text{ è della forma } (\exists x)(\phi') \text{ e } x \neq y \text{ e } x \in var(t') \text{ con } z \notin \{y\} \cup var(t') \cup var(\phi') \end{cases}$$
dove nella nona e nell'undicesima clausola è necessaria una sostituzione preliminare della forma $[^{z}/_{x}]$, con z diversa da tutte le variabili presenti, altrimenti le occorrenze della variabile quantificata x che compaiono in t' verrebbero erroneamente legate. Ad esempio, la formula aperta $(\forall x)(P(x, y))$ verrebbe trasformata nella formula chiusa $(\forall x)(P(x, x))$ se le fosse direttamente applicata la sostituzione $[^{x}/_{y}]$, mentre la nona clausola produce la formula $(\forall z)(P(x, y)[^{z}/_{x}][^{x}/_{y}]) = (\forall z)(P(z, x))$ che è ancora aperta.

- Esempi (i nomi attribuiti a costanti, funzioni e predicati evocano le loro consuete interpretazioni):

  – Un predicato molto importante in aritmetica è l'uguaglianza. Le sue proprietà di riflessività, simmetria, transitività e congruenza rispetto a un operatore f con $n \geq 1$ argomenti possono essere definite attraverso le seguenti formule ben formate della logica dei predicati:
  $(\forall x)(\mathit{Uguale}(x,x))$
  $(\forall x \forall y)(\mathit{Uguale}(x,y) \rightarrow \mathit{Uguale}(y,x))$
  $(\forall x \forall y \forall z)((\mathit{Uguale}(x,y) \wedge \mathit{Uguale}(y,z)) \rightarrow \mathit{Uguale}(x,z))$
  $(\forall x_1 \ldots \forall x_n \forall y_1 \ldots \forall y_n)((\mathit{Uguale}(x_1,y_1) \wedge \cdots \wedge \mathit{Uguale}(x_n,y_n)) \rightarrow \mathit{Uguale}(f(x_1,\ldots,x_n), f(y_1,\ldots,y_n)))$

  – L'aritmetica può essere definita a partire da una costante 0, una funzione unaria *succ*, un generico predicato P per enunciare il principio di induzione, due funzioni binarie *add* e *molt* e il predicato binario *Uguale*, attraverso le seguenti formule ben formate della logica dei predicati che danno luogo all'aritmetica di Peano:
  $(\forall x)(\neg \mathit{Uguale}(\mathit{succ}(x), 0))$
  $(\forall x \forall y)(\mathit{Uguale}(\mathit{succ}(x), \mathit{succ}(y)) \rightarrow \mathit{Uguale}(x,y))$
  $((P(0) \wedge (\forall x)(P(x) \rightarrow P(\mathit{succ}(x)))) \rightarrow (\forall x)(P(x)))$
  $(\forall x)(\mathit{Uguale}(\mathit{add}(x,0),x))$
  $(\forall x \forall y)(\mathit{Uguale}(\mathit{add}(x,\mathit{succ}(y)), \mathit{add}(\mathit{succ}(x),y)))$
  $(\forall x)(\mathit{Uguale}(\mathit{molt}(x,0),0))$
  $(\forall x \forall y)(\mathit{Uguale}(\mathit{molt}(x,\mathit{succ}(y)), \mathit{add}(x,\mathit{molt}(x,y))))$

  – Le strutture algebriche possono essere definite attraverso la logica dei predicati. Per esempio, un gruppo abeliano può essere definito a partire da una costante ν che indica l'elemento neutro, una funzione unaria *inv* che indica l'operazione di inversione, una funzione binaria *comp* che indica l'operazione di composizione e il predicato binario *Uguale*, attraverso le seguenti formule ben formate della logica dei predicati:
  $(\forall x \forall y \forall z)(\mathit{Uguale}(\mathit{comp}(\mathit{comp}(x,y),z), \mathit{comp}(x,\mathit{comp}(y,z))))$
  $(\forall x \forall y)(\mathit{Uguale}(\mathit{comp}(x,y), \mathit{comp}(y,x)))$
  $(\forall x)(\mathit{Uguale}(\mathit{comp}(x,\nu),x) \wedge \mathit{Uguale}(\mathit{comp}(\nu,x),x))$
  $(\forall x)(\mathit{Uguale}(\mathit{comp}(x,\mathit{inv}(x)),\nu) \wedge \mathit{Uguale}(\mathit{comp}(\mathit{inv}(x),x),\nu))$

  – Le strutture dati possono essere definite attraverso la logica dei predicati. Per esempio, una pila può essere definita a partire da una costante ε che indica la pila vuota, una funzione unaria *cima* che restituisce l'elemento che sta in cima alla pila, una funzione unaria *togli* che restituisce la pila dopo aver tolto l'elemento che sta in cima, una funzione binaria *metti* che restituisce la pila dopo aver messo un nuovo elemento in cima e il predicato binario *Uguale*, attraverso le seguenti formule ben formate della logica dei predicati dove x rappresenta una pila e y un elemento:
  $(\forall x \forall y)(\mathit{Uguale}(\mathit{cima}(\mathit{metti}(x,y)),y))$
  $(\forall x \forall y)(\mathit{Uguale}(\mathit{togli}(\mathit{metti}(x,y)),x))$
  $(\forall x)(\mathit{Uguale}(\mathit{metti}(\mathit{togli}(x), \mathit{cima}(x)),x))$
  $(\forall x)(\mathit{Uguale}(x,\varepsilon) \rightarrow \mathit{Uguale}(\mathit{togli}(x),x))$ ■fltplf_13

6.2 Semantica e Indecidibilità della Logica dei Predicati

- Per stabilire il valore di verità di una formula ben formata della logica dei predicati, è necessario fissare un dominio, interpretare in quel dominio i simboli di costante, funzione e predicato che compaiono nella formula e assegnare un valore del dominio a ciascuna variabile che occorre libera nella formula (quindi il valore di verità della formula non è assoluto ma dipende dal dominio e dall'interpretazione). Il significato dei connettivi logici è lo stesso della logica proposizionale, mentre il significato dei quantificatori può essere informalmente descritto tramite sostituzioni sintattiche come segue:

  – $(\forall x)(\phi)$ è vera sse per ogni $t \in \mathit{Ter}$ con $\mathit{var}(t) = \emptyset$ risulta che $\phi[^t/_x]$ è vera
  – $(\exists x)(\phi)$ è vera sse esiste $t \in \mathit{Ter}$ con $\mathit{var}(t) = \emptyset$ tale che $\phi[^t/_x]$ è vera.

- Chiamiamo ambiente ogni tripla $\mathcal{E} = (D, \mathcal{I}, \mathcal{V})$ dove:
  - D è un insieme non vuoto di valori detto dominio (o universo).
  - $\mathcal{I}$ è una funzione di interpretazione che associa a ogni simbolo di:
    - costante $a \in \mathit{Cos}$ un valore $\mathcal{I}(a) \in D$;
    - funzione $f \in \mathit{Fun}$ con $n \geq 1$ argomenti una funzione $\mathcal{I}(f) : D^n \to D$;
    - predicato $P \in \mathit{Pred}$ con $n \geq 1$ argomenti una relazione $\mathcal{I}(P) \subseteq D^n$ (l'idea è che le n-uple di valori di D in $\mathcal{I}(P)$ sono tutte e sole quelle per cui il predicato è vero).
  - $\mathcal{V}$ è una funzione che assegna a ogni occorrenza libera di una variabile $x \in \mathit{Var}$ un valore $\mathcal{V}(x) \in D$.
- Dato un ambiente $\mathcal{E} = (D, \mathcal{I}, \mathcal{V})$, definiamo l'interpretazione di un termine non costante $t \in \mathit{Ter} \setminus \mathit{Cos}$ estendendo la funzione di interpretazione $\mathcal{I}$ per induzione sulla struttura sintattica di t come segue:
$$\mathcal{I}(t) = \begin{cases} \mathcal{V}(t) & \text{se } t \in \mathit{Var} \\ \mathcal{I}(f)(\mathcal{I}(t_1), \ldots, \mathcal{I}(t_n)) & \text{se } t \text{ è della forma } f(t_1, \ldots, t_n) \text{ con } f \in \mathit{Fun} \text{ e } t_1, \ldots, t_n \in \mathit{Ter} \end{cases}$$
- Sia Amb l'insieme di tutti gli ambienti. La semantica della logica dei predicati viene formalizzata attraverso la relazione di soddisfacimento (o relazione di verità) $\models$ definita come il più piccolo sottoinsieme di $\mathit{Amb} \times \mathit{FBF}$ tale che:
  - $(D, \mathcal{I}, \mathcal{V}) \models P(t_1, \ldots, t_n)$ se $(\mathcal{I}(t_1), \ldots, \mathcal{I}(t_n)) \in \mathcal{I}(P)$.
  - $(D, \mathcal{I}, \mathcal{V}) \models (\neg\phi)$ se $(D, \mathcal{I}, \mathcal{V}) \not\models \phi$.
  - $(D, \mathcal{I}, \mathcal{V}) \models (\phi_1 \vee \phi_2)$ se $(D, \mathcal{I}, \mathcal{V}) \models \phi_1$ o $(D, \mathcal{I}, \mathcal{V}) \models \phi_2$.
  - $(D, \mathcal{I}, \mathcal{V}) \models (\phi_1 \wedge \phi_2)$ se $(D, \mathcal{I}, \mathcal{V}) \models \phi_1$ e $(D, \mathcal{I}, \mathcal{V}) \models \phi_2$.
  - $(D, \mathcal{I}, \mathcal{V}) \models (\phi_1 \to \phi_2)$ se $(D, \mathcal{I}, \mathcal{V}) \not\models \phi_1$ oppure $(D, \mathcal{I}, \mathcal{V}) \models \phi_2$.
  - $(D, \mathcal{I}, \mathcal{V}) \models (\phi_1 \leftrightarrow \phi_2)$ se $(D, \mathcal{I}, \mathcal{V}) \models \phi_1$ e $(D, \mathcal{I}, \mathcal{V}) \models \phi_2$, oppure $(D, \mathcal{I}, \mathcal{V}) \not\models \phi_1$ e $(D, \mathcal{I}, \mathcal{V}) \not\models \phi_2$.
  - $(D, \mathcal{I}, \mathcal{V}) \models (\forall x)(\phi)$ se per ogni $d \in D$ risulta che $(D, \mathcal{I}, \mathcal{V} \setminus \{(x, \mathcal{V}(x))\} \cup \{(x, d)\}) \models \phi$.
  - $(D, \mathcal{I}, \mathcal{V}) \models (\exists x)(\phi)$ se esiste $d \in D$ tale che $(D, \mathcal{I}, \mathcal{V} \setminus \{(x, \mathcal{V}(x))\} \cup \{(x, d)\}) \models \phi$.

  Notiamo che nelle ultime due clausole $(D, \mathcal{I}, \mathcal{V} \setminus \{(x, \mathcal{V}(x))\} \cup \{(x, d)\}) \models \phi$ equivale a $(D, \mathcal{I}, \mathcal{V}) \models \phi[^d/_x]$. Dato $\mathcal{E} = (D, \mathcal{I}, \mathcal{V})$, quando $\mathcal{E} \models \phi$ diciamo che $\mathcal{E}$ soddisfa ϕ oppure che $\mathcal{E}$ è un modello di ϕ.
- Esempio: Consideriamo la formula $((\forall x)(P(x, f(x))) \wedge Q(g(a, z)))$. Se prendiamo un ambiente in cui il dominio è $\mathbb{N}$, la costante a è interpretata come il numero 2, la funzione f è interpretata come la funzione successore, la funzione g è interpretata come l'operazione $+$, il predicato P è interpretato come la relazione $<$, il predicato Q è interpretato come il fatto di essere un numero primo e all'occorrenza libera della variabile z viene assegnato valore 1, allora l'ambiente soddisfa la formula. Se a z avessimo assegnato 2, l'ambiente non avrebbe soddisfatto la formula.
- Teorema: Siano $\phi \in \mathit{FBF}$ ed $\mathcal{E}_1 = (D, \mathcal{I}, \mathcal{V}_1), \mathcal{E}_2 = (D, \mathcal{I}, \mathcal{V}_2) \in \mathit{Amb}$. Se per ogni $x \in \mathit{varlib}(\phi)$ risulta che $\mathcal{V}_1(x) = \mathcal{V}_2(x)$, allora $\mathcal{E}_1 \models \phi$ sse $\mathcal{E}_2 \models \phi$.
- Corollario: Siano $\phi \in \mathit{FBF}$ ed $\mathcal{E}_1 = (D, \mathcal{I}, \mathcal{V}_1), \mathcal{E}_2 = (D, \mathcal{I}, \mathcal{V}_2) \in \mathit{Amb}$. Se ϕ è chiusa allora $\mathcal{E}_1 \models \phi$ sse $\mathcal{E}_2 \models \phi$.
- Data una formula $\phi \in \mathit{FBF}$, diciamo che essa è:
  - Soddisfacibile sse esiste un ambiente $\mathcal{E}$ tale che $\mathcal{E} \models \phi$.
  - Valida, ovvero una tautologia, sse $\mathcal{E} \models \phi$ per ogni ambiente $\mathcal{E}$.
  - Insoddisfacibile, ovvero una contraddizione, sse $\mathcal{E} \not\models \phi$ per ogni ambiente $\mathcal{E}$.
- Teorema: Sia $\phi \in \mathit{FBF}$. Allora:
  - ϕ è una tautologia sse $(\neg\phi)$ è una contraddizione.
  - ϕ non è una tautologia sse $(\neg\phi)$ è soddisfacibile.
  - ϕ è soddisfacibile sse ϕ non è una contraddizione.

- Siano $\phi \in FBF$ e $varlib(\phi) = \{x_1, \ldots, x_k\}$ con $k \in \mathbb{N}$:
  - La chiusura universale di ϕ è definita come $cu(\phi) = (\forall x_1 \ldots \forall x_k)(\phi)$.
  - La chiusura esistenziale di ϕ è definita come $ce(\phi) = (\exists x_1 \ldots \exists x_k)(\phi)$.
- Teorema: Sia $\phi \in FBF$. Allora:
  - ϕ è una tautologia sse $cu(\phi)$ è una tautologia.
  - ϕ è soddisfacibile sse $ce(\phi)$ è soddisfacibile.
- Esempi:
  - La formula $((\exists x \forall y)(P(x,y)) \rightarrow (\forall y \exists x)(P(x,y)))$ è una tautologia sse per ogni ambiente $(D, \mathcal{I}, \mathcal{V})$ risulta che $(D, \mathcal{I}, \mathcal{V}) \models (\exists x \forall y)(P(x,y))$ implica $(D, \mathcal{I}, \mathcal{V}) \models (\forall y \exists x)(P(x,y))$. Prendiamo dunque un ambiente $(D, \mathcal{I}, \mathcal{V})$ tale che $(D, \mathcal{I}, \mathcal{V}) \models (\exists x \forall y)(P(x,y))$, cioè tale che esiste $d \in D$ per cui $(D, \mathcal{I}, \mathcal{V} \setminus \{(x, \mathcal{V}(x)), (y, \mathcal{V}(y))\} \cup \{(x,d), (y,d')\}) \models P(x,y)$ per ogni $d' \in D$. Allora per ogni $d' \in D$ esiste $d'' \in D$ tale che $(D, \mathcal{I}, \mathcal{V} \setminus \{(y, \mathcal{V}(y)), (x, \mathcal{V}(x))\} \cup \{(y,d'), (x,d'')\}) \models P(x,y)$ come si vede ponendo $d'' = d$ e quindi $(D, \mathcal{I}, \mathcal{V}) \models (\forall y \exists x)(P(x,y))$. Dunque $(D, \mathcal{I}, \mathcal{V})$ è un modello per l'intera formula e perciò quest'ultima è una tautologia in virtù della generalità del modello.
  - Verificare l'insoddisfacibilità di una formula è spesso più facile che verificarne la validità. Supponiamo che in un piccolo paese ci sia un barbiere che rade tutti e soli gli uomini che non si radono da soli. Indicato il barbiere con la costante b, il fatto che egli rade solo coloro che non si radono da soli è formalizzato come $(\forall x)(Rade(b,x) \rightarrow (\neg Rade(x,x)))$, mentre il fatto che egli rade tutti coloro che non si radono da soli è formalizzato dalla formula $(\forall x)((\neg Rade(x,x)) \rightarrow Rade(b,x))$. Dunque lo scenario è descritto dalla formula $(\forall x)(Rade(b,x) \leftrightarrow (\neg Rade(x,x)))$. La domanda è: chi rade il barbiere? Se il barbiere si rade da solo, allora il barbiere non può radersi perché egli rade solo coloro che non si radono da soli. Se il barbiere non si rade da solo, allora il barbiere deve radersi perché egli rade tutti coloro che non si radono da soli. Abbiamo dunque un paradosso. In effetti la formula $(\forall x)(Rade(b,x) \leftrightarrow (\neg Rade(x,x)))$ è una contraddizione perchè, preso un qualsiasi ambiente $(D, \mathcal{I}, \mathcal{V})$, questo non può mai essere un modello della formula in quanto per $x = b$ abbiamo $(D, \mathcal{I}, \mathcal{V} \setminus \{(x, \mathcal{V}(x))\} \cup \{(x, \mathcal{I}(b))\}) \not\models (Rade(b,x) \leftrightarrow (\neg Rade(x,x)))$.
- Il problema di stabilire se una formula ben formata della logica dei predicati sia soddisfacibile, una tautologia o una contraddizione è molto più complicato rispetto al caso della logica proposizionale. Questo è dovuto non solo al fatto che esistono infinite interpretazioni di una formula predicativa, ma anche al fatto che i domini delle interpretazioni possono essere insiemi infiniti e quindi stabilire la verità di formule quantificate può richiedere un numero illimitato di controlli. Nel 1936 Alonzo Church dimostrò che nella logica dei predicati il problema della validità non è decidibile. Più precisamente, il problema risulta essere semi-decidibile, cioè se una formula arbitraria è una tautologia allora è possibile stabilirlo in tempo finito tramite un algoritmo, ma se *non* lo è l'algoritmo potrebbe *non* terminare. Da ciò segue che pure il problema della insoddisfacibilità è semi-decidibile – perché ϕ è una contraddizione sse $(\neg\phi)$ è una tautologia – mentre il problema della soddisfacibilità non è nemmeno semi-decidibile – perché ϕ è soddisfacibile sse $(\neg\phi)$ non è una tautologia, ovvero sse ϕ non è una contraddizione.

6.3 Conseguenza ed Equivalenza nella Logica dei Predicati

- Le nozioni di conseguenza ed equivalenza per la logica dei predicati sono definite come le corrispondenti nozioni per la logica proposizionale e quindi godono delle stesse proprietà.
- Dato un insieme di formule $\Phi \in 2^{FBF}$, denotiamo con $modelli(\Phi)$ l'insieme degli ambienti che soddisfano tutte le formule di Φ.

- La relazione di soddisfacimento $\models$ può essere trasformata in una relazione su 2^{FBF} detta relazione di conseguenza logica ponendo $\Phi \models \Gamma$ (o $\Phi \Rightarrow \Gamma$) sse $\mathit{modelli}(\Phi) \subseteq \mathit{modelli}(\Gamma)$ per ogni $\Phi, \Gamma \in 2^{FBF}$. Indichiamo con $\models \Gamma$ il fatto che tutte le formule di Γ siano delle tautologie.

- Teorema (di conseguenza logica): Dato $n \in \mathbb{N}$, siano $\Phi = \{\phi_1, \ldots, \phi_n\} \in 2^{FBF}$ e $\gamma \in \mathit{FBF}$. Allora $\Phi \models \{\gamma\}$ sse $((\bigwedge_{i=1}^{n} \phi_i) \rightarrow \gamma)$ è una tautologia, ovvero sse $((\bigwedge_{i=1}^{n} \phi_i) \wedge (\neg\gamma))$ è una contraddizione.

- Corollario: Siano $\phi, \gamma \in \mathit{FBF}$. Allora $\{\phi\} \models \{\gamma\}$ sse $\models \{(\phi \rightarrow \gamma)\}$.

- La relazione di equivalenza logica è una relazione su 2^{FBF} definita ponendo $\Phi \equiv \Gamma$ (o $\Phi \Leftrightarrow \Gamma$) sse $\Phi \models \Gamma$ e $\Gamma \models \Phi$, cioè sse $\mathit{modelli}(\Phi) = \mathit{modelli}(\Gamma)$.

- Teorema (di equivalenza logica): Dato $n \in \mathbb{N}$, siano $\Phi = \{\phi_1, \ldots, \phi_n\} \in 2^{FBF}$ e $\gamma \in \mathit{FBF}$. Allora $\Phi \equiv \{\gamma\}$ sse $((\bigwedge_{i=1}^{n} \phi_i) \leftrightarrow \gamma)$ è una tautologia.

- Corollario: Siano $\phi, \gamma \in \mathit{FBF}$. Allora $\{\phi\} \equiv \{\gamma\}$ sse $\models \{(\phi \leftrightarrow \gamma)\}$.

- La relazione di equivalenza logica ristretta a FBF, cioè a insiemi singoletto (che verranno da ora in poi denotati senza parentesi graffe), risulta essere una congruenza rispetto a tutti i connettivi logici e ai quantificatori. Per i quantificatori vale anche una proprietà di ridenominazione delle variabili legate ad essi che è analoga all'α-conversione del λ-calcolo (v. Sez. 3.2).

- Teorema (di sostituzione): Siano $\phi_1, \phi_2 \in \mathit{FBF}$. Se $\phi_1 \equiv \phi_2$ allora:

  - $(\neg\phi_1) \equiv (\neg\phi_2)$.
  - $(\phi_1 \vee \gamma) \equiv (\phi_2 \vee \gamma)$ e $(\gamma \vee \phi_1) \equiv (\gamma \vee \phi_2)$ per ogni $\gamma \in \mathit{FBF}$.
  - $(\phi_1 \wedge \gamma) \equiv (\phi_2 \wedge \gamma)$ e $(\gamma \wedge \phi_1) \equiv (\gamma \wedge \phi_2)$ per ogni $\gamma \in \mathit{FBF}$.
  - $(\phi_1 \rightarrow \gamma) \equiv (\phi_2 \rightarrow \gamma)$ e $(\gamma \rightarrow \phi_1) \equiv (\gamma \rightarrow \phi_2)$ per ogni $\gamma \in \mathit{FBF}$.
  - $(\phi_1 \leftrightarrow \gamma) \equiv (\phi_2 \leftrightarrow \gamma)$ e $(\gamma \leftrightarrow \phi_1) \equiv (\gamma \leftrightarrow \phi_2)$ per ogni $\gamma \in \mathit{FBF}$.
  - $(\forall x)(\phi_1) \equiv (\forall x)(\phi_2)$ per ogni $x \in \mathit{Var}$.
  - $(\exists x)(\phi_1) \equiv (\exists x)(\phi_2)$ per ogni $x \in \mathit{Var}$.

- Teorema (di α-conversione): Siano $\phi \in \mathit{FBF}$ e $y \in \mathit{Var}$. Se $y \notin \mathit{varlib}(\phi)$ allora per ogni $x \in \mathit{Var}$:

  - $(\forall x)(\phi) \equiv (\forall y)(\phi[^{y}/_{x}])$.
  - $(\exists x)(\phi) \equiv (\exists y)(\phi[^{y}/_{x}])$.

- Dimostrazione: Il risultato segue dal fatto che per ogni ambiente $(D, \mathcal{I}, \mathcal{V})$ e per ogni $d \in D$ risulta $(D, \mathcal{I}, \mathcal{V} \setminus \{(x, \mathcal{V}(x))\} \cup \{(x, d)\}) \models \phi$ sse $(D, \mathcal{I}, \mathcal{V} \setminus \{(y, \mathcal{V}(y))\} \cup \{(y, d)\}) \models \phi[^{y}/_{x}]$ qualora $y \notin \mathit{varlib}(\phi)$. Dimostriamo questo fatto procedendo per induzione sulla struttura sintattica di ϕ tale che $y \notin \mathit{varlib}(\phi)$, dopo aver osservato che se ϕ non è una formula quantificata allora $y \notin \mathit{varlib}(\phi)$ implica $y \notin \mathit{varlib}(\phi')$ per ogni sottoformula immediata ϕ' di ϕ:

  - Se ϕ è della forma $P(t_1, \ldots, t_n)$ con $P \in \mathit{Pred}$ e $t_1, \ldots, t_n \in \mathit{Ter}$ allora per ogni ambiente $(D, \mathcal{I}, \mathcal{V})$ e per ogni $d \in D$ risulta $(D, \mathcal{I}, \mathcal{V} \setminus \{(x, \mathcal{V}(x))\} \cup \{(x, d)\}) \models \phi$ sse $(D, \mathcal{I}, \mathcal{V} \setminus \{(y, \mathcal{V}(y))\} \cup \{(y, d)\}) \models \phi[^{y}/_{x}]$ perché $y \notin \mathit{varlib}(\phi)$ e quindi $y \notin \mathit{var}(\phi)$ essendo $\mathit{var}(\phi) = \mathit{varlib}(\phi)$.
  - Sia ϕ della forma $(\neg\phi')$ e supponiamo che per ogni ambiente $(D, \mathcal{I}, \mathcal{V})$ e per ogni $d \in D$ risulti $(D, \mathcal{I}, \mathcal{V} \setminus \{(x, \mathcal{V}(x))\} \cup \{(x, d)\}) \models \phi'$ sse $(D, \mathcal{I}, \mathcal{V} \setminus \{(y, \mathcal{V}(y))\} \cup \{(y, d)\}) \models \phi'[^{y}/_{x}]$. Allora per ogni ambiente $(D, \mathcal{I}, \mathcal{V})$ e per ogni $d \in D$ abbiamo che $(D, \mathcal{I}, \mathcal{V} \setminus \{(x, \mathcal{V}(x))\} \cup \{(x, d)\}) \models \phi$ sse $(D, \mathcal{I}, \mathcal{V} \setminus \{(x, \mathcal{V}(x))\} \cup \{(x, d)\}) \not\models \phi'$, cioè sse $(D, \mathcal{I}, \mathcal{V} \setminus \{(y, \mathcal{V}(y))\} \cup \{(y, d)\}) \not\models \phi'[^{y}/_{x}]$ sfruttando l'ipotesi induttiva, cioè sse $(D, \mathcal{I}, \mathcal{V} \setminus \{(y, \mathcal{V}(y))\} \cup \{(y, d)\}) \models \phi[^{y}/_{x}]$.

- Sia ϕ della forma $(\phi'_1 \vee \phi'_2)$ e supponiamo che per ogni ambiente $(D, \mathcal{I}, \mathcal{V})$ e per ogni $d \in D$ risulti $(D, \mathcal{I}, \mathcal{V}\setminus\{(x, \mathcal{V}(x))\} \cup \{(x, d)\}) \models \phi'_i$ sse $(D, \mathcal{I}, \mathcal{V}\setminus\{(y, \mathcal{V}(y))\} \cup \{(y, d)\}) \models \phi'_i[^y/_x]$ per $i \in \{1, 2\}$. Allora per ogni ambiente $(D, \mathcal{I}, \mathcal{V})$ e per ogni $d \in D$ abbiamo che $(D, \mathcal{I}, \mathcal{V}\setminus\{(x, \mathcal{V}(x))\}\cup\{(x, d)\}) \models \phi$ sse $(D, \mathcal{I}, \mathcal{V}\setminus\{(x, \mathcal{V}(x))\} \cup \{(x, d)\}) \models \phi'_1$ o $(D, \mathcal{I}, \mathcal{V}\setminus\{(x, \mathcal{V}(x))\} \cup \{(x, d)\}) \models \phi'_2$, cioè sse $(D, \mathcal{I}, \mathcal{V}\setminus\{(y, \mathcal{V}(y))\} \cup \{(y, d)\}) \models \phi'_1[^y/_x]$ o $(D, \mathcal{I}, \mathcal{V}\setminus\{(y, \mathcal{V}(y))\} \cup \{(y, d)\}) \models \phi'_2[^y/_x]$ sfruttando l'ipotesi induttiva, cioè sse $(D, \mathcal{I}, \mathcal{V}\setminus\{(y, \mathcal{V}(y))\} \cup \{(y, d)\}) \models \phi[^y/_x]$.
- Sia ϕ della forma $(\phi'_1 \wedge \phi'_2)$ e supponiamo che per ogni ambiente $(D, \mathcal{I}, \mathcal{V})$ e per ogni $d \in D$ risulti $(D, \mathcal{I}, \mathcal{V}\setminus\{(x, \mathcal{V}(x))\} \cup \{(x, d)\}) \models \phi'_i$ sse $(D, \mathcal{I}, \mathcal{V}\setminus\{(y, \mathcal{V}(y))\} \cup \{(y, d)\}) \models \phi'_i[^y/_x]$ per $i \in \{1, 2\}$. Allora per ogni ambiente $(D, \mathcal{I}, \mathcal{V})$ e per ogni $d \in D$ abbiamo che $(D, \mathcal{I}, \mathcal{V}\setminus\{(x, \mathcal{V}(x))\}\cup\{(x, d)\}) \models \phi$ sse $(D, \mathcal{I}, \mathcal{V}\setminus\{(x, \mathcal{V}(x))\} \cup \{(x, d)\}) \models \phi'_1$ e $(D, \mathcal{I}, \mathcal{V}\setminus\{(x, \mathcal{V}(x))\} \cup \{(x, d)\}) \models \phi'_2$, cioè sse $(D, \mathcal{I}, \mathcal{V}\setminus\{(y, \mathcal{V}(y))\} \cup \{(y, d)\}) \models \phi'_1[^y/_x]$ e $(D, \mathcal{I}, \mathcal{V}\setminus\{(y, \mathcal{V}(y))\} \cup \{(y, d)\}) \models \phi'_2[^y/_x]$ sfruttando l'ipotesi induttiva, cioè sse $(D, \mathcal{I}, \mathcal{V}\setminus\{(y, \mathcal{V}(y))\} \cup \{(y, d)\}) \models \phi[^y/_x]$.
- Sia ϕ della forma $(\phi'_1 \to \phi'_2)$ e supponiamo che per ogni ambiente $(D, \mathcal{I}, \mathcal{V})$ e per ogni $d \in D$ risulti $(D, \mathcal{I}, \mathcal{V}\setminus\{(x, \mathcal{V}(x))\} \cup \{(x, d)\}) \models \phi'_i$ sse $(D, \mathcal{I}, \mathcal{V}\setminus\{(y, \mathcal{V}(y))\} \cup \{(y, d)\}) \models \phi'_i[^y/_x]$ per $i \in \{1, 2\}$. Allora per ogni ambiente $(D, \mathcal{I}, \mathcal{V})$ e per ogni $d \in D$ abbiamo che $(D, \mathcal{I}, \mathcal{V}\setminus\{(x, \mathcal{V}(x))\}\cup\{(x, d)\}) \models \phi$ sse $(D, \mathcal{I}, \mathcal{V}\setminus\{(x, \mathcal{V}(x))\} \cup \{(x, d)\}) \not\models \phi'_1$ oppure $(D, \mathcal{I}, \mathcal{V}\setminus\{(x, \mathcal{V}(x))\} \cup \{(x, d)\}) \models \phi'_2$, cioè sse $(D, \mathcal{I}, \mathcal{V}\setminus\{(y, \mathcal{V}(y))\} \cup \{(y, d)\}) \not\models \phi'_1[^y/_x]$ oppure $(D, \mathcal{I}, \mathcal{V}\setminus\{(y, \mathcal{V}(y))\} \cup \{(y, d)\}) \models \phi'_2[^y/_x]$ sfruttando l'ipotesi induttiva, cioè sse $(D, \mathcal{I}, \mathcal{V}\setminus\{(y, \mathcal{V}(y))\} \cup \{(y, d)\}) \models \phi[^y/_x]$.
- Sia ϕ della forma $(\phi'_1 \leftrightarrow \phi'_2)$ e supponiamo che per ogni ambiente $(D, \mathcal{I}, \mathcal{V})$ e per ogni $d \in D$ risulti $(D, \mathcal{I}, \mathcal{V}\setminus\{(x, \mathcal{V}(x))\} \cup \{(x, d)\}) \models \phi'_i$ sse $(D, \mathcal{I}, \mathcal{V}\setminus\{(y, \mathcal{V}(y))\} \cup \{(y, d)\}) \models \phi'_i[^y/_x]$ per $i \in \{1, 2\}$. Allora per ogni ambiente $(D, \mathcal{I}, \mathcal{V})$ e per ogni $d \in D$ abbiamo che $(D, \mathcal{I}, \mathcal{V}\setminus\{(x, \mathcal{V}(x))\}\cup\{(x, d)\}) \models \phi$ sse $(D, \mathcal{I}, \mathcal{V}\setminus\{(x, \mathcal{V}(x))\} \cup \{(x, d)\}) \models \phi'_1$ e $(D, \mathcal{I}, \mathcal{V}\setminus\{(x, \mathcal{V}(x))\} \cup \{(x, d)\}) \models \phi'_2$ oppure $(D, \mathcal{I}, \mathcal{V}\setminus\{(x, \mathcal{V}(x))\} \cup \{(x, d)\}) \not\models \phi'_1$ e $(D, \mathcal{I}, \mathcal{V}\setminus\{(x, \mathcal{V}(x))\} \cup \{(x, d)\}) \not\models \phi'_2$, cioè sse $(D, \mathcal{I}, \mathcal{V}\setminus\{(y, \mathcal{V}(y))\} \cup \{(y, d)\}) \models \phi'_1[^y/_x]$ e $(D, \mathcal{I}, \mathcal{V}\setminus\{(y, \mathcal{V}(y))\} \cup \{(y, d)\}) \models \phi'_2[^y/_x]$ oppure $(D, \mathcal{I}, \mathcal{V}\setminus\{(y, \mathcal{V}(y))\} \cup \{(y, d)\}) \not\models \phi'_1[^y/_x]$ e $(D, \mathcal{I}, \mathcal{V}\setminus\{(y, \mathcal{V}(y))\} \cup \{(y, d)\}) \not\models \phi'_2[^y/_x]$ sfruttando l'ipotesi induttiva, cioè sse $(D, \mathcal{I}, \mathcal{V}\setminus\{(y, \mathcal{V}(y))\} \cup \{(y, d)\}) \models \phi[^y/_x]$.
- Sia ϕ della forma $(\forall z)(\phi')$ oppure $(\exists z)(\phi')$. Ci sono tre casi:
  - Se $x = z$ allora $\phi[^y/_x] = \phi$ e quindi per ogni ambiente $(D, \mathcal{I}, \mathcal{V})$ e per ogni $d \in D$ risulta $(D, \mathcal{I}, \mathcal{V}\setminus\{(x, \mathcal{V}(x))\} \cup \{(x, d)\}) \models \phi$ sse $(D, \mathcal{I}, \mathcal{V}\setminus\{(y, \mathcal{V}(y))\} \cup \{(y, d)\}) \models \phi[^y/_x]$ perché il terzo elemento dell'ambiente riguarda le occorrenze libere di variabili mentre $x, y \notin \mathit{varlib}(\phi)$.
  - Siano $x \neq z$ e $z \neq y$, cosicché $y \notin \mathit{varlib}(\phi')$, e supponiamo che per ogni ambiente $(D, \mathcal{I}, \mathcal{V})$ e per ogni $d \in D$ risulti $(D, \mathcal{I}, \mathcal{V}\setminus\{(x, \mathcal{V}(x))\} \cup \{(x, d)\}) \models \phi'$ sse $(D, \mathcal{I}, \mathcal{V}\setminus\{(y, \mathcal{V}(y))\} \cup \{(y, d)\}) \models \phi'[^y/_x]$. Ci sono due sottocasi:
    - Se ϕ è della forma $(\forall z)(\phi')$ allora per ogni ambiente $(D, \mathcal{I}, \mathcal{V})$ e per ogni $d \in D$ risulta $(D, \mathcal{I}, \mathcal{V}\setminus\{(x, \mathcal{V}(x))\} \cup \{(x, d)\}) \models \phi$ sse $(D, \mathcal{I}, \mathcal{V}_{d'}\setminus\{(x, \mathcal{V}_{d'}(x))\} \cup \{(x, d)\}) \models \phi'$ per ogni $d' \in D$ con $\mathcal{V}_{d'} = \mathcal{V}\setminus\{(z, \mathcal{V}(z))\} \cup \{(z, d')\}$, cioè sse $(D, \mathcal{I}, \mathcal{V}_{d'}\setminus\{(y, \mathcal{V}_{d'}(y))\} \cup \{(y, d)\}) \models \phi'[^y/_x]$ per ogni $d' \in D$ sfruttando l'ipotesi induttiva, cioè sse $(D, \mathcal{I}, \mathcal{V}\setminus\{(y, \mathcal{V}(y))\} \cup \{(y, d)\}) \models \phi[^y/_x]$.
    - Se ϕ è della forma $(\exists z)(\phi')$ allora per ogni ambiente $(D, \mathcal{I}, \mathcal{V})$ e per ogni $d \in D$ risulta $(D, \mathcal{I}, \mathcal{V}\setminus\{(x, \mathcal{V}(x))\} \cup \{(x, d)\}) \models \phi$ sse esiste $d' \in D$ tale che $(D, \mathcal{I}, \mathcal{V}_{d'}\setminus\{(x, \mathcal{V}_{d'}(x))\} \cup \{(x, d)\}) \models \phi'$ con $\mathcal{V}_{d'} = \mathcal{V}\setminus\{(z, \mathcal{V}(z))\} \cup \{(z, d')\}$, cioè sse esiste $d' \in D$ tale che $(D, \mathcal{I}, \mathcal{V}_{d'}\setminus\{(y, \mathcal{V}_{d'}(y))\} \cup \{(y, d)\}) \models \phi'[^y/_x]$ sfruttando l'ipotesi induttiva, cioè sse $(D, \mathcal{I}, \mathcal{V}\setminus\{(y, \mathcal{V}(y))\} \cup \{(y, d)\}) \models \phi[^y/_x]$.
  - Se $x \neq z$ e $z = y$ allora $(\forall z)(\phi')[^y/_x] = (\forall z')(\phi'_{z'}[^y/_x])$ ed $(\exists z)(\phi')[^y/_x] = (\exists z')(\phi'_{z'}[^y/_x])$ dove $\phi'_{z'} = \phi'[^{z'}/_z]$ con $z' \notin \{x, y\} \cup \mathit{var}(\phi')$. Indicata con $\phi_{z'}$ la formula $(\forall z')(\phi'_{z'})$ oppure $(\exists z')(\phi'_{z'})$, si procede come in uno dei due casi precedenti per dimostrare che per ogni ambiente $(D, \mathcal{I}, \mathcal{V})$ e per ogni $d \in D$ risulta:
    - $(D, \mathcal{I}, \mathcal{V}\setminus\{(z, \mathcal{V}(z))\} \cup \{(z, d)\}) \models \phi$ sse $(D, \mathcal{I}, \mathcal{V}\setminus\{(z', \mathcal{V}(z'))\} \cup \{(z', d)\}) \models \phi_{z'}$ perché $z' \notin \mathit{var}(\phi')$ implica $z' \notin \mathit{varlib}(\phi')$ e $z = y \neq z'$ implica $z \neq z'$.
    - $(D, \mathcal{I}, \mathcal{V}\setminus\{(x, \mathcal{V}(x))\} \cup \{(x, d)\}) \models \phi_{z'}$ sse $(D, \mathcal{I}, \mathcal{V}\setminus\{(y, \mathcal{V}(y))\} \cup \{(y, d)\}) \models \phi_{z'}[^y/_x]$ perché $y \notin \mathit{varlib}(\phi'_{z'})$ e $z' \neq y$.

    Osservato che $z' \notin \{x, y\}$ implica $\phi_{z'}[^y/_x] = \phi[^y/_x]$, il risultato segue dai due fatti sopra.

6.4 Proprietà Algebriche dei Quantificatori

- Nella logica dei predicati i connettivi logici godono delle stesse proprietà algebriche di cui godono nella logica proposizionale. Inoltre vi sono ulteriori proprietà algebriche che riguardano i quantificatori.

- Teorema: Valgono le seguenti proprietà algebriche:
  - $(\forall x)(\phi) \equiv \bigwedge_{t \in Ter}^{var(t)=\emptyset} \phi[^t/_x]$.
  - $(\exists x)(\phi) \equiv \bigvee_{t \in Ter}^{var(t)=\emptyset} \phi[^t/_x]$.
  - $(\neg(\forall x)(\phi)) \equiv (\exists x)(\neg\phi)$.
  - $(\neg(\exists x)(\phi)) \equiv (\forall x)(\neg\phi)$.
  - $(\forall x)(\phi) \equiv (\neg(\exists x)(\neg\phi))$.
  - $(\exists x)(\phi) \equiv (\neg(\forall x)(\neg\phi))$.
  - $(\forall x \forall y)(\phi) \equiv (\forall y \forall x)(\phi)$.
  - $(\exists x \exists y)(\phi) \equiv (\exists y \exists x)(\phi)$.
  - $(\forall x)(\phi) \equiv \phi$ se $x \notin varlib(\phi)$.
  - $(\exists x)(\phi) \equiv \phi$ se $x \notin varlib(\phi)$.
  - $(\forall x)(\phi_1 \wedge \phi_2) \equiv ((\forall x)(\phi_1) \wedge (\forall x)(\phi_2))$.
  - $(\exists x)(\phi_1 \vee \phi_2) \equiv ((\exists x)(\phi_1) \vee (\exists x)(\phi_2))$.
  - $(\forall x)(\phi_1 \vee \phi_2) \equiv ((\forall x)(\phi_1) \vee \phi_2)$ se $x \notin varlib(\phi_2)$.
  - $(\exists x)(\phi_1 \wedge \phi_2) \equiv ((\exists x)(\phi_1) \wedge \phi_2)$ se $x \notin varlib(\phi_2)$.
  - $((\natural_1 x)(\phi_1) \vee (\natural_2 x)(\phi_2)) \equiv (\natural_1 x \natural_2 y)(\phi_1 \vee \phi_2[^y/_x])$ se $y \notin varlib(\phi_1) \cup varlib(\phi_2)$, con $\natural_1, \natural_2 \in \{\forall, \exists\}$.
  - $((\natural_1 x)(\phi_1) \wedge (\natural_2 x)(\phi_2)) \equiv (\natural_1 x \natural_2 y)(\phi_1 \wedge \phi_2[^y/_x])$ se $y \notin varlib(\phi_1) \cup varlib(\phi_2)$, con $\natural_1, \natural_2 \in \{\forall, \exists\}$.
  - $((\forall x)(\phi_1) \rightarrow \phi_2) \equiv (\exists x)(\phi_1 \rightarrow \phi_2)$ se $x \notin varlib(\phi_2)$.
  - $((\exists x)(\phi_1) \rightarrow \phi_2) \equiv (\forall x)(\phi_1 \rightarrow \phi_2)$ se $x \notin varlib(\phi_2)$.
  - $(\phi_1 \rightarrow (\forall x)(\phi_2)) \equiv (\forall x)(\phi_1 \rightarrow \phi_2)$ se $x \notin varlib(\phi_1)$.
  - $(\phi_1 \rightarrow (\exists x)(\phi_2)) \equiv (\exists x)(\phi_1 \rightarrow \phi_2)$ se $x \notin varlib(\phi_1)$.

- Dimostrazione: In virtù del teorema di equivalenza logica, per ogni proprietà si tratta di verificare tramite l'applicazione della definizione della relazione di soddisfacimento che un generico ambiente soddisfa la formula di sinistra sse soddisfa la formula di destra.

- Esempi:
  - La proprietà $(\forall x)(\phi_1 \vee \phi_2) \equiv ((\forall x)(\phi_1) \vee (\forall x)(\phi_2))$ non vale. Si ottiene un controesempio nel dominio dei numeri naturali prendendo $Pari(x)$ come ϕ_1 e $Dispari(x)$ come ϕ_2: mentre è vero che ogni numero è pari o dispari, non è vero che ogni numero è pari oppure che ogni numero è dispari.
  - La proprietà $(\exists x)(\phi_1 \wedge \phi_2) \equiv ((\exists x)(\phi_1) \wedge (\exists x)(\phi_2))$ non vale. Si ottiene un controesempio nel dominio dei numeri naturali prendendo di nuovo $Pari(x)$ come ϕ_1 e $Dispari(x)$ come ϕ_2: mentre non è vero che esiste un numero che è sia pari che dispari, è vero che esiste un numero pari e che esiste un numero dispari.

6.5 Sistemi Deduttivi per la Logica dei Predicati

- Il sistema di deduzione naturale di Gentzen si estende alla logica dei predicati attraverso le seguenti regole di inferenza aggiuntive:
  - Regole che introducono quantificatori rispetto alle premesse:
    $$\frac{P[^{y}/_{x}]}{(\forall x)(P)} \qquad \frac{P[^{t}/_{x}]}{(\exists x)(P)}$$
    dove il simbolo di predicato P denota una formula atomica in cui y non occorre libera. La prima regola introduce un quantificatore universale perché y è una generica variabile che sostituisce x, mentre la seconda regola introduce una quantificazione esistenziale perché t è uno specifico termine che sostituisce x.
  - Regole che eliminano quantificatori rispetto alle premesse:
    $$\frac{(\forall x)(P)}{P[^{t}/_{x}]} \qquad \frac{(\exists x)(P) \quad [P[^{y}/_{x}]]Q}{Q}$$
    dove i simboli di predicato P e Q denotano due formule atomiche in cui y non occorre libera. La prima regola è simile a quella per eliminare $\wedge$, mentre la seconda regola è simile a quella per eliminare $\vee$ ed è condizionale perché da $(\exists x)(P)$ non si può inferire una specifica variante di P, ma ogni Q valida sotto qualsiasi variante di P.
- Teorema: Il sistema di deduzione naturale di Gentzen esteso è corretto e completo rispetto alla logica dei predicati.
- Un sistema deduttivo alla Hilbert si estende alla logica dei predicati attraverso la seguente regola di inferenza aggiuntiva detta regola di generalizzazione:
  $$\frac{P[^{y}/_{x}]}{(\forall x)(P)}$$
  dove il simbolo di predicato P denota una formula atomica in cui y non occorre libera, assieme ai seguenti assiomi aggiuntivi:
  $$\frac{}{(P[^{t}/_{x}] \rightarrow (\exists x)(P))}$$
  $$\frac{}{((\forall x)(P) \rightarrow P[^{t}/_{x}])}$$
  $$\frac{}{((\forall x)(P \rightarrow Q) \rightarrow ((\exists x)(P) \rightarrow Q))}$$
  $$\frac{}{((\forall x)(Q \rightarrow P) \rightarrow (Q \rightarrow (\forall x)(P)))}$$
  dove i simboli di predicato P e Q denotano due formule atomiche tali che x non occorre libera in Q.
- Teorema: Il sistema deduttivo alla Hilbert esteso è corretto e completo rispetto alla logica dei predicati.
- Teorema (di deduzione): Siano $\Phi \in 2^{FBF}$ finito e $\phi, \gamma \in FBF$. Se $\Phi \cup \{\phi\} \vdash_{\text{HE}} \gamma$, allora $\Phi \vdash_{\text{HE}} (\phi \rightarrow \gamma)$. ■fltplf_14

Capitolo 7

Refutazione di Formule Logiche

7.1 Forme Normali per la Logica Proposizionale e dei Predicati

- Il problema di stabilire se una formula ben formata è valida (tautologia) o insoddisfacibile (contraddizione) è risolvibile seppur in tempo esponenziale nel caso della logica proposizionale, mentre è soltanto semi-decidibile nel caso della logica dei predicati. Il problema della soddisfacibilità è anch'esso intrattabile nel primo caso, mentre non è nemmeno semi-decidibile nel secondo. Una tecnica per affrontare l'elevato costo computazionale o la parziale decidibilità che – a seconda della logica – caratterizza l'analisi della verità di una formula ben formata consiste nel trasformare la formula in un'altra formula equivalente espressa in un formato, solitamente detto forma normale, che è più semplice da studiare.

- Denotiamo con FBF_{prop} ed FBF_{pred} l'insieme delle formule ben formate della logica proposizionale e della logica dei predicati, rispettivamente. Limitiamo inoltre l'uso delle parentesi al minimo.

- Data una formula $\phi \in FBF_{\text{prop}}$, diciamo che essa è in:

  - forma normale disgiuntiva sse è della forma $\bigvee_{i=1}^{n} (\bigwedge_{j=1}^{m_i} \lambda_{i,j})$ con $n \geq 1$ ed $m_i \geq 1$ per ogni $1 \leq i \leq n$
  - forma normale congiuntiva sse è della forma $\bigwedge_{i=1}^{n} (\bigvee_{j=1}^{m_i} \lambda_{i,j})$ con $n \geq 1$ ed $m_i \geq 1$ per ogni $1 \leq i \leq n$

  dove ciascun $\lambda_{i,j}$, detto letterale, è della forma p oppure $\neg p$ con $p \in \mathit{Prop}$.

- Grazie all'associatività e alla commutatività dei connettivi logici $\vee$ e $\wedge$ (vedi Sez. 5.4), l'ordine dei letterali all'interno di una forma normale disgiuntiva o congiuntiva è irrilevante. Poiché l'insieme di connettivi logici $\{\vee, \wedge, \neg\}$ è funzionalmente completo (vedi Sez. 5.4), ogni formula ben formata della logica proposizionale può essere trasformata in forma normale disgiuntiva e in forma normale congiuntiva.

- La trasformazione sfrutta il teorema di sostituzione (vedi Sez. 5.3) e procede nel seguente modo:

  1. Rimpiazzare tutti i connettivi logici diversi da $\vee$, $\wedge$, $\neg$ applicando le leggi di derivabilità semantica (vedi Sez. 5.4).
  2. Usare ripetutamente le leggi di De Morgan e la legge di doppia inversione (vedi Sez. 5.4) per portare tutti i connettivi logici $\neg$ immediatamente davanti alle proposizioni.
  3. Usare ripetutamente la distributività dei connettivi logici $\vee$ e $\wedge$ (vedi Sez. 5.4) per ottenere una disgiunzione di congiunzioni o una congiunzione di disgiunzioni a seconda della forma normale.

- Esempio: La formula $(p \vee \neg q) \rightarrow r$ viene trasformata prima in $\neg(p \vee \neg q) \vee r$ e poi in $(\neg p \wedge q) \vee (r)$ che è già in forma normale disgiuntiva. Applicando la proprietà distributiva si ottiene l'equivalente forma normale congiuntiva $(\neg p \vee r) \wedge (q \vee r)$.

- Un procedimento più meccanico, che però genera di solito forme normali più lunghe, per trasformare una formula della logica proposizionale in forma normale disgiuntiva o congiuntiva è il seguente:

  1. Costruire la tabella di verità della formula.

  $2^{\vee}$. Nel caso della forma normale disgiuntiva, per ogni riga il cui ultimo elemento è 1 formare una congiunzione considerando per ogni proposizione p come letterale p stesso oppure $\neg p$ a seconda che il valore di verità assegnato a p in quella riga sia 1 oppure 0, poi prendere la disgiunzione di tutte queste congiunzioni.

  $2^{\wedge}$. Nel caso della forma normale congiuntiva, per ogni riga il cui ultimo elemento è 0 formare una disgiunzione considerando per ogni proposizione p come letterale p stesso oppure $\neg p$ a seconda che il valore di verità assegnato a p in quella riga sia 0 oppure 1, poi prendere la congiunzione di tutte queste disgiunzioni.

- Esempio: Data la formula $(p \vee \neg q) \rightarrow r$, una volta costruita la sua tabella di verità:

$\mathcal{I}_A(p)$	$\mathcal{I}_A(q)$	$\mathcal{I}_A(r)$	$\mathcal{I}_A(p \vee \neg q)$	$\mathcal{I}_A((p \vee \neg q) \rightarrow r)$
1	1	1	1	1
1	1	0	1	0
1	0	1	1	1
1	0	0	1	0
0	1	1	0	1
0	1	0	0	1
0	0	1	1	1
0	0	0	1	0

  si ottengono $(p \wedge q \wedge r) \vee (p \wedge \neg q \wedge r) \vee (\neg p \wedge q \wedge r) \vee (\neg p \wedge q \wedge \neg r) \vee (\neg p \wedge \neg q \wedge r)$ come forma normale disgiuntiva e $(\neg p \vee \neg q \vee r) \wedge (\neg p \vee q \vee r) \wedge (p \vee q \vee r)$ come forma normale congiuntiva. Queste due forme normali sono più complesse di quelle ottenute nell'esempio precedente.

- La (in)soddisfacibilità di una formula in forma normale disgiuntiva può essere stabilita in tempo polinomiale. Infatti una formula siffatta è soddisfacibile sse contiene almeno una congiunzione di letterali che è soddisfacibile, cioè che non comprende due letterali della forma p e $\neg p$. Questo fatto può certamente essere verificato impiegando un tempo che cresce linearmente con il numero di letterali presenti nella formula, senza costruire la tabella di verità. Il problema della (in)soddisfacibilità rimane invece risolvibile in tempo esponenziale per le formule in forma normale congiuntiva.

- La validità di una formula in forma normale congiuntiva può essere stabilita in tempo polinomiale. Infatti una formula siffatta è valida sse ogni sua disgiunzione di letterali è valida, cioè comprende due letterali della forma p e $\neg p$. Questo fatto può certamente essere verificato impiegando un tempo che cresce linearmente con il numero di letterali presenti nella formula, senza costruire la tabella di verità. Il problema della validità rimane invece risolvibile in tempo esponenziale per le formule in forma normale disgiuntiva. In effetti ogni formula siffatta è valida sse la sua negazione è insoddisfacibile, ma quest'ultima formula diventa immediatamente in forma normale congiuntiva applicando le leggi di De Morgan e quindi sappiamo stabilirne la insoddisfacibilità solo in tempo esponenziale.

- Diciamo che un letterale λ è positivo oppure negativo a seconda che sia della forma p oppure $\neg p$ con $p \in \mathit{Prop}$. Chiamiamo clausola una disgiunzione di letterali. Chiamiamo poi clausola di Horn (risp. clausola di Horn definita) una clausola che contiene al più (risp. esattamente) un letterale positivo, la quale viene classificata come:

  - un fatto se è della forma p, che è equivalente a $\mathbb{I} \rightarrow p$;
  - un vincolo se è della forma $\bigvee_{i=1}^{n} \neg p_i$, che è equivalente a $(\bigwedge_{i=1}^{n} p_i) \rightarrow \mathbb{O}$;
  - una regola se è della forma $p \vee (\bigvee_{i=1}^{n} \neg p_i)$, che è equivalente a $(\bigwedge_{i=1}^{n} p_i) \rightarrow p$.

- Una formula in forma normale congiuntiva è detta essere una formula di Horn sse ogni sua clausola è una clausola di Horn. Osserviamo che:
  - Una formula di Horn priva di fatti è soddisfacibile, come si vede prendendo un qualsiasi assegnamento di verità che non contiene nessuna delle proposizioni che compaiono nella formula.
  - Una formula di Horn priva di vincoli è soddisfacibile, come si vede prendendo un qualsiasi assegnamento di verità che contiene tutte le proposizioni che compaiono nei letterali positivi della formula.
- La (in)soddisfacibilità di una formula in forma normale congiuntiva che è anche una formula di Horn può essere stabilita in tempo polinomiale attraverso il seguente algoritmo:
  1. Trasformare la formula ϕ nella formula equivalente in cui ogni clausola viene ad avere il connettivo logico $\rightarrow$ come connettivo primario e tutti i letterali in forma positiva.
  2. Prendere un assegnamento di verità A inizialmente vuoto.
  3. Aggiungere ad A ogni proposizione p che compare come fatto in ϕ.
  4. Finché A non è stabile, per ogni regola $(\bigwedge_{i=1}^{n} p_i) \rightarrow p$ presente in ϕ tale che ogni $p_i \in A$ ma $p \notin A$, aggiungere p ad A.
  5. Se esiste un vincolo $(\bigwedge_{i=1}^{n} p_i) \rightarrow \mathbb{O}$ in ϕ tale che ogni $p_i \in A$, allora ϕ è insoddisfacibile, altrimenti ϕ è soddisfatta da A.

  Se ϕ è soddisfacibile, allora A è il minimo assegnamento di verità che soddisfa ϕ.
- Esempio: La formula di Horn $(p \vee \neg q \vee \neg r) \wedge (\neg s \vee q) \wedge (\neg s \vee r) \wedge (s) \wedge (\neg p' \vee \neg s)$, che è equivalente a $((q \wedge r) \rightarrow p) \wedge (s \rightarrow q) \wedge (s \rightarrow r) \wedge (\mathbb{I} \rightarrow s) \wedge ((p' \wedge s) \rightarrow \mathbb{O})$, è soddisfacibile perché risulta:
  - $A = \{s\}$ dopo aver esaminato l'unico fatto presente nella formula.
  - $A = \{s, q, r\}$ dopo aver esaminato per la prima volta le tre regole presenti nella formula.
  - $A = \{s, q, r, p\}$ dopo aver esaminato per la seconda volta le tre regole presenti nella formula.
  - $p' \notin A$ per quanto riguarda l'unico vincolo presente nella formula.
- In conclusione, per una formula $\phi \in FBF_{\text{prop}}$ in forma normale congiuntiva che è anche una formula di Horn è possibile stabilire in tempo polinomiale sia la sua validità che la sua (in)soddisfacibilità. Tuttavia non tutte le formule di logica proposizionale sono trasformabili in formule di Horn (p.e. $p \vee q$).
- Data una formula $\phi \in FBF_{\text{pred}}$, diciamo che essa è in forma normale rettificata sse non ha variabili che occorrono sia legate che libere – cioè $varleg(\phi) \cap varlib(\phi) = \emptyset$ – e non ha molteplici quantificatori che si riferiscono alla stessa variabile. In virtù dell'ultimo teorema della Sez. 6.3, ogni formula ben formata della logica dei predicati può essere trasformata in forma normale rettificata, così da evitare confusione tra le variabili.
- Data una formula $\phi \in FBF_{\text{pred}}$, diciamo che essa è in forma normale prenessa sse è della forma $\natural_1 x_1 \ldots \natural_n x_n \, \phi'$ dove $n \geq 0$, ogni $\natural_i \in \{\forall, \exists\}$ e ϕ' è priva di quantificatori. La sequenza di quantificatori è detta il prefisso della formula, mentre ϕ' è detta la matrice della formula. In virtù delle proprietà algebriche dei quantificatori (vedi Sez. 6.4) e del teorema di sostituzione (vedi Sez. 6.3), ogni formula ben formata della logica dei predicati può essere trasformata in forma normale prenessa, così da raccogliere in testa tutti i quantificatori.
- Esempio: La formula $\forall x \, P(x) \rightarrow \neg \forall y \, Q(y)$ è equivalente a $\forall x \, P(x) \rightarrow \exists y \, \neg Q(y)$, che è equivalente a $\exists y \, (\forall x \, P(x) \rightarrow \neg Q(y))$, che è equivalente a $\exists y \exists x \, (P(x) \rightarrow \neg Q(y))$, che è in forma normale prenessa.
- Data una formula $\phi \in FBF_{\text{pred}}$, diciamo che essa è in forma normale di Skolem sse è della forma $\forall x_1 \ldots \forall x_n \, \phi'$ dove $n \geq 0$ e ϕ' è priva di quantificatori. Ogni formula in forma normale prenessa può essere trasformata in forma normale di Skolem – eliminando quindi tutti i suoi quantificatori esistenziali – introducendo degli opportuni simboli di costante e di funzione detti costanti e funzioni di Skolem. Diversamente dalle precedenti forme normali, la formula ottenuta non è equivalente a quella originaria ma preserva la (in)soddisfacibilità di quest'ultima, cioè è (in)soddisfacibile sse quella originaria lo è (la forma normale duale $\exists x_1 \ldots \exists x_n \, \phi'$, detta forma normale di Herbrand, preserva invece la validità).

- L'algoritmo di skolemizzazione applicato a una formula $\phi \in FBF_{\text{pred}}$ in forma normale prenessa procede come segue (l'algoritmo di herbrandizzazione procede in modo analogo):
  - Ogni quantificatore $\exists x$ presente in ϕ che non è nel campo d'azione di un quantificatore universale viene rimosso da ϕ dopo aver applicato alla sua matrice ϕ' la sostituzione $[{}^{a}/_{x}]$ dove a è una costante che non compare in ϕ'.
  - Ogni quantificatore $\exists x$ presente in ϕ che è nel campo d'azione dei quantificatori universali $\forall x_1 ... \forall x_m$ viene rimosso da ϕ dopo aver applicato alla sua matrice ϕ' la sostituzione $[{}^{f(x_1,\ldots,x_m)}/_{x}]$ dove f è un simbolo di funzione che non compare in ϕ'.
- Esempio: La formula in forma normale prenessa $\exists x \forall y \exists z\, P(x, y, z, f(a))$ ha come forma normale di Skolem $\forall y\, P(b, y, g(y), f(a))$. Osserviamo che $(D, \mathcal{I}, \mathcal{V}) \models \exists x \forall y \exists z\, P(x, y, z, f(a))$ sse esiste $d_1 \in D$ tale che per ogni $d_2 \in D$ esiste $d_3 \in D$ tale che $(D, \mathcal{I}, \mathcal{V} \setminus \{(x, \mathcal{V}(x)), (y, \mathcal{V}(y)), (z, \mathcal{V}(z))\} \cup \{(x, d_1), (y, d_2), (z, d_3)\}) \models P(x, y, z, f(a))$, cioè sse per ogni $d_2 \in D$ risulta $(D, \mathcal{I}, \mathcal{V} \setminus \{(y, \mathcal{V}(y))\} \cup \{(y, d_2)\}) \models P(b, y, g(y), f(a))$ ponendo $\mathcal{I}(b) = d_1$ e $\mathcal{I}(g(y)) = d_3$, cioè sse $(D, \mathcal{I}, \mathcal{V}) \models \forall y\, P(b, y, g(y), f(a))$.
- Data una formula $\phi \in FBF_{\text{pred}}$, al fine di agevolare la verifica della sua (in)soddisfacibilità – o della sua validità studiando l'insoddisfacibilità di $\neg\phi$ – conviene eseguire i seguenti passi preliminari:
  1. Trasformare ϕ in una formula ρ equivalente in forma normale rettificata.
  2. Calcolare la chiusura esistenziale di ρ, che è una formula chiusa soddisfacibile sse ρ è soddisfacibile.
  3. Trasformare $ce(\rho)$ in una formula ξ equivalente in forma normale prenessa.
  4. Trasformare ξ in una formula κ in forma normale di Skolem, che è soddisfacibile sse lo è ξ (e quindi ϕ).

  Denotiamo con $FBF_{\text{pred,crs}}$ l'insieme delle formule di FBF_{pred} che sono chiuse e in forma normale rettificata e di Skolem.

7.2 Unificazione di Formule di Logica dei Predicati

- Anche l'unificazione di formule logiche ha un ruolo nello studio della validità e della (in)soddisfacibilità. Essa richiede di generalizzare la nozione di sostituzione sintattica, introdotta in Sez. 6.1, come una sequenza finita ed eventualmente vuota – nel qual caso è denotata con ε – di legami $[{}^{t_1}/_{x_1}, \ldots, {}^{t_n}/_{x_n}]$ dove:
  - $t_i \in Ter$ e $x_i \in Var$ per ogni $i = 1, \ldots, n$.
  - $t_i \neq x_i$ per ogni $i = 1, \ldots, n$ (assenza di legami inutili).
  - $x_i \neq x_j$ per ogni $i, j = 1, \ldots, n$ tali che $i \neq j$ (assenza di legami in conflitto).
- Sia Θ l'insieme di tutte le sostituzioni basate su Ter e Var. Data $\theta \in \Theta$, indichiamo con $t\theta$ il termine ottenuto da $t \in Ter$ applicandogli θ, con $\phi\theta$ la formula ottenuta da $\phi \in FBF_{\text{pred}}$ applicandole θ e con $\Phi\theta$ l'insieme di formule ottenuto da $\Phi \in 2^{FBF_{\text{pred}}}$ applicando θ a tutte le formule di Φ.
- Esempio: Se ϕ è la formula $P(x, f(y)) \wedge Q(z)$ e $\theta = [{}^{c}/_{x}, {}^{g(a)}/_{y}, {}^{z'}/_{z}]$, allora $\phi\theta = P(c, f(g(a))) \wedge Q(z')$.
- Date due sostituzioni $\theta_1 = [{}^{t_1}/_{x_1}, \ldots, {}^{t_n}/_{x_n}]$ e $\theta_2 = [{}^{u_1}/_{y_1}, \ldots, {}^{u_m}/_{y_m}]$, la composizione di θ_1 e θ_2, denotata con $\theta_1 \circ \theta_2$, è ottenuta da $[{}^{t_1\theta_2}/_{x_1}, \ldots, {}^{t_n\theta_2}/_{x_n}, {}^{u_1}/_{y_1}, \ldots, {}^{u_m}/_{y_m}]$ eliminando ogni legame ${}^{t_i\theta_2}/_{x_i}$ tale che $t_i\theta_2 = x_i$ così come ogni legame ${}^{u_j}/_{y_j}$ tale che $y_j \in \{x_1, \ldots, x_n\}$.
- La composizione di sostituzioni ha ε come elemento neutro ed è associativa ma non commutativa. Inoltre risulta $\phi(\theta_1 \circ \theta_2) = (\phi\theta_1)\theta_2$ per ogni $\phi \in FBF_{\text{pred}}$ e $\theta_1, \theta_2 \in \Theta$, quindi $[{}^{t_1}/_{x_1}, \ldots, {}^{t_n}/_{x_n}]$ può essere vista come $[{}^{t_1}/_{x_1}] \circ \cdots \circ [{}^{t_n}/_{x_n}]$.
- Esempio: Se $\theta_1 = [{}^{f(z')}/_{x}, {}^{b}/_{y}, {}^{y}/_{z}]$ e $\theta_2 = [{}^{c}/_{z'}, {}^{a}/_{y}, {}^{b}/_{z}]$, allora $\theta_1 \circ \theta_2 = [{}^{f(c)}/_{x}, {}^{b}/_{y}, {}^{a}/_{z}, {}^{c}/_{z'}]$ mentre $\theta_2 \circ \theta_1 = [{}^{c}/_{z'}, {}^{a}/_{y}, {}^{b}/_{z}, {}^{f(z')}/_{x}]$.

- Dati un insieme di formule $\Phi \in 2^{FBF_{\text{pred}}}$ e una sostituzione $\theta \in \Theta$, diciamo che θ è un unificatore di Φ sse $\phi_1\theta = \phi_2\theta$ per ogni $\phi_1, \phi_2 \in \Phi$. In questo caso diciamo che Φ è unificabile.
- È evidente che un insieme di formule sintatticamente diverse non può essere unificabile se in quelle formule non compaiono occorrenze libere di variabili. Il problema dell'unificazione è certamente rilevante quando si ha a che fare con le matrici di formule in forma normale prenessa.
- Dati un insieme unificabile di formule $\Phi \in 2^{FBF_{\text{pred}}}$ e un suo unificatore $\theta \in \Theta$, diciamo che θ è l'unificatore più generale (upg) di Φ sse per ogni unificatore θ' di Φ esiste una sostituzione θ'' tale che $\theta' = \theta \circ \theta''$. Intuitivamente, l'upg è l'unificatore col minor numero di legami. Quando esiste, l'upg è unico a meno di ridenominazione delle variabili.
- Esempio: L'insieme di formule $\Phi = \{P(x, a), P(y, b)\}$ non è unificabile perché a e b sono due costanti diverse e non esiste nessuna sostituzione che le trasforma nella stessa costante. L'insieme di formule $\Phi' = \{P(x, a), P(y, a)\}$ è invece unificabile e due possibili unificatori per esso sono $\theta_1 = [{}^{b}/_{x}, {}^{b}/_{y}]$ e $\theta_2 = [{}^{y}/_{x}]$. Osserviamo che θ_2 è più generale di θ_1 in quanto θ_1 può essere ottenuto da θ_2 componendo quest'ultimo con la sostituzione $[{}^{b}/_{y}]$. Risulta che $[{}^{y}/_{x}]$ e $[{}^{x}/_{y}]$ sono entrambi upg di Φ'.
- Nel 1965 Alan Robinson sviluppò un algoritmo per calcolare l'upg, se esiste, di un insieme di formule ben formate. L'idea dell'algoritmo è di attraversare da sinistra a destra le formule fino a raggiungere simboli differenti e tentando a quel punto di unificare le sottoformule che iniziano con quei simboli. L'algoritmo risultante non è deterministico, nel senso che per lo stesso insieme di formule può generare upg che differiscono per i nomi delle variabili. Inoltre l'algoritmo è intuitivo ma inefficiente, in quanto la complessità nel caso pessimo è esponenziale rispetto alla lunghezza delle formule. Algoritmi di unificazione più efficienti sono stati sviluppati successivamente; ad esempio l'algoritmo di unificazione di Martelli e Montanari del 1982 ha complessità quasi lineare.
- Dato un insieme finito $\Phi \in 2^{FBF_{\text{pred}}}$, l'algoritmo di unificazione di Robinson procede come segue:

  1. Presumere che Φ sia unificabile.
  2. Assegnare la sostituzione vuota ε a θ.
  3. Finché Φ è presumibilmente unificabile e $|\Phi\theta| > 1$, ripetere i seguenti passi:
     (a) Attraversare da sinistra a destra le formule di $\Phi\theta$ fino a incontrare la posizione più a sinistra in cui le formule differiscono.
     (b) Se qualcuno dei simboli che occupano quella posizione nelle varie formule di $\Phi\theta$ non è un termine oppure nessuno di essi è una variabile, allora Φ non è unificabile, altrimenti:
        i. siano $x \in \mathit{Var}$ e $t \in \mathit{Ter}$ due di quei simboli con $t \neq x$ (scelta non deterministica);
        ii. se x compare in t allora Φ non è unificabile, altrimenti assegnare $\theta \circ [{}^{t}/_{x}]$ a θ.
  4. Se $|\Phi\theta| > 1$ segnalare che Φ non è unificabile, altrimenti segnalare che θ è l'upg.
- Il passo (ii) dell'algoritmo applicato a termini diversi è rappresentabile attraverso la seguente tabella:

$\neq$	c_2	x_2	ξ_2
c_1	non unificabili	$[{}^{c_1}/_{x_2}]$	non unificabili
x_1	$[{}^{c_2}/_{x_1}]$	$[{}^{x_1}/_{x_2}]$ oppure $[{}^{x_2}/_{x_1}]$	$[{}^{\xi_2}/_{x_1}]$ previo occur check
ξ_1	non unificabili	$[{}^{\xi_1}/_{x_2}]$ previo occur check	$(*)$ ricorsione più coerenza

  dove:

  - $c_1, c_2 \in \mathit{Cos}$.
  - $x_1, x_2 \in \mathit{Var}$.
  - $\xi_1, \xi_2 \in \mathit{Ter} \setminus (\mathit{Cos} \cup \mathit{Var})$.
  - Per occur check si intende il controllo consistente nel verificare che x_i non compaia in ξ_j, altrimenti non è possibile unificare. Tale controllo è necessario per garantire la terminazione dell'algoritmo; in sua assenza, l'algoritmo tenterebbe invano di unificare termini come ad esempio x ed $f(x)$ senza mai arrestarsi.

  - Nel caso $(*)$, ξ_1 e ξ_2 non sono unificabili se il simbolo di funzione all'inizio di ξ_1 differisce dal simbolo di funzione all'inizio di ξ_2, altrimenti bisogna procedere ricorsivamente sugli argomenti a cui è applicato il simbolo di funzione all'inizio di ξ_1 e di ξ_2 verificando che gli upg per le varie coppie di argomenti siano coerenti tra loro. Ad esempio $g(1,2)$ e $g(y,y)$ non sono unificabili: considerando separatamente le loro due coppie di argomenti corrispondenti si avrebbero i due legami confliggenti ${}^{1}/_{y}$ e ${}^{2}/_{y}$.

- Teorema (di Robinson): Ogni insieme unificabile di formule ha l'upg.

- Dimostrazione: Si tratta di verificare che l'algoritmo di unificazione di Robinson termina per ogni insieme finito di formule e, nel caso in cui l'insieme sia unificabile, calcola l'upg dell'insieme.
  Sia $\Phi \in 2^{FBF_{\text{pred}}}$ un insieme finito di formule. Poiché Φ contiene un numero finito di variabili distinte, l'algoritmo termina sempre. Infatti, in una generica iterazione, o si stabilisce che Φ non è unificabile, o il numero di variabili distinte viene ridotto di uno, quindi il numero di iterazioni non può superare il numero di variabili distinte che compaiono nelle formule di Φ.
  Se Φ è unificabile l'algoritmo termina con successo e θ è un unificatore di Φ. Questo unificatore è proprio l'upg di Φ perché, indicato con θ_i il valore di θ dopo i iterazioni dell'algoritmo, per ogni altro unificatore θ' di Φ esiste una sostituzione θ''_i tale che $\theta' = \theta_i \circ \theta''_i$ come si vede procedendo per induzione su $i \in \mathbb{N}$:

  - Se $i = 0$ allora posto $\theta''_i = \theta'$ risulta $\theta' = \theta_i \circ \theta''_i$ perché $\theta_i = \varepsilon$.
  - Sia $i \geq 0$ e supponiamo che esista θ''_i tale che $\theta' = \theta_i \circ \theta''_i$. Ci sono due casi:
    * Se $\Phi\theta_i$ contiene una sola formula, allora il risultato segue banalmente dal fatto che in questo caso l'algoritmo termina.
    * Se $\Phi\theta_i$ contiene più di una formula, allora l'algoritmo esegue un'ulteriore iterazione. Poiché Φ è unificabile, esisteranno $x \in \mathit{Var}$ e $t \in \mathit{Ter}$ tali che x non compare in t e quindi $\theta_{i+1} = \theta_i \circ [{}^{t}/_{x}]$. Sia θ''_{i+1} la sostituzione ottenuta da θ''_i eliminando $[{}^{t\theta''_i}/_{x}]$. Allora $\theta_{i+1} \circ \theta''_{i+1} = \theta_i \circ [{}^{t}/_{x}] \circ \theta''_{i+1} = \theta_i \circ \theta''_{i+1} \circ [{}^{t\theta''_{i+1}}/_{x}]$ perché x non viene rimpiazzata in θ''_{i+1}. Poiché $\theta_i \circ \theta''_{i+1} \circ [{}^{t\theta''_{i+1}}/_{x}] = \theta_i \circ \theta''_{i+1} \circ [{}^{t\theta''_i}/_{x}]$ in quanto x non compare in t, risulta $\theta_{i+1} \circ \theta''_{i+1} = \theta_i \circ \theta''_{i+1} \circ [{}^{t\theta''_i}/_{x}] = \theta_i \circ \theta''_i = \theta'$ sfruttando l'ipotesi induttiva.

- Esempi:

  - L'insieme di formule:
    $$\begin{aligned}\Phi \;=\; \{&P(a, f(x,c), y),\\ &P(z, f(g(z'), y), y),\\ &P(a, f(g(a), y), c)\}\end{aligned}$$
    è unificabile perché:
    * Partendo con $\theta = \varepsilon$, le tre formule di $\Phi\theta$ differiscono sui simboli a, z, a, i quali sono unificabili ponendo $\theta = [{}^{a}/_{z}]$ alla prima iterazione dell'algoritmo.
    * Le tre formule di $\Phi\theta$ differiscono sui simboli $x, g(z'), g(a)$, i quali sono unificabili ponendo $\theta = [{}^{a}/_{z}, {}^{g(z')}/_{x}]$ alla seconda iterazione dell'algoritmo e $\theta = [{}^{a}/_{z}, {}^{g(z')}/_{x}, {}^{a}/_{z'}]$ alla terza iterazione dell'algoritmo.
    * Le tre formule di $\Phi\theta$ differiscono sui simboli c, y, y, i quali sono unificabili ponendo $\theta = [{}^{a}/_{z}, {}^{g(z')}/_{x}, {}^{a}/_{z'}, {}^{c}/_{y}]$ alla quarta iterazione dell'algoritmo. Questa sostituzione è l'upg di Φ e riduce tutte le formule di Φ alla unica formula $P(a, f(g(a), c), c)$.
  - L'insieme di formule:
    $$\begin{aligned}\Phi \;=\; \{&P(x_1, x_2, \ldots, x_n),\\ &P(f(x_0, x_0), f(x_1, x_1), \ldots, f(x_{n-1}, x_{n-1}))\}\end{aligned}$$
    è unificabile ma l'algoritmo di unificazione di Robinson impiega un tempo esponenziale rispetto a n per stabilirlo. Infatti, l'upg di Φ contiene legami quali ${}^{f(x_0,x_0)}/_{x_1}, {}^{f(f(x_0,x_0),f(x_0,x_0))}/_{x_2}, \ldots$ dove il termine che sostituisce x_i, $1 \leq i \leq n$, contiene 2^i occorrenze di variabili e quindi l'occur check che deve essere effettuato prima di applicare la sostituzione a x_i impiega un tempo proporzionale a 2^i. ■fltplf_15

7.3 Teoria di Herbrand e Algoritmo di Refutazione

- Per stabilire la validità di una formula ben formata della logica dei predicati semanticamente, cioè non per via deduttiva, bisogna assicurare che la formula sia soddisfatta da ogni possibile ambiente. In generale, verificare la validità di una formula della logica dei predicati è più complicato che verificarne l'insoddisfacibilità. Di conseguenza, data una formula $\phi \in FBF_{\text{pred}}$, per stabilirne la validità conviene procedere per refutazione, cioè verificare che nessun ambiente soddisfa $\neg\phi$ lavorando sulla corrispondente formula appartenente a $FBF_{\text{pred,crs}}$ che ne preserva l'insoddisfacibilità.
- Poichè il numero di interpretazioni è illimitato e i domini delle interpretazioni possono essere insiemi infiniti, la suddetta verifica è fattibile solo se esiste un dominio canonico, cioè un dominio tale che una formula della logica dei predicati è insoddisfacibile sse è falsa in ogni interpretazione su tale dominio. Nel 1930 Jacques Herbrand scoprì che tale dominio esiste ed è costruibile a partire dai simboli di costante e di funzione presenti nella formula, aprendo così la via agli algoritmi di refutazione.
- Diciamo che un termine $t \in Ter$ è ground sse $var(t) = \emptyset$ e indichiamo con Ter_{g} l'insieme dei termini ground. Analogamente diciamo che una formula $\phi \in FBF_{\text{pred}}$ è ground sse $var(\phi) = \emptyset$ e indichiamo con $FBF_{\text{pred,g}}$ l'insieme delle formule ground e con $FBF_{\text{pred,a,g}}$ l'insieme delle formule atomiche ground.
- Data una formula $\phi \in FBF_{\text{pred}}$:
  - L'universo di Herbrand di ϕ è il più piccolo sottoinsieme $U_{\text{H}}(\phi)$ di Ter_{g} tale che:
    - $a \in U_{\text{H}}(\phi)$ per ogni $a \in Cos$ presente in ϕ; se ϕ non contiene costanti, scegliamo una costante $c \in Cos$ e poniamo $c \in U_{\text{H}}(\phi)$.
    - $f(u_1, ..., u_n) \in U_{\text{H}}(\phi)$ per ogni $f \in Fun$ presente in ϕ con $n \geq 1$ argomenti e $u_1, ..., u_n \in U_{\text{H}}(\phi)$.
  - La base di Herbrand di ϕ è il più piccolo sottoinsieme $B_{\text{H}}(\phi)$ di $FBF_{\text{pred,a,g}}$ tale che:
    - $P(u_1, ..., u_n) \in B_{\text{H}}(\phi)$ per ogni $P \in Pred$ presente in ϕ con $n \geq 1$ argomenti e $u_1, ..., u_n \in U_{\text{H}}(\phi)$.

  Questi due insiemi sono finiti sse ϕ non contiene simboli di funzione.
- Esempio: Se ϕ è la formula $\forall x\, P(f(x))$, allora $U_{\text{H}}(\phi)$ comprende c, $f(c)$, $f(f(c))$, $f(f(f(c)))$ e così via, mentre $B_{\text{H}}(\phi)$ comprende $P(c)$, $P(f(c))$, $P(f(f(c)))$, $P(f(f(f(c))))$ e così via.
- Data una formula $\phi \in FBF_{\text{pred}}$, chiamiamo interpretazione di Herbrand di ϕ ogni ambiente $(U_{\text{H}}(\phi), \mathcal{I}, \mathcal{V})$ tale che:
  - $\mathcal{I}(a) = a$ per ogni $a \in Cos$ appartenente a $U_{\text{H}}(\phi)$.
  - $\mathcal{I}(f) = f$ per ogni $f \in Fun$ presente in ϕ.

  Diciamo che $(U_{\text{H}}(\phi), \mathcal{I}, \mathcal{V})$ è un modello di Herbrand di ϕ sse $(U_{\text{H}}(\phi), \mathcal{I}, \mathcal{V}) \models \phi$.
- L'interpretazione di Herbrand di una formula $\phi \in FBF_{\text{pred}}$ non deve soggiacere a nessuna restrizione su nessun predicato P presente in ϕ, quindi due diverse interpretazioni di Herbrand di ϕ differiscono soltanto per il modo in cui vengono interpretati i simboli di predicato. Poiché gli elementi di $B_{\text{H}}(\phi)$ sono assimilabili a formule atomiche di logica proposizionale, ogni interpretazione di Herbrand di ϕ può essere vista come un sottoinsieme A di $B_{\text{H}}(\phi)$ assumendo che una sottoformula atomica di ϕ è vera sse ogni sua istanza ground appartiene ad A. Il seguente teorema e il suo corollario stabiliscono che $U_{\text{H}}(\phi)$ è un dominio canonico.
- Teorema: Sia $\phi \in FBF_{\text{pred,crs}}$. Per ogni ambiente esiste una corrispondente interpretazione di Herbrand di ϕ tale che se l'ambiente soddisfa ϕ allora anche quell'interpretazione di Herbrand soddisfa ϕ.
- Dimostrazione: Sia $\phi \in FBF_{\text{pred,crs}}$. Dato un ambiente $(D, \mathcal{I}, \mathcal{V})$, l'interpretazione di Herbrand $(U_{\text{H}}(\phi), \mathcal{I}', \mathcal{V})$ di ϕ corrispondente a quell'ambiente è definita ponendo:
  - $\mathcal{I}'(a) = a$ per ogni $a \in Cos$ in $U_{\text{H}}(\phi)$ come da definizione di interpretazione di Herbrand.
  - $\mathcal{I}'(f) = f$ per ogni $f \in Fun$ presente in ϕ come da definizione di interpretazione di Herbrand.
  - $\mathcal{I}'(P) = \{(u_1, \ldots, u_n) \in (U_{\text{H}}(\phi))^n \mid (D, \mathcal{I}, \mathcal{V}) \models P(u_1, \ldots, u_n)\}$ per ogni $P \in Pred$ presente in ϕ.

Osservato che per ogni sottoformula immediata ϕ' di ϕ risulta $\phi' \in FBF_{\text{pred,crs}}$ se ϕ non è quantificata, dimostriamo per induzione sulla struttura sintattica di ϕ che se $(D,\mathcal{I},\mathcal{V}) \models \phi$ allora $(U_{\text{H}}(\phi),\mathcal{I}',\mathcal{V}) \models \phi$:

- Sia ϕ della forma $P(t_1,\ldots,t_n)$ con $P \in Pred$ e $t_1,\ldots,t_n \in Ter$. Poichè ϕ è chiusa e quindi $t_1,\ldots,t_n$ sono termini ground appartenenti a $U_{\text{H}}(\phi)$, se $(D,\mathcal{I},\mathcal{V}) \models \phi$ allora $(t_1,\ldots,t_n) \in \mathcal{I}'(P)$ per definizione di $\mathcal{I}'(P)$, cioè $(U_{\text{H}}(\phi),\mathcal{I}',\mathcal{V}) \models \phi$.
- Sia ϕ della forma $(\neg\phi')$ e supponiamo che se $(D,\mathcal{I},\mathcal{V}) \models \phi'$ allora $(U_{\text{H}}(\phi'),\mathcal{I}',\mathcal{V}) \models \phi'$. Se $(D,\mathcal{I},\mathcal{V}) \models \phi$, cioè $(D,\mathcal{I},\mathcal{V}) \not\models \phi'$, allora $(U_{\text{H}}(\phi'),\mathcal{I}',\mathcal{V}) \not\models \phi'$ sfruttando l'ipotesi induttiva, cioè $(U_{\text{H}}(\phi),\mathcal{I}',\mathcal{V}) \models \phi$.
- Sia ϕ della forma $(\phi'_1 \vee \phi'_2)$ e supponiamo che se $(D,\mathcal{I},\mathcal{V}) \models \phi'_i$ allora $(U_{\text{H}}(\phi'_i),\mathcal{I}',\mathcal{V}) \models \phi'_i$ per $i \in \{1,2\}$. Se $(D,\mathcal{I},\mathcal{V}) \models \phi$, cioè $(D,\mathcal{I},\mathcal{V}) \models \phi'_1$ o $(D,\mathcal{I},\mathcal{V}) \models \phi'_2$, allora $(U_{\text{H}}(\phi'_1),\mathcal{I}',\mathcal{V}) \models \phi'_1$ o $(U_{\text{H}}(\phi'_2),\mathcal{I}',\mathcal{V}) \models \phi'_2$ sfruttando l'ipotesi induttiva, cioè $(U_{\text{H}}(\phi),\mathcal{I}',\mathcal{V}) \models \phi$.
- Sia ϕ della forma $(\phi'_1 \wedge \phi'_2)$ e supponiamo che se $(D,\mathcal{I},\mathcal{V}) \models \phi'_i$ allora $(U_{\text{H}}(\phi'_i),\mathcal{I}',\mathcal{V}) \models \phi'_i$ per $i \in \{1,2\}$. Se $(D,\mathcal{I},\mathcal{V}) \models \phi$, cioè $(D,\mathcal{I},\mathcal{V}) \models \phi'_1$ e $(D,\mathcal{I},\mathcal{V}) \models \phi'_2$, allora $(U_{\text{H}}(\phi'_1),\mathcal{I}',\mathcal{V}) \models \phi'_1$ e $(U_{\text{H}}(\phi'_2),\mathcal{I}',\mathcal{V}) \models \phi'_2$ sfruttando l'ipotesi induttiva, cioè $(U_{\text{H}}(\phi),\mathcal{I}',\mathcal{V}) \models \phi$.
- Sia ϕ della forma $(\phi'_1 \rightarrow \phi'_2)$ e supponiamo che se $(D,\mathcal{I},\mathcal{V}) \models \phi'_i$ allora $(U_{\text{H}}(\phi'_i),\mathcal{I}',\mathcal{V}) \models \phi'_i$ per $i \in \{1,2\}$. Se $(D,\mathcal{I},\mathcal{V}) \models \phi$, cioè $(D,\mathcal{I},\mathcal{V}) \not\models \phi'_1$ oppure $(D,\mathcal{I},\mathcal{V}) \models \phi'_2$, allora $(U_{\text{H}}(\phi'_1),\mathcal{I}',\mathcal{V}) \not\models \phi'_1$ oppure $(U_{\text{H}}(\phi'_2),\mathcal{I}',\mathcal{V}) \models \phi'_2$ sfruttando l'ipotesi induttiva, cioè $(U_{\text{H}}(\phi),\mathcal{I}',\mathcal{V}) \models \phi$.
- Sia ϕ della forma $(\phi'_1 \leftrightarrow \phi'_2)$ e supponiamo che se $(D,\mathcal{I},\mathcal{V}) \models \phi'_i$ allora $(U_{\text{H}}(\phi'_i),\mathcal{I}',\mathcal{V}) \models \phi'_i$ per $i \in \{1,2\}$. Se $(D,\mathcal{I},\mathcal{V}) \models \phi$, cioè $(D,\mathcal{I},\mathcal{V}) \models \phi'_1$ e $(D,\mathcal{I},\mathcal{V}) \models \phi'_2$ oppure $(D,\mathcal{I},\mathcal{V}) \not\models \phi'_1$ e $(D,\mathcal{I},\mathcal{V}) \not\models \phi'_2$, allora $(U_{\text{H}}(\phi'_1),\mathcal{I}',\mathcal{V}) \models \phi'_1$ e $(U_{\text{H}}(\phi'_2),\mathcal{I}',\mathcal{V}) \models \phi'_2$ oppure $(U_{\text{H}}(\phi'_1),\mathcal{I}',\mathcal{V}) \not\models \phi'_1$ e $(U_{\text{H}}(\phi'_2),\mathcal{I}',\mathcal{V}) \not\models \phi'_2$ sfruttando l'ipotesi induttiva, cioè $(U_{\text{H}}(\phi),\mathcal{I}',\mathcal{V}) \models \phi$.
- Sia ϕ della forma $(\forall x)(\phi')$ e, osservato che ϕ' potrebbe non essere chiusa, supponiamo che se $(D,\mathcal{I},\mathcal{V}) \models \phi'[{}^{u}/_{x}]$ allora $(U_{\text{H}}(\phi'[{}^{u}/_{x}]),\mathcal{I}',\mathcal{V}) \models \phi'[{}^{u}/_{x}]$ per ogni $u \in U_{\text{H}}(\phi)$, dove $\phi'[{}^{u}/_{x}] \in FBF_{\text{pred,crs}}$. Se $(D,\mathcal{I},\mathcal{V}) \models \phi$, cioè $(D,\mathcal{I},\mathcal{V}\setminus\{(x,\mathcal{V}(x))\}\cup\{(x,d)\}) \models \phi'$ per ogni $d \in D$, allora in particolare $(D,\mathcal{I},\mathcal{V}\setminus\{(x,\mathcal{V}(x))\}\cup\{(x,d)\}) \models \phi'$ per ogni $d \in D$ tale che $d = \mathcal{I}(u)$ per qualche $u \in U_{\text{H}}(\phi)$, cioè $(D,\mathcal{I},\mathcal{V}) \models \phi'[{}^{u}/_{x}]$ per ogni $u \in U_{\text{H}}(\phi)$, e quindi $(U_{\text{H}}(\phi'[{}^{u}/_{x}]),\mathcal{I}',\mathcal{V}) \models \phi'[{}^{u}/_{x}]$ per ogni $u \in U_{\text{H}}(\phi)$ sfruttando l'ipotesi induttiva, cioè $(U_{\text{H}}(\phi),\mathcal{I}',\mathcal{V}\setminus\{(x,\mathcal{V}(x))\}\cup\{(x,u)\}) \models \phi'$ per ogni $u \in U_{\text{H}}(\phi)$, cioè $(U_{\text{H}}(\phi),\mathcal{I}',\mathcal{V}) \models \phi$.

- Il precedente teorema non vale per formule che contengono quantificatori esistenziali. Ad esempio, la formula ϕ data da $P(a) \wedge \exists x\, \neg P(x)$ è soddisfacibile, come si vede prendendo un qualsiasi ambiente $(D,\mathcal{I},\mathcal{V})$ dove D ha almeno due valori d_1, d_2 tali che $\mathcal{I}(a) = d_1 \in \mathcal{I}(P)$ mentre $d_2 \notin \mathcal{I}(P)$. La formula ϕ non ha però alcun modello di Herbrand. Infatti $U_{\text{H}}(\phi) = \{a\}$ e $B_{\text{H}}(\phi) = \{P(a)\}$, quindi gli unici due possibili sottoinsiemi di $B_{\text{H}}(\phi)$ sono $\emptyset$ e $\{P(a)\}$. L'interpretazione di Herbrand corrispondente al primo sottoinsieme rende $P(a)$ falsa ed $\exists x\, \neg P(x)$ vera, mentre l'interpretazione di Herbrand corrispondente al secondo sottoinsieme rende $P(a)$ vera ed $\exists x\, \neg P(x)$ falsa. Dunque nessuna delle due possibili interpretazioni di Herbrand soddisfa ϕ.

- Corollario: Sia $\phi \in FBF_{\text{pred,crs}}$. Allora ϕ è soddisfacibile sse ha un modello di Herbrand.

- Dimostrazione: Se ϕ è soddisfacibile, cioè se esiste un ambiente $(D,\mathcal{I},\mathcal{V})$ che soddisfa ϕ, allora ϕ ha un modello di Herbrand che è l'interpretazione di Herbrand di ϕ ottenuta da $(D,\mathcal{I},\mathcal{V})$ tramite la costruzione mostrata nella dimostrazione del precedente teorema. Viceversa, se ϕ ha un modello di Herbrand, allora ϕ è banalmente soddisfacibile.

- Data una formula $\phi \in FBF_{\text{pred,crs}}$ della forma $\forall x_1 \ldots \forall x_n\, \phi'$ dove ϕ' è la matrice, l'espansione di Herbrand di ϕ è definita ponendo $E_{\text{H}}(\phi) = \{\phi'[{}^{u_1}/_{x_1},\ldots,{}^{u_n}/_{x_n}] \mid u_1,\ldots,u_n \in U_{\text{H}}(\phi)\}$. In altri termini, $E_{\text{H}}(\phi)$ viene ottenuta da ϕ rimuovendo tutti i quantificatori universali e prendendo tutte le istanze ground della matrice. Poiché non contengono quantificatori né variabili, le formule appartenenti a $E_{\text{H}}(\phi)$ sono assimilabili a formule della logica proposizionale (però potrebbero essercene infinite).

- Esempio: Se ϕ è la formula $\forall x\,(P(a) \vee P(f(x)))$, allora $E_{\text{H}}(\phi)$ comprende $P(a) \vee P(f(a))$, $P(a) \vee P(f(f(a)))$, $P(a) \vee P(f(f(f(a))))$ e così via, dove $P(a)$, $P(f(a))$, $P(f(f(a)))$, $P(f(f(f(a))))$, ecc. sono assimilabili a proposizioni.

- Teorema: Sia $\phi \in FBF_{\text{pred,crs}}$. Allora ϕ è soddisfacibile sse $E_{\text{H}}(\phi)$ lo è, cioè sse esiste un ambiente che soddisfa tutte le formule di $E_{\text{H}}(\phi)$.

- Dimostrazione: Sia ϕ della forma $\forall x_1 \dots \forall x_n \, \phi'$ dove ϕ' è la matrice. Allora ϕ è soddisfacibile sse ϕ ha un modello di Herbrand $(U_{\text{H}}(\phi), \mathcal{I}, \mathcal{V})$, cioè sse $(U_{\text{H}}(\phi), \mathcal{I}, \mathcal{V} \setminus \{(x_i, \mathcal{V}(x_i)) \mid 1 \leq i \leq n\} \cup \{(x_i, u_i) \mid 1 \leq i \leq n\}) \models \phi'$ per ogni $u_1, \dots, u_n \in U_{\text{H}}(\phi)$, cioè sse $(U_{\text{H}}(\phi), \mathcal{I}, \mathcal{V}) \models \phi'[{}^{u_1}/_{x_1}, \dots, {}^{u_n}/_{x_n}]$ per ogni $u_1, \dots, u_n \in U_{\text{H}}(\phi)$, cioè sse $(U_{\text{H}}(\phi), \mathcal{I}, \mathcal{V})$ soddisfa tutte le formule di $E_{\text{H}}(\phi)$ – ne è un modello di Herbrand perché $U_{\text{H}}(\phi'[{}^{u_1}/_{x_1}, \dots, {}^{u_n}/_{x_n}]) = U_{\text{H}}(\phi)$ per ogni $u_1, \dots, u_n \in U_{\text{H}}(\phi)$.

- Il problema della (in)soddisfacibilità di una formula ben formata della logica dei predicati è dunque riducibile al problema della (in)soddisfacibilità di un insieme di formule ben formate della logica proposizionale. L'aspetto critico è che questo insieme è infinito, tranne nel caso in cui la formula originaria non contiene simboli di funzione.

- Dato un insieme di formule $\Phi \in 2^{FBF_{\text{prop}}}$, diciamo che esso è:

  - Soddisfacibile sse esiste $A \in 2^{Prop}$ tale che $A \models \phi$ per ogni $\phi \in \Phi$.
  - Finitamente soddisfacibile sse ogni sottoinsieme finito di Φ è soddisfacibile.

- Teorema (di compattezza): Un insieme di formule è soddisfacibile sse è finitamente soddisfacibile.

- Dimostrazione: Sia $\Phi \in 2^{FBF_{\text{prop}}}$. Se Φ è soddisfacibile allora ovviamente ogni sottoinsieme finito di Φ è soddisfacibile e quindi Φ è finitamente soddisfacibile.
  Supponiamo ora che Φ sia finitamente soddisfacibile. Dopo aver enumerato come $p_0, p_1, \dots, p_k, \dots$ le proposizioni presenti nelle formule di Φ (il cui insieme denotiamo con $Prop'$) e come $\gamma_0, \gamma_1, \dots, \gamma_k, \dots$ le formule ben formate che possono essere costruite con quelle proposizioni (il cui insieme denotiamo con $FBF_{Prop'}$ e include Φ), definiamo l'insieme di formule $\Gamma = \bigcup_{n \in \mathbb{N}} \Gamma_n$ dove:
  $$\Gamma_n = \begin{cases} \Phi & \text{se } n = 0 \\ \Gamma_{n-1} \cup \{\gamma_{n-1}\} & \text{se } n > 0 \text{ e } \Gamma_{n-1} \cup \{\gamma_{n-1}\} \text{ è finitamente soddisfacibile} \\ \Gamma_{n-1} \cup \{\neg\gamma_{n-1}\} & \text{se } n > 0 \text{ e } \Gamma_{n-1} \cup \{\gamma_{n-1}\} \text{ non è finitamente soddisfacibile} \end{cases}$$
  Dimostriamo che ogni Γ_n è finitamente soddisfacibile procedendo per induzione su $n \in \mathbb{N}$:

  - Se $n = 0$ allora Γ_n coincide con Φ che è finitamente soddisfacibile per ipotesi iniziale.
  - Sia $n \geq 0$ e supponiamo che Γ_n sia finitamente soddisfacibile. Assumiamo che Γ_{n+1} non sia finitamente soddisfacibile, cioè che esistano $\Gamma'_n, \Gamma''_n \subseteq \Gamma_n$ tali che $\Gamma'_n \cup \{\gamma_n\}$ e $\Gamma''_n \cup \{\neg\gamma_n\}$ sono insoddisfacibili. Poiché $\Gamma'_n \cup \Gamma''_n \subseteq \Gamma_n$ e Γ_n è finitamente soddisfacibile per ipotesi induttiva, $\Gamma'_n \cup \Gamma''_n$ è soddisfacibile. Sia A un assegnamento di verità che soddisfa tutte le formule di $\Gamma'_n \cup \Gamma''_n$. Dal fatto che o $A \models \gamma_n$ o $A \models \neg\gamma_n$, segue che o A soddisfa tutte le formule di $\Gamma'_n \cup \{\gamma_n\}$ o A soddisfa tutte le formule di $\Gamma''_n \cup \{\neg\gamma_n\}$. Di conseguenza abbiamo raggiunto una contraddizione e quindi Γ_{n+1} è anch'esso finitamente soddisfacibile.

  L'insieme Γ è finitamente soddisfacibile. Infatti, sia $\{\gamma'_1, \dots, \gamma'_h\}$ un sottoinsieme finito di Γ. Per ogni $j = 1, \dots, h$ da $\gamma'_j \in \Gamma$ segue che $\gamma'_j \in \Gamma_{n_j}$ per qualche $n_j \in \mathbb{N}$. Pertanto $\{\gamma'_1, \dots, \gamma'_h\}$ è un sottoinsieme finito di Γ_n dove $n = \max_{1 \leq j \leq h} n_j$ e, poiché Γ_n è finitamente soddisfacibile, $\{\gamma'_1, \dots, \gamma'_h\}$ è soddisfacibile. L'insieme Γ è anche esaustivo, nel senso che o $\gamma_k \in \Gamma$ o $\neg\gamma_k \in \Gamma$ per ogni $k \in \mathbb{N}$. Infatti, se $\{\gamma_k, \neg\gamma_k\} \subseteq \Gamma$ per qualche $k \in \mathbb{N}$, allora Γ conterrebbe un sottoinsieme finito insoddisfacibile e quindi non potrebbe essere finitamente soddisfacibile.
  Dal fatto che Γ è finitamente soddisfacibile ed esaustivo segue che Γ è soddisfacibile. Infatti, osservato che o $p_k \in \Gamma$ o $\neg p_k \in \Gamma$ per ogni $k \in \mathbb{N}$ essendo Γ esaustivo, l'assegnamento di verità $A \in 2^{Prop'}$ definito ponendo $p_k \in A$ se $p_k \in \Gamma$ e $p_k \notin A$ se $\neg p_k \in \Gamma$ è tale che $A \models \gamma_k$ sse $\gamma_k \in \Gamma$ come si dimostra procedendo per induzione sulla struttura sintattica di $\gamma_k \in FBF_{Prop'}$:

  - Se $\gamma_k \in Prop'$ allora $A \models \gamma_k$ sse $\gamma_k \in \Gamma$ per definizione di A.
  - Sia γ_k della forma $(\neg\gamma')$ e supponiamo che $A \models \gamma'$ sse $\gamma' \in \Gamma$. Allora $A \models \gamma_k$ sse $A \not\models \gamma'$, cioè sse $\gamma' \notin \Gamma$ sfruttando l'ipotesi induttiva, cioè sse $\gamma_k \in \Gamma$ essendo Γ esaustivo.

- Sia γ_k della forma $(\gamma'_1 \vee \gamma'_2)$ e supponiamo che $A \models \gamma'_i$ sse $\gamma'_i \in \Gamma$ per $i \in \{1,2\}$. Allora $A \models \gamma_k$ sse $A \models \gamma'_1$ o $A \models \gamma'_2$, cioè sse $\gamma'_1 \in \Gamma$ o $\gamma'_2 \in \Gamma$ sfruttando l'ipotesi induttiva, cioè sse $\gamma_k \in \Gamma$. Infatti, se $\gamma_k \in \Gamma$ ma $\gamma'_1, \gamma'_2 \notin \Gamma$, allora $\neg\gamma'_1, \neg\gamma'_2 \in \Gamma$ essendo Γ esaustivo. Di conseguenza $\{(\gamma'_1 \vee \gamma'_2), \neg\gamma'_1, \neg\gamma'_2\}$ sarebbe un sottoinsieme finito insoddisfacibile di Γ, contraddicendo così il fatto che Γ è finitamente soddisfacibile. Analogamente, se $\gamma'_1 \in \Gamma$ (risp. $\gamma'_2 \in \Gamma$) ma $\gamma_k \notin \Gamma$, allora $\neg\gamma_k \in \Gamma$ essendo Γ esaustivo. Di conseguenza $\{(\neg\gamma'_1 \wedge \neg\gamma'_2), \gamma'_1\}$ (risp. $\{(\neg\gamma'_1 \wedge \neg\gamma'_2), \gamma'_2\}$) sarebbe un sottoinsieme finito insoddisfacibile di Γ, contraddicendo così il fatto che Γ è finitamente soddisfacibile.
- Sia γ_k della forma $(\gamma'_1 \wedge \gamma'_2)$ e supponiamo che $A \models \gamma'_i$ sse $\gamma'_i \in \Gamma$ per $i \in \{1,2\}$. Allora $A \models \gamma_k$ sse $A \models \gamma'_1$ e $A \models \gamma'_2$, cioè sse $\gamma'_1 \in \Gamma$ e $\gamma'_2 \in \Gamma$ sfruttando l'ipotesi induttiva, cioè sse $\gamma_k \in \Gamma$. Infatti, se $\gamma_k \in \Gamma$ ma $\gamma'_1 \notin \Gamma$ (risp. $\gamma'_2 \notin \Gamma$), allora $\neg\gamma'_1 \in \Gamma$ (risp. $\neg\gamma'_2 \in \Gamma$) essendo Γ esaustivo. Di conseguenza $\{(\gamma'_1 \wedge \gamma'_2), \neg\gamma'_1\}$ (risp. $\{(\gamma'_1 \wedge \gamma'_2), \neg\gamma'_2\}$) sarebbe un sottoinsieme finito insoddisfacibile di Γ, contraddicendo così il fatto che Γ è finitamente soddisfacibile. Analogamente, se $\gamma'_1 \in \Gamma$ e $\gamma'_2 \in \Gamma$ ma $\gamma_k \notin \Gamma$, allora $\neg\gamma_k \in \Gamma$ essendo Γ esaustivo. Di conseguenza $\{(\neg\gamma'_1 \vee \neg\gamma'_2), \gamma'_1, \gamma'_2\}$ sarebbe un sottoinsieme finito insoddisfacibile di Γ, contraddicendo così il fatto che Γ è finitamente soddisfacibile.
- Sia γ_k della forma $(\gamma'_1 \rightarrow \gamma'_2)$ e supponiamo che $A \models \gamma'_i$ sse $\gamma'_i \in \Gamma$ per $i \in \{1,2\}$. Allora $A \models \gamma_k$ sse $A \not\models \gamma'_1$ oppure $A \models \gamma'_2$, cioè sse $\gamma'_1 \notin \Gamma$ oppure $\gamma'_2 \in \Gamma$ sfruttando l'ipotesi induttiva, cioè sse $\gamma_k \in \Gamma$. Infatti, se $\gamma_k \in \Gamma$ ma $\gamma'_1 \in \Gamma$ e $\gamma'_2 \notin \Gamma$, allora $\neg\gamma'_2 \in \Gamma$ essendo Γ esaustivo. Di conseguenza $\{(\gamma'_1 \rightarrow \gamma'_2), \gamma'_1, \neg\gamma'_2\}$ sarebbe un sottoinsieme finito insoddisfacibile di Γ, contraddicendo così il fatto che Γ è finitamente soddisfacibile. Analogamente, se $\gamma'_1 \notin \Gamma$ (risp. $\gamma'_2 \in \Gamma$) ma $\gamma_k \notin \Gamma$, allora $\neg\gamma'_1 \in \Gamma$ e $\neg\gamma_k \in \Gamma$ essendo Γ esaustivo. Di conseguenza $\{\neg(\gamma'_1 \rightarrow \gamma'_2), \neg\gamma'_1\}$ (risp. $\{\neg(\gamma'_1 \rightarrow \gamma'_2), \gamma'_2\}$) sarebbe un sottoinsieme finito insoddisfacibile di Γ, contraddicendo così il fatto che Γ è finitamente soddisfacibile.
- Sia γ_k della forma $(\gamma'_1 \leftrightarrow \gamma'_2)$ e supponiamo che $A \models \gamma'_i$ sse $\gamma'_i \in \Gamma$ per $i \in \{1,2\}$. Allora $A \models \gamma_k$ sse $A \models \gamma'_1$ e $A \models \gamma'_2$ oppure $A \not\models \gamma'_1$ e $A \not\models \gamma'_2$, cioè sse $\gamma'_1, \gamma'_2 \in \Gamma$ oppure $\gamma'_1, \gamma'_2 \notin \Gamma$ sfruttando l'ipotesi induttiva, cioè sse $\gamma_k \in \Gamma$. Infatti, se $\gamma_k \in \Gamma$ ma $\gamma'_1 \in \Gamma$ e $\gamma'_2 \notin \Gamma$, allora $\neg\gamma'_2 \in \Gamma$ essendo Γ esaustivo. Di conseguenza $\{(\gamma'_1 \leftrightarrow \gamma'_2), \gamma'_1, \neg\gamma'_2\}$ sarebbe un sottoinsieme finito insoddisfacibile di Γ, contraddicendo così il fatto che Γ è finitamente soddisfacibile. Analogamente, se $\gamma'_1, \gamma'_2 \in \Gamma$ (risp. $\gamma'_1, \gamma'_2 \notin \Gamma$) ma $\gamma_k \notin \Gamma$, allora ($\neg\gamma'_1, \neg\gamma'_2 \in \Gamma$ e) $\neg\gamma_k \in \Gamma$ essendo Γ esaustivo. Di conseguenza $\{\neg(\gamma'_1 \leftrightarrow \gamma'_2), \gamma'_1, \gamma'_2\}$ (risp. $\{\neg(\gamma'_1 \leftrightarrow \gamma'_2), \neg\gamma'_1, \neg\gamma'_2\}$) sarebbe un sottoinsieme finito insoddisfacibile di Γ, contraddicendo così il fatto che Γ è finitamente soddisfacibile.

Poiché $\Phi \subseteq \Gamma$ e Γ è soddisfacibile, anche Φ è soddisfacibile.

- Corollario: Un insieme di formule è insoddisfacibile sse ammette un sottoinsieme finito tale che la congiunzione delle sue formule è insoddisfacibile.
- Il seguente teorema, che è alla base della maggior parte degli algoritmi di dimostrazione automatica di teoremi, mostra che il problema originario è ulteriormente riducibile a trovare un opportuno sottoinsieme finito di quell'insieme di formule ben formate della logica proposizionale.
- Teorema (di Herbrand): Sia $\phi \in FBF_{\text{pred,crs}}$. Allora ϕ è insoddisfacibile sse esiste un sottoinsieme finito di $E_{\text{H}}(\phi)$ tale che la congiunzione delle sue formule è insoddisfacibile.
- Dimostrazione: Segue dai due teoremi precedenti.
- Sulla base del teorema di Herbrand possiamo costruire il seguente algoritmo che opera per refutazione al fine di stabilire la validità di una formula $\phi \in FBF_{\text{pred}}$:

  1. Trasformare $\neg\phi$ in una formula $\gamma \in FBF_{\text{pred,crs}}$ che è insoddisfacibile sse lo è $\neg\phi$ (vedi Sez. 7.1).
  2. Enumerare come $\gamma_0, \gamma_1, \gamma_2, \ldots$ le formule in $E_{\text{H}}(\gamma)$.
  3. Assegnare 0 a una variabile intera n.
  4. Finché $\bigwedge_{i=0}^{n} \gamma_i$ è soddisfacibile, incrementare n di 1.
  5. Segnalare che γ è insoddisfacibile, cioè che ϕ è valida.

- Poichè il problema affrontato è soltanto semi-decidibile (vedi Sez. 6.2), il precedente algoritmo termina sicuramente solo se ϕ è valida, altrimenti va avanti considerando approssimazioni di $E_{\mathrm{H}}(\gamma)$ che sono costituite da insiemi sempre più grandi di formule senza mai terminare. Poiché le formule appartenenti a $E_{\mathrm{H}}(\gamma)$ sono assimilabili a formule della logica proposizionale, la soddisfacibilità di ogni $\bigwedge_{i=0}^{n} \gamma_i$ può essere stabilita costruendone la tabella di verità.

- Esempio: Se ϕ è la formula $\exists x \exists y\,(\neg(P(a) \vee P(f(x))) \vee P(y))$, allora la formula γ usata dall'algoritmo è $\forall x \forall y\,((P(a) \vee P(f(x))) \wedge \neg P(y))$. Notato che $U_{\mathrm{H}}(\gamma)$ comprende a, $f(a)$, $f(f(a))$, $f(f(f(a)))$ e così via, abbiamo che $E_{\mathrm{H}}(\gamma)$ comprende $(P(a) \vee P(f(a))) \wedge \neg P(a)$, $(P(a) \vee P(f(a))) \wedge \neg P(f(a))$ e così via. Posto $p = P(a)$ e $q = P(f(a))$, l'algoritmo esamina come prima formula $\gamma_0 = (p \vee q) \wedge \neg p$ e determina che essa è soddisfacibile come si vede prendendo l'assegnamento di verità $A = \{q\}$. Poi l'algoritmo esamina la congiunzione di $\gamma_0 = (p \vee q) \wedge \neg p$ e $\gamma_1 = (p \vee q) \wedge \neg q$, cioè $(p \vee q) \wedge \neg p \wedge (p \vee q) \wedge \neg q \equiv (p \vee q) \wedge (p \vee q) \wedge (\neg p \wedge \neg q) \equiv (p \vee q) \wedge (\neg p \wedge \neg q) \equiv (p \vee q) \wedge \neg(p \vee q)$, che è insoddisfacibile. Pertanto l'algoritmo termina dopo due iterazioni segnalando che ϕ è valida. ■fltplf_16

7.4 Risoluzione di Robinson e Algoritmo di Refutazione

- L'algoritmo di refutazione basato sulla teoria di Herbrand deve calcolare a ogni iterazione il valore di verità della congiunzione di un numero sempre crescente di formule ben formate della logica proposizionale, impiegando così a ogni passo un tempo esponenziale rispetto al numero di proposizioni presenti in quelle formule. Un miglioramento delle prestazioni dell'algoritmo di refutazione può essere ottenuto se, invece di costruire a ogni iterazione la tabella di verità della congiunzione delle formule dell'espansione di Herbrand generate fino a quel momento, si trasforma inizialmente la matrice della formula considerata in forma normale congiuntiva e poi si applica il metodo di risoluzione.

- Il metodo di risoluzione venne introdotto nel 1965 da Alan Robinson allo scopo di definire un sistema deduttivo che fosse più efficiente dei sistemi deduttivi alla Hilbert per dimostrare automaticamente teoremi. Questo metodo è particolarmente semplice da impiegare perché comprende soltanto una regola di inferenza. Esso procede per refutazione e si applica solo a insiemi di clausole.

- La regola di inferenza del metodo di risoluzione per la logica proposizionale è la seguente:

$$\frac{\lambda_{1,1} \vee \cdots \vee \lambda_{1,m} \vee p \qquad \lambda_{2,1} \vee \cdots \vee \lambda_{2,n} \vee \neg p}{\lambda_{1,1} \vee \cdots \vee \lambda_{1,m} \vee \lambda_{2,1} \vee \cdots \vee \lambda_{2,n}}$$

  dove $m, n \in \mathbb{N}$, ogni $\lambda_{i,j}$ è un letterale e la conclusione della regola è detta clausola risolvente (la clausola risolvente è una clausola di Horn qualora le premesse siano due clausole di Horn). Nel caso $m = n = 0$ la clausola risolvente è la clausola vuota, la quale viene denotata con $\square$.

- Ogni formula $\phi \in FBF_{\mathrm{prop}}$ può essere trasformata in forma normale congiuntiva e quindi può essere vista come un insieme di clausole in cui ogni clausola può essere vista come un insieme di letterali. La notazione insiemistica è pienamente adeguata per rappresentare forme normali congiuntive e clausole grazie alle proprietà commutative, associative e di idempotenza di congiunzione e disgiunzione.

- Esempio: La forma norma congiuntiva $(p \vee q \vee \neg r) \wedge (p \vee \neg q \vee r)$ può essere rappresentata come $\{\{p, q, \neg r\}, \{p, \neg q, r\}\}$, dove le due clausole hanno come risolventi $\{p, \neg r, r\}$ e $\{p, q, \neg q\}$.

- Poiché una formula in forma normale congiuntiva è soddisfacibile sse tutte le sue clausole sono soddisfacibili, l'insieme vuoto di clausole ($\emptyset$) è banalmente soddisfacibile. Poiché una clausola è soddisfacibile sse almeno uno dei suoi letterali è soddisfacibile, la clausola vuota ($\square$) è banalmente insoddisfacibile e quindi rappresenta una contraddizione. Ogni clausola che contiene sia una proposizione che la sua negazione è banalmente valida e quindi rappresenta una tautologia.

- Dato un insieme di clausole C e una clausola c, scriviamo $C \vdash_{\mathrm{R}} c$ per indicare che c è derivabile da C tramite risoluzione (cioè che esiste una sequenza di clausole non vuota e di lunghezza finita tale che ogni clausola della sequenza o appartiene a C o è la risolvente di due clausole che la precedono nella sequenza, dove l'ultima clausola della sequenza è c – vedi Sez. 5.5). Diciamo che C è refutabile sse $C \vdash_{\mathrm{R}} \square$.

- La regola di inferenza del metodo di risoluzione preserva i modelli delle clausole originarie e il sistema deduttivo basato su tale regola risulta essere corretto e completo rispetto all'insoddisfacibilità.
- Teorema: Siano c_1, c_2, c delle clausole. Se c è una risolvente di c_1 e c_2, allora $c_1 \models c$ e $c_2 \models c$.
- Dimostrazione: Poiché c_1 e c_2 ammettono una clausola risolvente, esse sono entrambe non vuote e contengono almeno una proposizione p e la sua negazione $\neg p$, rispettivamente. Dunque $c = (c_1 \setminus \{p\}) \cup (c_2 \setminus \{\neg p\})$. Per ogni assegnamento di verità $A \in 2^{Prop}$ tale che $A \models c_1$ e $A \models c_2$, ci sono due casi:
  - Se $p \in A$ allora da $A \models c_2$ e $A \not\models \neg p$ segue che $A \models (c_2 \setminus \{\neg p\})$ e quindi $A \models c$.
  - Se $p \notin A$ allora da $A \models c_1$ e $A \not\models p$ segue che $A \models (c_1 \setminus \{p\})$ e quindi $A \models c$.
- Teorema (di risoluzione): Un insieme di clausole è insoddisfacibile sse è refutabile.
- Dimostrazione: In virtù del corollario al teorema di compattezza (vedi Sez. 7.3), un insieme di clausole è insoddisfacibile sse ammette un sottoinsieme finito insoddisfacibile, quindi possiamo ragionare su un insieme finito di clausole C.
  Se C è refutabile allora C è insoddisfacibile. Infatti, se C fosse soddisfacibile allora esisterebbe un assegnamento di verità $A \in 2^{Prop}$ che soddisfa tutte le clausole di C. Poiché C è refutabile e il metodo di risoluzione preserva i modelli delle clausole originarie, A sarebbe un modello anche di $\square$, ma ciò è assurdo perchè $\square$ non ammette modelli.
  Se C è insoddisfacibile allora C è refutabile come si dimostra procedendo per induzione sul numero $n \in \mathbb{N}$ di letterali presenti nelle clausole di C:
  - Se $n = 0$ allora $C = \{\square\}$ e quindi C è banalmente refutabile (il caso $C = \emptyset$ non è compatibile con l'ipotesi che C sia insoddisfacibile).
  - Sia $n \geq 1$ e supponiamo che ogni insieme finito di clausole con un numero di letterali minore di n sia refutabile qualora sia insoddisfacibile. Assumendo $\square \notin C$ per evitare casi banali, dal fatto che C è insoddisfacibile ed $n > 0$ segue che C deve contenere almeno due clausole e ci deve essere almeno una proposizione p che compare in un letterale positivo di una clausola $c_1 \in C$ e in un letterale negativo di un'altra clausola $c_2 \in C$.
    Indichiamo con $R_p(C)$ l'insieme di clausole ottenuto da C eliminando c_1 e c_2 e aggiungendo la risolvente di c_1 e c_2 rispetto a p, cioè $c_1' \cup c_2'$ dove $c_1' = c_1 \setminus \{p\}$ e $c_2' = c_2 \setminus \{\neg p\}$. L'insieme $R_p(C)$ è insoddisfacibile. Infatti, se esistesse un assegnamento di verità $A \in 2^{Prop}$ che soddisfa tutte le clausole di $R_p(C)$, allora da A potremmo ricavare due assegnamenti di verità $A', A'' \in 2^{Prop}$ identici ad A su $Prop \setminus \{p\}$ ponendo $p \notin A'$ e $p \in A''$. Poiché C è insoddisfacibile, A' e A'' non possono soddisfare tutte le clausole di C. In particolare, esisteranno $c' \in C$ tale che $A' \not\models c'$ e $c'' \in C$ tale che $A'' \not\models c''$. Poiché A' e A'' sono ottenuti da A che soddisfa tutte le clausole di $R_p(C)$, le due clausole c' e c'' non possono appartenere a $R_p(C)$ e quindi deve essere $c' = c_1$ e $c'' = c_2$. Poiché $c_1' \cup c_2' \in R_p(C)$, deve inoltre valere che A soddisfa $c_1' \cup c_2'$. Tuttavia, se A soddisfa c_1' allora A' soddisfa c_1, cioè c'. Analogamente, se A soddisfa c_2' allora A'' soddisfa c_2, cioè c''. Poiché abbiamo raggiunto una contraddizione, l'insieme $R_p(C)$ non può essere soddisfacibile.
    Dal fatto che $R_p(C)$ è insoddisfacibile e contiene $n - 2$ letterali, sfruttando l'ipotesi induttiva segue che $R_p(C)$ è refutabile e quindi tale è C in quanto $R_p(C)$ è ottenuto da C per risoluzione.
- Dato un insieme di clausole C, il metodo di risoluzione determina:
  - l'insieme $ris(C) = C \cup \{c \mid c \text{ risolvente di } c_1, c_2 \in C\}$ quando viene applicato una sola volta;
  - l'insieme $ris^*(C) = \bigcup_{n \in \mathbb{N}} ris^n(C)$ quando viene applicato un numero arbitrario di volte, dove:
$$ris^n(C) = \begin{cases} C & \text{se } n = 0 \\ ris(ris^{n-1}(C)) & \text{se } n > 0 \end{cases}$$

  Poiché $ris^*(C) \equiv C$, abbiamo che C è refutabile, cioè insoddisfacibile, sse $\square \in ris^*(C)$.

- Sulla base del metodo di risoluzione possiamo costruire il seguente algoritmo che opera per refutazione al fine di stabilire la validità di una formula $\phi \in FBF_{\text{prop}}$ (alternativo alla costruzione della tabella di verità di ϕ):

  1. Trasformare $\neg\phi$ in una formula $\gamma \in FBF_{\text{prop}}$ in forma normale congiuntiva (vedi Sez. 7.1).
  2. Trasformare γ in un insieme di clausole C dove le clausole sono insiemi di letterali.
  3. Ripetere i seguenti passi:
     (a) assegnare C a C';
     (b) assegnare $ris(C)$ a C;
     finché non vale che $\square \in C$ oppure $C = C'$.
  4. Se $\square \in C$ segnalare che ϕ è valida, altrimenti segnalare che ϕ non è valida.

- Esempio: La formula $((p \to (q \to r)) \to ((p \to q) \to (p \to r)))$ è valida perché:
  - La formula è equivalente a $(\neg(\neg p \vee (\neg q \vee r)) \vee (\neg(\neg p \vee q) \vee (\neg p \vee r)))$.
  - La sua negazione è equivalente a $((\neg p \vee (\neg q \vee r)) \wedge ((\neg p \vee q) \wedge (p \wedge \neg r)))$.
  - La corrispondente forma normale congiuntiva è $(\neg p \vee \neg q \vee r) \wedge (\neg p \vee q) \wedge (p) \wedge (\neg r)$.
  - Il corrispondente insieme di clausole è $\{\{\neg p, \neg q, r\}, \{\neg p, q\}, \{p\}, \{\neg r\}\}$.
  - L'applicazione della risoluzione alle clausole $\{\neg p, q\}$ e $\{p\}$ dà luogo a $\{q\}$.
  - L'applicazione della risoluzione alle clausole $\{\neg p, \neg q, r\}$ e $\{p\}$ dà luogo a $\{\neg q, r\}$.
  - L'applicazione della risoluzione alle clausole $\{\neg q, r\}$ e $\{q\}$ dà luogo a $\{r\}$.
  - L'applicazione della risoluzione alle clausole $\{r\}$ e $\{\neg r\}$ dà luogo a $\square$.

- Il metodo di risoluzione può essere esteso alla logica dei predicati facendo uso di sostituzioni e unificazioni. In effetti, la regola di inferenza del metodo di risoluzione per la logica proposizionale è un caso particolare della seguente regola:

$$\frac{\{\pi_{1,1}, \ldots, \pi_{1,h}\} \subseteq c_1\theta_1 \qquad \{\pi_{2,1}, \ldots, \pi_{2,k}\} \subseteq c_2\theta_2 \qquad \{\pi_{1,1}, \ldots, \pi_{1,h}, \neg\pi_{2,1}, \ldots, \neg\pi_{2,k}\} \text{ unificabile}}{((c_1\theta_1 \setminus \{\pi_{1,1}, \ldots, \pi_{1,h}\}) \cup (c_2\theta_2 \setminus \{\pi_{2,1}, \ldots, \pi_{2,k}\}))\,\theta}$$

  dove:

  - c_1, c_2 sono due clausole della logica dei predicati. Esse fanno parte della forma normale congiuntiva $\bigwedge_{i=1}^{n} (\bigvee_{j=1}^{m_i} \pi'_{i,j})$ della matrice di una formula $\gamma \in FBF_{\text{pred,crs}}$ che è insoddisfacibile sse lo è $\neg\phi$, dove $\phi \in FBF_{\text{pred}}$ è la formula di partenza.
  - θ_1, θ_2 sono due sostituzioni tali che $c_1\theta_1, c_2\theta_2$ non hanno variabili in comune. Esse consentono di unificare (a meno della negazione iniziale) letterali come $P(x)$ e $\neg P(f(x))$ che sono congiuntamente insoddisfacibili ma non possono essere unificati per via dell'occur check.
  - $h, k \in \mathbb{N}_{\geq 1}$. Diversamente dal caso della logica proposizionale, in cui si risolvono due soli letterali alla volta, h e k non sono necessariamente entrambi uguali a uno. Se si imponesse $h = k = 1$, l'insieme di clausole $\{\{P(x), P(y)\}, \{\neg P(x), \neg P(y)\}\}$, che è insoddisfacibile perché x e y sono quantificate universalmente e quindi il caso $x = y$ non può essere escluso, non sarebbe refutabile col metodo di risoluzione in quanto come ulteriori clausole si otterrebbero soltanto $\{P(y), \neg P(y)\}$ e $\{P(x), \neg P(x)\}$ senza mai arrivare alla generazione di $\square$.
  - θ è l'upg di $\{\pi_{1,1}, \ldots, \pi_{1,h}, \neg\pi_{2,1}, \ldots, \neg\pi_{2,k}\}$. I $\pi_{1,i}$ sono tutti positivi/negativi, i $\pi_{2,j}$ sono tutti negativi/positivi e la negazione va introdotta per consentire l'unificazione di letterali opposti.

- In logica proposizionale la regola si riduce a $\dfrac{\{p\} \subseteq c_1 \qquad \{\neg p\} \subseteq c_2}{(c_1 \setminus \{p\}) \cup (c_2 \setminus \{\neg p\})}$.

- Esempio: Le clausole $c_1 = \{P(z), \neg Q(y), P(f(x))\}$ e $c_2 = \{\neg P(x), \neg R(x)\}$ possono essere trasformate rispettivamente in $\{P(z), \neg Q(y), P(f(x))\}$ e $\{\neg P(x'), \neg R(x')\}$ mediante le sostituzioni $\theta_1 = \varepsilon$ e $\theta_2 = [^{x'}/_{x}]$ così da evitare variabili condivise. Le due clausole risultanti hanno come risolvente $\{\neg Q(y), \neg R(f(x))\}$ essendo $\theta = [^{f(x)}/_{z}, ^{f(x)}/_{x'}]$ l'upg di $\{P(z), P(f(x)), P(x')\}$.

- Anche nella logica dei predicati la regola di inferenza del metodo di risoluzione preserva i modelli delle clausole originarie e il sistema deduttivo basato su tale regola risulta essere corretto e completo rispetto all'insoddisfacibilità. Questo può essere dimostrato stabilendo dapprima un collegamento tra la regola di risoluzione proposizionale e la regola di risoluzione predicativa.

- Teorema (lifting lemma): Siano c_1, c_2 due clausole della logica dei predicati e siano c_1', c_2' due loro istanze ground, rispettivamente. Per ogni risolvente c' di c_1', c_2' (secondo la regola di risoluzione proposizionale) esiste una risolvente c di c_1, c_2 (secondo la regola di risoluzione predicativa) tale che c' è un'istanza ground di c.

- Dimostrazione: Date due clausole della logica dei predicati c_1, c_2, siano θ_1, θ_2 due sostituzioni tali che $c_1\theta_1, c_2\theta_2$ non hanno variabili in comune. Date due istanze ground c_1', c_2' di c_1, c_2, rispettivamente, risulta che c_1', c_2' sono anche istanze ground di $c_1\theta_1, c_2\theta_2$, rispettivamente, e quindi esistono due sostituzioni θ_1', θ_2' tali che $c_1' = (c_1\theta_1)\theta_1'$ e $c_2' = (c_2\theta_2)\theta_2'$. Sia $\theta' = \theta_1' \circ \theta_2'$. Poiché θ_1', θ_2' rimpiazzano variabili distinte, risulta $c_1' = (c_1\theta_1)\theta'$ e $c_2' = (c_2\theta_2)\theta'$.
  Al fine di evitare casi banali, supponiamo che la regola di risoluzione proposizionale sia applicabile a c_1', c_2'. Sia dunque c' una risolvente di c_1', c_2', diciamo $c' = (c_1' \setminus \{p\}) \cup (c_2' \setminus \{\neg p\})$ dove $p \in c_1'$ e $\neg p \in c_2'$. Poiché i letterali p e $\neg p$ derivano dall'applicazione di θ' ai letterali di $c_1\theta_1, c_2\theta_2$, rispettivamente, esisteranno $h \in \mathbb{N}_{\geq 1}$ letterali $\pi_{1,1}, \ldots, \pi_{1,h}$ in $c_1\theta_1$ tali che $p = \pi_{1,1}\theta' = \cdots = \pi_{1,h}\theta'$ e $k \in \mathbb{N}_{\geq 1}$ letterali $\pi_{2,1}, \ldots, \pi_{2,k}$ in $c_2\theta_2$ tali che $\neg p = \pi_{2,1}\theta' = \cdots = \pi_{2,k}\theta'$. Pertanto θ' è un unificatore di $\{\pi_{1,1}, \ldots, \pi_{1,h}, \neg\pi_{2,1}, \ldots, \neg\pi_{2,k}\}$ e quindi la regola di risoluzione predicativa è applicabile a $c_1\theta_1, c_2\theta_2$. Poichè $\{\pi_{1,1}, \ldots, \pi_{1,h}, \neg\pi_{2,1}, \ldots, \neg\pi_{2,k}\}$ è unificabile, in virtù del teorema di Robinson questo insieme ha l'upg che denotiamo con θ e quindi $\theta' = \theta \circ \theta''$. Di conseguenza c' è un'istanza ground della risolvente $c = ((c_1\theta_1 \setminus \{\pi_{1,1}, \ldots, \pi_{1,h}\}) \cup (c_2\theta_2 \setminus \{\pi_{2,1}, \ldots, \pi_{2,k}\}))\,\theta$ di c_1, c_2 perché $c' = (c_1' \setminus \{p\}) \cup (c_2' \setminus \{\neg p\}) = ((c_1\theta_1)\theta' \setminus \{p\}) \cup ((c_2\theta_2)\theta' \setminus \{\neg p\}) = ((c_1\theta_1 \setminus \{\pi_{1,1}, \ldots, \pi_{1,h}\}) \cup (c_2\theta_2 \setminus \{\pi_{2,1}, \ldots, \pi_{2,k}\}))\,\theta' = ((c_1\theta_1 \setminus \{\pi_{1,1}, \ldots, \pi_{1,h}\}) \cup (c_2\theta_2 \setminus \{\pi_{2,1}, \ldots, \pi_{2,k}\}))\,(\theta \circ \theta'') = c\theta''$.

- Teorema: Siano c_1, c_2, c delle clausole della logica dei predicati. Se c è una risolvente di c_1 e c_2, allora $c_1 \models c$ e $c_2 \models c$.

- Dimostrazione: Poiché c_1 e c_2 ammettono una clausola risolvente, date due sostituzioni θ_1, θ_2 tali che $c_1\theta_1, c_2\theta_2$ non hanno variabili in comune risulta che esistono $h \in \mathbb{N}_{\geq 1}$ letterali $\pi_{1,1}, \ldots, \pi_{1,h}$ in $c_1\theta_1$ e $k \in \mathbb{N}_{\geq 1}$ letterali $\pi_{2,1}, \ldots, \pi_{2,k}$ in $c_2\theta_2$ tali che $\{\pi_{1,1}, \ldots, \pi_{1,h}, \neg\pi_{2,1}, \ldots, \neg\pi_{2,k}\}$ è unificabile. Indicato con θ l'upg di questo insieme, siano $c = ((c_1\theta_1 \setminus \{\pi_{1,1}, \ldots, \pi_{1,h}\}) \cup (c_2\theta_2 \setminus \{\pi_{2,1}, \ldots, \pi_{2,k}\}))\,\theta$ e $\pi = \pi_{1,1}\theta = \cdots = \pi_{1,h}\theta = \neg\pi_{2,1}\theta = \cdots = \neg\pi_{2,k}\theta$. Per ogni ambiente $\mathcal{E} \in Amb$ tale che $\mathcal{E} \models c_1$ ed $\mathcal{E} \models c_2$, ci sono due casi:

  - Se $\mathcal{E} \models \pi$ allora da $\mathcal{E} \models (c_2\theta_2)\theta$ e $\mathcal{E} \not\models \neg\pi$ segue che $\mathcal{E} \models (c_2\theta_2)\theta \setminus \{\pi_{2,1}, \ldots, \pi_{2,k}\}\theta$ e quindi $\mathcal{E} \models c$.
  - Se $\mathcal{E} \not\models \pi$ allora da $\mathcal{E} \models (c_1\theta_1)\theta$ segue che $\mathcal{E} \models (c_1\theta_1)\theta \setminus \{\pi_{1,1}, \ldots, \pi_{1,k}\}\theta$ e quindi $\mathcal{E} \models c$.

- Dato un insieme di formule $\Phi \in 2^{FBF_{\text{pred}}}$, diciamo che esso è:

  - Soddisfacibile sse esiste $\mathcal{E} \in Amb$ tale che $\mathcal{E} \models \phi$ per ogni $\phi \in \Phi$.
  - Finitamente soddisfacibile sse ogni sottoinsieme finito di Φ è soddisfacibile.

- Teorema (di compattezza): Un insieme di formule della logica dei predicati è soddisfacibile sse è finitamente soddisfacibile.

- Corollario: Un insieme di formule della logica dei predicati è insoddisfacibile sse ammette un sottoinsieme finito tale che la congiunzione delle sue formule è insoddisfacibile.

- Teorema (di risoluzione): Un insieme di clausole della logica dei predicati è insoddisfacibile sse è refutabile.

- Dimostrazione: In virtù del corollario al teorema di compattezza, un insieme di clausole è insoddisfacibile sse ammette un sottoinsieme finito insoddisfacibile, quindi possiamo ragionare su un insieme finito di clausole C.
  Se C è refutabile allora C è insoddisfacibile. Infatti, se C fosse soddisfacibile allora esisterebbe un ambiente $\mathcal{E} \in Amb$ che soddisfa tutte le clausole di C. Poiché C è refutabile e il metodo di risoluzione preserva i modelli delle clausole originarie, $\mathcal{E}$ sarebbe un modello anche di $\square$, ma ciò è assurdo perchè $\square$ non ammette modelli.
  Se C è insoddisfacibile allora in virtù del teorema di Herbrand esiste un insieme C' di istanze ground delle clausole di C che è insoddisfacibile. Dal teorema di risoluzione per la logica proposizionale segue che C' è refutabile. Dal lifting lemma segue che anche C è refutabile.

- Dato un insieme di clausole C della logica dei predicati, come in logica proposizionale il metodo di risoluzione determina:

  - l'insieme $ris(C) = C \cup \{c \mid c \text{ risolvente di } c_1, c_2 \in C\}$ quando viene applicato una sola volta;
  - l'insieme $ris^*(C) = \bigcup_{n\in\mathbb{N}} ris^n(C)$ quando viene applicato un numero arbitrario di volte, dove:

$$ris^n(C) = \begin{cases} C & \text{se } n = 0 \\ ris(ris^{n-1}(C)) & \text{se } n > 0 \end{cases}$$

  Poiché $ris^*(C) \equiv C$, abbiamo che C è refutabile, cioè insoddisfacibile, sse $\square \in ris^*(C)$.

- Sulla base del metodo di risoluzione possiamo costruire il seguente algoritmo che opera per refutazione al fine di stabilire la validità di una formula $\phi \in FBF_{\text{pred}}$ (alternativo all'algoritmo di refutazione basato sulla teoria di Herbrand):

  1. Trasformare $\neg\phi$ in una formula $\gamma \in FBF_{\text{pred,crs}}$ che è insoddisfacibile sse lo è $\neg\phi$ (vedi Sez. 7.1).
  2. Trasformare la matrice γ' di γ in una formula γ'' in forma normale congiuntiva (vedi Sez. 7.1).
  3. Trasformare γ'' in un insieme di clausole C dove le clausole sono insiemi di letterali.
  4. Ripetere i seguenti passi:
     (a) assegnare C a C';
     (b) assegnare $ris(C)$ a C;

     finché non vale che $\square \in C$ oppure $C = C'$.
  5. Se $\square \in C$ segnalare che ϕ è valida, altrimenti segnalare che ϕ non è valida.

- Diversamente dal caso della logica proposizionale, l'algoritmo mostrato sopra potrebbe non terminare in quanto il problema della validità è soltanto semi-decidibile nella logica dei predicati. Diversamente dall'algoritmo di refutazione basato sulla teoria di Herbrand, questo algoritmo di refutazione basato sul metodo di risoluzione e sul calcolo dell'upg evita di generare tutte le istanze ground della matrice della formula ottenuta da quella di partenza e quindi ha delle prestazioni migliori.

- Esempi:

  - L'insieme soddisfacibile di clausole $C = \{\{P(0)\}, \{\neg P(x), P(succ(x))\}\}$ derivante dalla formulazione in logica dei predicati del principio di induzione determina un insieme $ris^*(C)$ che è infinito in quanto contiene elementi come $\{P(0)\}, \{P(succ(0))\}, \{P(succ(succ(0)))\}, \ldots$ – dove a parte $\{P(0)\}$ ciascuno di essi è ottenuto applicando la risoluzione al precedente e $\{\neg P(x), P(succ(x))\}$ – che sono in corrispondenza biunivoca coi numeri naturali.

– Riconsideriamo il paradosso del barbiere e utilizziamo l'algoritmo di refutazione basato sul metodo di risoluzione e sul calcolo dell'upg per dimostrare che la formula $\forall x\,(Rade(b,x) \leftrightarrow (\neg Rade(x,x)))$ è insoddisfacibile:

  * Poiché siamo interessati all'insoddisfacibilità della formula, non dobbiamo negarla.
  * La formula appartiene già a $FBF_{\text{pred,crs}}$.
  * La matrice della formula è $(Rade(b,x) \leftrightarrow (\neg Rade(x,x)))$.
  * La matrice è equivalente a $((Rade(b,x) \rightarrow (\neg Rade(x,x))) \wedge ((\neg Rade(x,x)) \rightarrow Rade(b,x)))$, cioè a $(((\neg Rade(b,x)) \vee (\neg Rade(x,x))) \wedge (Rade(x,x) \vee Rade(b,x)))$ che è in forma normale congiuntiva.
  * Il corrispondente insieme di clausole è $\{\{\neg Rade(b,x), \neg Rade(x,x)\}, \{Rade(x,x), Rade(b,x)\}\}$.
  * Poiché x è una variabile comune alle due clausole, applichiamo a esse le sostituzioni $[^{x_1}/_{x}]$ e $[^{x_2}/_{x}]$, rispettivamente, ottenendo pertanto $\{\{\neg Rade(b,x_1), \neg Rade(x_1,x_1)\}, \{Rade(x_2,x_2), Rade(b,x_2)\}\}$.
  * L'insieme di letterali $\{\neg Rade(b,x_1), \neg Rade(x_1,x_1), \neg Rade(x_2,x_2), \neg Rade(b,x_2)\}$ risulta essere unificabile.
  * Il suo upg è $[^{b}/_{x_1}, ^{b}/_{x_2}]$.
  * L'applicazione della risoluzione alle due clausole dà immediatamente luogo a $\square$. ■fltplf_17

Capitolo 8

Programmazione Logica: Il Linguaggio Prolog

8.1 Dalla Logica alla Programmazione Logica

- Come si è arrivati dalla logica alla programmazione logica?
  La logica può essere definita come la disciplina che studia la verità di un'affermazione deducendola (tramite regole) da altre affermazioni che sono state poste essere vere (postulati, assiomi, principi) o che sono già state dedotte essere vere (teoremi). Dal punto di vista concettuale, la logica si contrappone all'intuizione e risponde all'esigenza di trovare un modo di distinguere ciò che è vero da ciò che è falso. La storia della logica si articola in tre epoche: logica simbolica (dal 500 a.C. alla metà del 1800), logica algebrica (seconda metà del 1800) e logica matematica (dalla fine del 1800 in poi).

- Nell'antica Grecia, la logica era già ritenuta una materia fondamentale. Essa venne originariamente studiata dai sofisti nel VI secolo a.C. allo scopo di definire un sistema oggettivo di regole per determinare oltre ogni dubbio chi avesse vinto una disputa. Nel IV secolo a.C. Aristotele diede alla logica la forma rigorosamente deduttiva del sillogismo, cioè del ragionamento concatenato attraverso cui si giunge a conclusioni coerenti con le premesse da cui si parte (tutte le persone sono mortali, Socrate è una persona, quindi Socrate è mortale). In ogni sillogismo, sia le premesse che la conclusione sono espressioni formate da un soggetto universale o particolare e da un predicato verbale affermativo o negativo, dove la verità della conclusione dipende dalla verità delle premesse. La logica di Aristotele si basa su due valori di verità ed è governata da tre leggi: il principio di identità (ogni oggetto è uguale a se stesso), il principio di non contraddizione (due espressioni in contraddizione non possono essere entrambe vere) e il principio del terzo escluso (tertium non datur: ogni espressione deve essere o vera o falsa).

- Inizialmente la logica trattava affermazioni espresse in linguaggio naturale e si basava su regole di ragionamento molto semplici. Sfortunatamente, l'intrinseca ambiguità del linguaggio naturale conduce facilmente a paradossi, spesso nella forma di autologie (affermazioni che fanno riferimento a se stesse, come il paradosso del bugiardo: "questa affermazione è una bugia").

- Nella seconda metà del 1600 Leibniz riteneva che una larga parte del ragionamento umano potesse essere ridotta a calcoli e che questi ultimi potessero essere usati per risolvere molte divergenze di opinioni (da cui il motto "calculemus"). A tale scopo, Leibniz immaginò un linguaggio formale universale detto "characteristica universalis" in grado di esprimere concetti matematici, scientifici e metafisici, da usare nel contesto di un calcolo logico universale detto "calculus ratiocinator". Leibniz enunciò inoltre le principali proprietà di ciò che ora chiamiamo congiunzione, disgiunzione e negazione e introdusse il sistema di numerazione binario, i cui due simboli di base 0 e 1 possono essere messi in corrispondenza biunivoca con i due valori di verità falso e vero.

- La prima formulazione della logica in termini di un linguaggio matematico venne espressa nel 1847 da Boole nel suo libro "The Mathematical Analysis of Logic". Il suo obiettivo era quello di investigare le leggi fondamentali delle operazioni che avvengono nella mente umana durante il ragionamento. Boole individuò un'analogia tra le operazioni aritmetiche di addizione e moltiplicazione e le operazioni insiemistiche di unione e intersezione: per esempio $a \cdot (b + c) = (a \cdot b) + (a \cdot c)$ è simile ad $A \cap (B \cup C) = (A \cap B) \cup (A \cap C)$. Egli estese questa analogia alla logica introducendo e studiando le proprietà algebriche di un operatore di disgiunzione logica (OR) e un operatore di congiunzione logica (AND) accompagnati da un operatore di negazione logica (NOT) e dai valori di verità falso (0) e vero (1). Nello stesso anno De Morgan fornì nel suo libro "Formal Logic" degli importanti contributi allo studio dei sillogismi numericamente definiti. Nel 1880 Venn propose nel suo articolo "On the Diagrammatic and Mechanical Representation of Propositions and Reasonings" l'uso di particolari diagrammi per rappresentare gli insiemi e le loro operazioni che, grazie all'analogia di Boole, potevano essere impiegati anche nella logica. Alla fine del 1800 Schröder anticipò nel suo libro "The Algebra of Logic" l'importanza di trovare metodi rapidi per determinare le conseguenze di premesse arbitrarie.

- Nel XIX secolo ci si rese conto che, come la logica necessitava della matematica per superare l'ambiguità del linguaggio naturale, così la matematica necessitava di fondamenta adeguate da un lato e meccanismi deduttivi rigorosi per migliorare la qualità delle dimostrazioni dei teoremi dall'altro. Nel 1879 Frege propose nel suo "Begriffsschrift" la logica come un linguaggio per la matematica funzionale alla dimostrazione dei teoremi, dove eventuali elementi intuitivi o primitivi devono essere isolati come assiomi (o postulati o principi) e la prova deve procedere a partire da questi assiomi in modo puramente logico e senza salti. Frege introdusse inoltre un sistema logico con negazione, implicazione, quantificazione universale e tabelle di verità, che può essere visto come il primo calcolo dei predicati.

- Anche la formulazione delle teorie matematiche necessitava della logica. Per esempio, la cosiddetta teoria ingenua degli insiemi formulata da Cantor contiene numerosi paradossi: denotata con T la collezione di tutti gli insiemi che non sono elementi di se stessi, non si riesce a stabilire se T appartiene o no a se stesso. Per evitare questo genere di paradossi, nel 1903 Russell riconobbe che era necessario formulare la teoria degli insiemi sulla base di un gruppo di assiomi che evitassero definizioni circolari. Come diventerà chiaro con la teoria assiomatica degli insiemi successivamente sviluppata da Zermelo e Fraenkel, non tutte le collezioni di insiemi definibili nel linguaggio usato per la teoria degli insiemi sono insiemi; terminologicamente, si distingue tra classi e insiemi.

- Come conseguenza, agli inizi del 1900 prese piede il logicismo, una scuola di pensiero basata sulla visione di Frege secondo cui la matematica è riducibile alla logica. In particolare, il logicismo di Russell consisteva in due tesi principali. In primo luogo, tutte le verità matematiche possono essere tradotte in verità logiche, cioè il vocabolario della matematica è incluso nel vocabolario della logica. In secondo luogo, tutte le dimostrazioni dei teoremi della matematica possono essere riformulate come prove logiche, cioè l'insieme dei teoremi della matematica è incluso nell'insieme dei teoremi della logica. A supporto del logicismo, tra il 1910 e il 1913 Russell e Whitehead dimostrarono formalmente nel loro libro "Principia Mathematica" il grosso delle conoscenze matematiche del loro tempo utilizzando pure manipolazioni simboliche.

- All'apice del logicismo di Russell, nel 1920 Hilbert lanciò un programma per trovare una singola procedura formale per derivare tutti i teoremi della matematica, che equivale a riuscire a ridurre tutta la matematica in forma assiomatica e a provare che tale assiomatizzazione non porta a contraddizioni. Hilbert sosteneva che "una volta stabilito un formalismo logico, ci si può aspettare che sia possibile un trattamento sistematico, diciamo computazionale, delle formule logiche, in parte analogo alla teoria delle equazioni in algebra". Secondo Hilbert, le teorie complesse della matematica potevano essere fondate su teorie più semplici fino a basare l'intera matematica sull'aritmetica, cosicché provando la correttezza di quest'ultima si sarebbe provata la non contraddittorietà di tutta la matematica.

- Il colpo di grazia al programma di Hilbert e quindi al logicismo di Russell venne inflitto nel 1931 dai due teoremi di incompletezza dimostrati da Gödel (basati sul metodo della diagonale con cui Cantor mostrò che $|\mathbb{N}| < |\mathbb{R}|$ e su opportune enumerazioni che associano numeri naturali univoci a elementi di insiemi), che esprimiamo informalmente come segue:
  - Primo teorema di incompletezza di Gödel: In ogni teoria assiomatizzabile sufficientemente espressiva da formare affermazioni su ciò che può dimostrare, ci saranno sempre delle affermazioni vere che la teoria può esprimere ma non derivare dai suoi assiomi attraverso le sue regole di inferenza. Ciò significa che in ogni teoria matematica sufficientemente potente ci sono dei teoremi che non sono dimostrabili all'interno della teoria stessa; per dimostrarli, bisognerà ricorrere a una teoria più potente, ma anch'essa soffrirà della stessa limitazione.
  - Secondo teorema di incompletezza di Gödel: Ogni teoria assiomatizzabile sufficientemente espressiva da formare affermazioni sull'aritmetica non potrà mai dimostrare la propria correttezza. Questo risultato evidenzia come già una branca fondamentale della matematica quale l'aritmetica sollevi problemi di incompletezza dal punto di vista di un suo trattamento computazionale.
- Nel 1936 apparvero risultati negativi analoghi in ambito computazionale:
  - Church dimostrò che la logica dei predicati non è decidibile, cioè che non esiste alcun algoritmo in grado di stabilire in tempo finito se una formula predicativa arbitraria è soddisfacibile o meno.
  - Turing dimostrò che il problema della terminazione non è decidibile, cioè che non esiste alcun algoritmo in grado di stabilire in tempo finito se l'esecuzione di un algoritmo arbitrario su un'istanza arbitraria dei suoi dati di ingresso termina oppure no.
- L'esistenza in ogni sistema deduttivo di teoremi che non possono essere dimostrati e l'esistenza di problemi computazionali che non possono essere risolti da nessun algoritmo determinò l'impossibilità di ridurre la matematica alla logica e quindi la fine del logicismo. Dal 1940 in poi la logica continuò a svilupparsi non più come il fondamento universale di tutta la matematica, ma come una branca di essa, pur rimanendo aperta la possibilità di mettere a punto sistemi deduttivi che servano come fondamenti per parti specifiche della matematica.
- La logica è particolarmente importante anche nell'informatica per via della natura computazionale del concetto di deduzione. Facendo un paragone con il ruolo che il calcolo infinitesimale ha avuto nella fisica, a volte la logica viene indicata come il calcolo dell'informatica. A partire dai primi risultati di indecidibilità (anni 1930) e ancor più dall'avvento dei primi elaboratori elettronici (anni 1940) e dei primi linguaggi di programmazione di alto livello (anni 1950), la logica ha avuto un ruolo fondamentale e pervasivo nell'informatica:
  - Nella teoria della calcolabilità e nella teoria della complessità, la logica ha fornito caratterizzazioni del concetto di risolvibilità di un problema computazionale sin dall'introduzione del modello astratto di macchina di Turing come pure della visione computazionale della nozione di funzione matematica attraverso il λ-calcolo di Church. Inoltre, la logica è stata cruciale nello sviluppo della teoria della NP-completezza, la quale formalizza il concetto di esplosione combinatoria. Ad esempio, il problema della soddisfacibilità di una formula di logica proposizionale è l'archetipo di problema computazionale che non sappiamo come risolvere in modo efficiente.
  - Nell'architettura degli elaboratori, i circuiti che formano i componenti hardware sono costituiti da porte che, come scoperto da Shannon, implementano le operazioni logiche introdotte da Boole. Sebbene Boole intendesse studiare le leggi del ragionamento della mente umana, l'applicazione principale della sua teoria si trova dunque nell'ingegneria elettronica e nell'industria informatica.
  - Nella progettazione dell'hardware e del software, la logica viene utilizzata per verificare la correttezza di un progetto con un grado di confidenza che va oltre quello del testing. Per la precisione, viene spesso impiegata una variante della logica classica detta logica temporale la quale comprende operatori che consentono di esprimere affermazioni sull'ordine in cui certe computazioni hanno luogo oppure sul fatto che certe computazioni possano o debbano avere luogo.

  - Nella gestione delle basi di dati, un linguaggio comunemente impiegato per reperire e manipolare dati è SQL. Questo linguaggio di interrogazione per basi di dati risulta essere un'implementazione della logica del primo ordine.
  - Nell'intelligenza artificiale, la logica è essenziale per formalizzare le conoscenze già acquisite e le regole di ragionamento sulla base delle quali i sistemi esperti riescono a dedurre nuove conoscenze. Lo stesso meccanismo si applica ai sistemi per la dimostrazione automatica di teoremi.
  - Nella programmazione degli elaboratori, è necessario formalizzare la semantica dei linguaggi di programmazione in modo che i loro costrutti siano coerenti in implementazioni diverse dello stesso linguaggio e vengano compresi senza ambiguità da parte dei programmatori. La logica fornisce strumenti per sviluppare tale semantica e per ragionare sulle proprietà dei programmi (come le triple di Hoare). Inoltre, esistono paradigmi di programmazione – come quello dichiarativo di natura logica di cui il linguaggio Prolog è il principale esponente – in cui la logica viene usata per esprimere i programmi e i suoi meccanismi di ragionamento vengono sfruttati per eseguire i programmi.

8.2 Prolog: Clausole di Horn e Strategia di Risoluzione SLD

- Il linguaggio Prolog (PROgrammation en LOGique) venne sviluppato nel 1972 da Colmerauer e Roussel per l'elaborazione del linguaggio naturale, basandosi sull'interpretazione procedurale delle clausole di Horn data da Kowalski e sul metodo di risoluzione di Robinson. Esso cominciò a essere riconosciuto come un utile strumento di programmazione grazie al compilatore implementato nel 1977 da Warren.
- Il Prolog si fonda sulla logica dei predicati limitandosi a formule espresse come clausole di Horn per motivi computazionali. Un programma Prolog si presenta come una collezione di definizioni di predicati, interpretati come relazioni matematiche, e relative applicazioni e composizioni che danno luogo a fatti e regole. I primi consentono di formalizzare la conoscenza su un certo dominio, mentre le seconde permettono di ricavare ulteriore conoscenza.
- L'esecuzione di un programma Prolog viene attivata mediante un'interrogazione espressa come vincolo e il suo risultato è un valore di verità. Se nell'interrogazione sono presenti delle variabili, il risultato è accompagnato dai legami creati via unificazione per tali variabili durante l'applicazione a tempo d'esecuzione dell'algoritmo di refutazione basato sul metodo di risoluzione di Robinson.
- Diversamente dalla programmazione imperativa, in Prolog:
  - I programmi non sono istruzioni di assegnamento da eseguire nell'ordine stabilito dalle istruzioni di controllo del flusso, ma collezioni di predicati da valutare.
  - Le esecuzioni dei programmi non sono sequenze di passi che modificano il contenuto della memoria, ma creano legami immutabili per le variabili e restituiscono valori di verità.
  - Non ci sono effetti collaterali in quanto la valutazione di un predicato non può modificare variabili al di fuori del suo ambiente locale.
  - La conseguente trasparenza referenziale fa sì che venga restituito sempre lo stesso risultato ogni volta che un predicato viene valutato sugli stessi argomenti.
  - La ricorsione è l'unico strumento linguistico per rappresentare l'iterazione e il tipo di dato ricorsivo lista è l'unico tipo di dato strutturato disponibile, con la gestione dinamica della memoria per le liste completamente automatizzata.
- Diversamente da Haskell, in Prolog è possibile definire solo funzioni che restituiscono un valore di verità – seguendo la sintassi delle clausole di Horn con $\rightarrow$ peraltro – e non esiste un reale sistema di tipi. D'altro canto, nelle definizioni delle funzioni Prolog (predicati) si esprime solo conoscenza positiva – i casi in cui il risultato è falso non vengono formalizzati in quanto automaticamente derivati dalla mancanza di unificazione – e gli argomenti effettivi possono essere specificati anche solo parzialmente nelle invocazioni – così vengono calcolati i valori di quegli argomenti che producono vero come risultato.
- Il linguaggio Prolog consente di definire delle relazioni e di compiere delle interrogazioni su di esse. Pertanto in Prolog le clausole di Horn costituite da fatti e regole vengono usate per rappresentare informazioni sotto forma di relazioni tra dati. Precisamente, la sintassi di un programma Prolog prevede una sequenza di clausole di Horn date da fatti e regole che sono terminate dal punto:

  - Ogni fatto presente nel programma rappresenta un'informazione ed è costituito da una formula atomica della logica dei predicati:
    `P(t_1, ..., t_n).`
  - Ogni regola presente nel programma rappresenta una relazione tra più informazioni:
    `P(t_1, ..., t_n) :- Q_1(t_1_1, ..., t_1_m1), ..., Q_k(t_k_1, ..., t_k_mk).`
    La parte sinistra della regola è una formula atomica della logica dei predicati detta testa, il simbolo `:-` rappresenta l'implicazione nel verso $\leftarrow$ e la parte destra della regola è una sequenza non vuota di formule atomiche della logica dei predicati detta corpo. Le virgole che separano le formule presenti all'interno del corpo della regola rappresentano delle congiunzioni.

- Il metodo di risoluzione di Robinson viene impiegato – insieme a un algoritmo di unificazione efficiente – per risolvere problemi definiti attraverso clausole di Horn costituite da vincoli, i quali vengono interpretati come obiettivi da raggiungere istanziandone le parti eventualmente non specificate. Precisamente, l'esecuzione di un programma Prolog viene attivata da un obiettivo descritto come una regola senza testa in cui il simbolo `:-` è sostituito dal simbolo `?-`, cioè una clausola di Horn data da un vincolo. Il raggiungimento dell'obiettivo viene perseguito applicando il metodo di risoluzione ai fatti e alle regole del programma e all'obiettivo dato. In caso di refutazione, se l'obiettivo non è una clausola ground allora le sue variabili vengono istanziate sulla base delle unificazioni effettuate durante le applicazioni della regola di inferenza per risoluzione.

- In un contesto guidato dagli obiettivi come quello del Prolog, ogni regola di un programma si presta a una duplice intepretazione:
  - Il significato dichiarativo della regola è che la formula di testa `P(t_1, ..., t_n)` è vera se le formule del corpo `Q_1(t_1_1, ..., t_1_m1), ..., Q_k(t_k_1, ..., t_k_mk)` sono vere.
  - Il significato procedurale della regola è che per raggiungere l'obiettivo `P(t_1, ..., t_n)` bisogna prima raggiungere gli obiettivi `Q_1(t_1_1, ..., t_1_m1), ..., Q_k(t_k_1, ..., t_k_mk)`.

- L'algoritmo di refutazione basato sul metodo di risoluzione di Robinson applica ripetutamente la regola di inferenza per risoluzione e questo può generare un numero eccessivo di clausole irrilevanti o ridondanti. Per evitare ciò, si utilizzano delle strategie di risoluzione che scelgono in modo mirato le clausole da cui generare una risolvente. Queste strategie si dividono in complete e incomplete a seconda che riescano sempre a derivare $\square$ da un insieme insoddisfacibile di clausole oppure no. La strategia di risoluzione adottata in Prolog è SLD.

- Dato un insieme di clausole C, diciamo che una dimostrazione per risoluzione $c_1 \dots c_n$ a partire da C è ottenuta applicando:
  - La strategia di risoluzione lineare sse per ogni $i = 2, \dots, n$ la clausola c_i è la risolvente della clausola precedente c_{i-1} e di una clausola $c' \in C \cup \{c_1, \dots, c_{i-1}\}$.
  - La strategia di risoluzione di input sse per ogni $i = 2, \dots, n$ la clausola c_i è la risolvente di una clausola di ingresso $c \in C$ e di una clausola $c' \in C \cup \{c_1, \dots, c_{i-1}\}$.
  - La strategia di risoluzione SLD sse per ogni $i = 2, \dots, n$ la clausola c_i è la risolvente della clausola precedente c_{i-1} e di una clausola di ingresso $c \in C$ (combinazione efficiente di risoluzione lineare e risoluzione di input).

- Teorema: La strategia di risoluzione lineare è completa per tutti gli insiemi di clausole.

- Teorema: La strategie di risoluzione di input ed SLD sono complete per gli insiemi di clausole di Horn.

- Dati un programma Prolog (insieme di clausole C) e un obiettivo (clausola iniziale c_1), in virtù dell'interpretazione procedurale e della strategia di risoluzione SLD bisogna trovare all'interno del programma, per tutte le formule atomiche nell'obiettivo considerate da sinistra a destra, dei fatti che siano unificabili con quelle formule o delle regole le cui teste siano unificabili con quelle formule. Nel caso la risoluzione avvenga con una regola, le formule atomiche nel corpo della regola entrano a far parte del nuovo obiettivo. Per motivi di efficienza, conviene che le regole ricorsive siano ricorsive a destra, cioè che i predicati presenti nella testa di tali regole compaiano nelle formule atomiche più a destra nel corpo delle regole stesse.

- Poiché ogni formula atomica presente in un obiettivo potrebbe essere unificabile con più fatti e teste di regole di un programma, in Prolog la risoluzione della formula atomica più a sinistra in un obiettivo viene esaminata rispetto a tutte le clausole del programma considerate nell'ordine in cui sono scritte. Per motivi di efficienza, all'interno dei programmi conviene quindi elencare i fatti prima delle regole.
- L'esecuzione di un programma Prolog su un obiettivo può essere rappresentata graficamente tramite un albero e-o, così chiamato perché ogni nodo rappresenta un obiettivo (congiunzione) e ha tanti nodi figli quanti sono i fatti e le regole alternativi tra loro con i quali può avvenire la risoluzione (disgiunzione). Durante l'applicazione della risoluzione, l'albero viene costruito in profondità per via dell'ordine LIFO imposto dalla strategia SLD.
- La radice dell'albero e-o contiene l'obiettivo iniziale, mentre ogni foglia può contenere $\square$ oppure un obiettivo alla cui formula atomica più a sinistra non è applicabile la risoluzione. Ogni ramo dell'albero è etichettato con i legami creati via unificazione durante la corrispondente applicazione della regola di inferenza per risoluzione. Ogni cammino dalla radice a una foglia contenente $\square$ è un cammino di successo, mentre tutti gli altri cammini sono cammini di fallimento o cammini infiniti (considerare le clausole di un programma Prolog sempre nello stesso ordine rende la strategia SLD incompleta).
- Esempio: Consideriamo il seguente programma Prolog:

```
genitore(giovanni, andrea).
genitore(andrea, paolo).
antenato(X, Y) :- genitore(X, Y).
antenato(X, Y) :- genitore(X, Z), antenato(Z, Y).
```

dove il predicato `genitore(X, Y)` (risp. `antenato(X, Y)`) indica che `X` è genitore (risp. antenato) di `Y`. Osserviamo che ogni clausola che contiene delle variabili ha una quantificazione universale implicita su tutte quelle variabili, quindi le due regole del programma dato vanno intese nel seguente modo:

$\forall X \forall Y \, (genitore(X, Y) \rightarrow antenato(X, Y))$
$\forall X \forall Y \forall Z \, (genitore(X, Z) \wedge antenato(Z, Y) \rightarrow antenato(X, Y))$

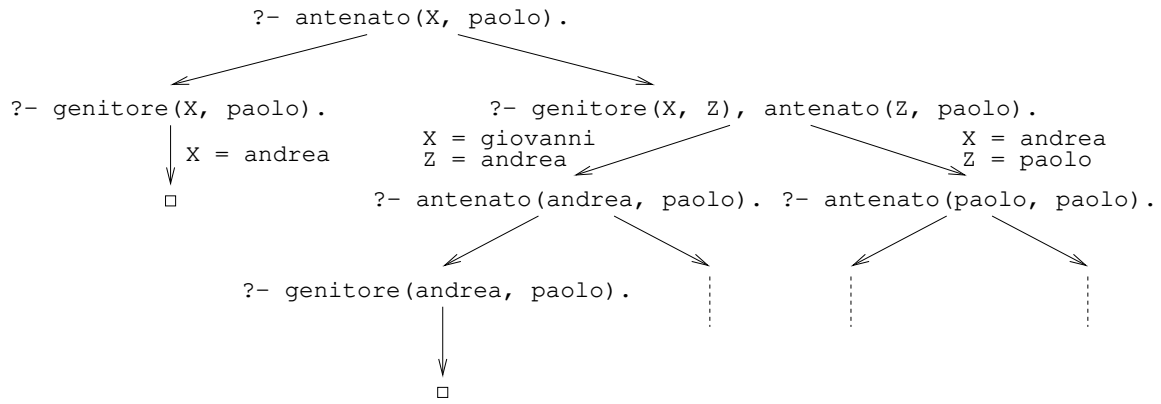
Consideriamo poi l'obiettivo:

```
?- antenato(X, paolo).
```

Osserviamo che ogni obiettivo che contiene delle variabili ha una quantificazione esistenziale implicita su tutte quelle variabili, quindi l'obiettivo dato va inteso nel seguente modo:

$\exists X \; antenato(X, paolo)\,?$

che dal punto di vista procedurale consiste nel trovare un antenato della persona specificata sulla base delle informazioni fornite dal programma. Poiché l'esecuzione di un programma Prolog procede per refutazione dell'obiettivo, l'obiettivo deve essere negato prima di iniziare ad applicare la regola di inferenza per risoluzione e quindi la quantificazione esistenziale che precede eventualmente l'obiettivo diventa universale.
L'albero e-o risultante è il seguente:



in cui si nota la presenza di due cammini di successo (soluzioni `X = andrea` e `X = giovanni`) assieme a diversi cammini che conducono al fallimento. ■fltplf_18

8.3 Prolog: Sintassi dei Termini e Predicati Predefiniti

- I caratteri utilizzabili all'interno di un programma Prolog comprendono le 26 lettere minuscole, le 26 lettere maiuscole, le 10 cifre decimali, il sottotratto, la barra verticale, i simboli di punteggiatura, le parentesi tonde e quadre, gli operatori aritmetici e relazionali e i caratteri di spaziatura. Le lettere minuscole sono considerate diverse dalle corrispondenti lettere maiuscole (case sensitivity).
- I caratteri possono essere combinati per formare argomenti di predicati così classificati:
  - Atomi che si dividono in:
    - Costanti costituite da:
      - numeri interi e reali;
      - sequenze di lettere, cifre e sottotratti che iniziano con una minuscola;
      - sequenze racchiuse tra apici di lettere, cifre e sottotratti che iniziano con una maiuscola.
    - Variabili costituite da:
      - sequenze di lettere, cifre e sottotratti che iniziano con una maiuscola;
      - singoli sottotratti `_`, che rappresentano un qualsiasi valore.
  - Termini composti costituiti dalla concatenazione di:
    - Funtori che sono espressi come le costanti non numeriche, cioè costituiti da:
      - sequenze di lettere, cifre e sottotratti che iniziano con una minuscola;
      - sequenze racchiuse tra apici di lettere, cifre e sottotratti che iniziano con una maiuscola.
    - Argomenti separati da virgole e racchiusi tra parentesi tonde, costituiti da:
      - atomi;
      - termini composti.
  - Dati strutturati che si dividono in:
    - Liste costituite da sequenze di elementi separati da virgole e racchiusi tra parentesi quadre, dotate di un operatore | per distinguere un certo numero di elementi iniziali dai successivi.
    - Stringhe costituite da sequenze di caratteri racchiusi tra virgolette, implementate come liste.
- Esempi: `10` e `-13.5` sono costanti numeriche, `gatto` e `'Proprietario_di'` sono costanti simboliche o funtori, `X` e `Risultato_operazione` sono variabili, `"ciao!"` e `"Divisione illegale"` sono stringhe, `[]` e `[1, 2, 3]` sono liste.
- I predicati, i cui nomi seguono le stesse convenzioni dei funtori, sono definiti, spesso ricorsivamente, tramite sequenze di fatti e regole. Eventuali commenti devono essere racchiusi tra `/*` e `*/` oppure iniziare con `%` ed estendersi su una singola linea.
- Esempi relativi alle liste:
  - Predicato per stabilire se una sequenza di caratteri è una lista:

    ```
    lista([]).
    lista([X | L]).                    /* lista([_ | _]). */
    ```

    I due fatti specificano le sequenze di caratteri che sono conformi alla sintassi delle liste in Prolog. Qualsiasi altra sequenza non unificabile col formato degli argomenti nei due fatti non è una lista.
  - Predicato per stabilire se un elemento appartiene a una lista (la lista vuota non va bene come caso base della ricorsione perché rappresenta la situazione in cui l'elemento non appartiene alla lista, mentre in Prolog si esprime solo conoscenza positiva):

    ```
    membro(X, [X | L]).                /* membro(X, [X | _]).                */
    membro(X, [Y | L]) :- membro(X, L).  /* membro(X, [_ | L]) :- membro(X, L). */
    ```

    Dati gli obiettivi:

    ```
    ?- membro(2, []).
    ?- membro(2, [0, 2, 4]).
    ?- membro(Z, [1, 3, 5]).
    ```

    abbiamo che il primo fallisce, il secondo ha successo e il terzo dà luogo a tanti cammini di successo (ciascuno con la propria istanziazione di `Z`) quanti sono gli elementi presenti nella lista. Dunque, a seconda di come viene specificato l'obiettivo, questo predicato può essere usato sia per cercare un elemento in una lista, sia per attraversare tutti gli elementi di una lista.

- Predicato per stabilire se una lista è un prefisso di un'altra lista (cioè compare all'inizio):

```
prefisso([], L)              :- lista(L).
prefisso([X | L1], [X | L2]) :- prefisso(L1, L2).
```

- Predicato per stabilire se una lista è un suffisso di un'altra lista (cioè compare alla fine):

```
suffisso([], L)        :- lista(L).
suffisso(L, L)         :- lista(L).
suffisso(L1, [_ | L2]) :- suffisso(L1, L2).
```

- Predicato per stabilire se una lista è una sottolista di un'altra lista (cioè compare all'interno):

```
sottolista(L1, L2)       :- prefisso(L1, L2).
sottolista(L1, [_ | L2]) :- sottolista(L1, L2).
```

- Predicato per aggiungere un elemento all'inizio di una lista:

```
inserisci_elem(X, L, [X | L]).
```

- Predicato per aggiungere un elemento alla fine di una lista:

```
accoda_elem(X, [], [X]).
accoda_elem(X, [Y | L], [Y | LX]) :- accoda_elem(X, L, LX).
```

- Predicato per concatenare due liste:

```
concatena(L, [], L)        :- lista(L).
concatena(L1, [X | L2], L) :- accoda_elem(X, L1, L1X), concatena(L1X, L2, L).
```

- Predicato per invertire l'ordine degli elementi di una lista:

```
inverti([], []).
inverti([X | L1], L2) :- inverti(L1, L1Inv), accoda_elem(X, L1Inv, L2).
```

- Al fine di agevolare l'attività di programmazione, il Prolog mette a disposizione simboli e predicati predefiniti. Tra questi abbiamo i consueti operatori aritmetici e relazionali in notazione infissa, su termini di tipo numerico, che abilitano la valutazione di espressioni aritmetiche in un contesto logico:
  - I simboli `+`, `-`, `*`, `/`, `mod`, `^` rappresentano rispettivamente le operazioni di addizione, sottrazione, moltiplicazione, divisione, resto della divisione, elevamento a potenza su operandi numerici (sono funtori, non predicati; tutte le variabili eventualmente presenti nei due operandi devono essere già state istanziate con valori numerici).
  - I simboli `=:=`, `=\=`, `>`, `>=`, `<`, `=<` rappresentano rispettivamente i predicati di uguale-a, diverso-da, maggiore-di, maggiore-uguale, minore-di, minore-uguale su operandi numerici (tutte le variabili eventualmente presenti nei due operandi devono essere già state istanziate con valori numerici).
  - Il predicato `is` ha successo sse il suo operando sinistro è unificabile con il valore numerico del suo operando destro (tutte le variabili eventualmente presenti nell'operando destro devono essere già state istanziate con valori numerici). Questo è l'unico predicato in Prolog che consente di valutare un'espressione aritmetica e di legarne il valore a una variabile.
- I seguenti predicati in notazione infissa costituiscono delle estensioni dei predicati di uguaglianza a termini di tipo arbitrario (quindi non necessariamente numerico):
  - Il predicato `=` ha successo sse i suoi due operandi sono unificabili e in tal caso produce il loro upg (non effettua però l'occur check).
  - Il predicato `\=` ha successo sse i suoi due operandi non sono unificabili.
  - Il predicato `==` ha successo sse i suoi due operandi sono sintatticamente identici (delle variabili istanziate eventualmente presenti nei due operandi viene considerato il valore anziché il nome).
  - Il predicato `\==` ha successo sse i suoi due operandi non sono sintatticamente identici.

- Esempi:

```
?- X is 1 + 2.          ?- 4 - 1 =:= 1 + 2.    ?- a < -30.     ?- X = 1 + 2.
  yes X=3                 yes                     domain error    yes X=1+2
?- X is Y + 2.          ?- 2.5 > 9.            ?- 2 = 2.       ?- f(a, Y) = f(Z, g(7)).
  instantiation error     no                      yes             yes Z=a, Y=g(7)
?- 3 is 1 + 2.          ?- Z < 15.             ?- a = b.       ?- X == 2.
  yes                     instantiation error     no              no
?- 4 - 1 is 1 + 2.      ?- Z is 4, Z < 15.     ?- a = A.       ?- X = 2, X == 2.
  no                      yes Z=4                 yes A=a         yes X=2
```

- Esempi relativi alle funzioni numeriche (se il risultato di una funzione con n argomenti non è un valore di verità, allora il corrispondente predicato in Prolog ha $n+1$ argomenti dei quali l'ultimo è il risultato):
  - Predicato per calcolare il fattoriale di un numero naturale:

```
fattoriale(0, 1).
fattoriale(N, F) :- N > 0, N1 is N - 1,
                    fattoriale(N1, F1), F is N * F1.
```

  - Predicato per calcolare l'n-esimo numero di Fibonacci:

```
fibonacci(0, 0).
fibonacci(1, 1).
fibonacci(N, F) :- N > 1, N1 is N - 1, N2 is N - 2,
                   fibonacci(N1, F1), fibonacci(N2, F2), F is F1 + F2.
```

- Ulteriori esempi relativi alle liste:
  - Predicato per calcolare il numero di elementi presenti in una lista:

```
lunghezza([], 0).
lunghezza([_ | L], N) :- lunghezza(L, M), N is M + 1.
```

  - Il predicato che stabilisce se un elemento appartiene a una lista può essere reso più efficiente ridefinendolo come segue:

```
membro(X, [X | _]).
membro(X, [Y | L]) :- X \== Y, membro(X, L).
```

    In questo modo le due clausole sono alternative tra loro e quindi al più una di esse può dare luogo a un cammino di successo. Per esempio, nel caso in cui l'obiettivo sia il seguente:

```
?- membro(1, [3, 1, 5, 1, 7]).
```

    c'è un unico cammino di successo che termina quando il secondo argomento diventa `[1, 5, 1, 7]`. La vecchia definizione del predicato avrebbe invece avuto due cammini di successo.
  - Predicato per rimuovere tutte le occorrenze di un elemento da una lista:

```
rimuovi_elem(_, [], []).
rimuovi_elem(X, [X | L], LNoX)       :- rimuovi_elem(X, L, LNoX).
rimuovi_elem(X, [Y | L], [Y | LNoX]) :- X \== Y, rimuovi_elem(X, L, LNoX).
```

  - Predicato per fondere ordinatamente due liste ordinate di valori numerici:

```
fondi(L, [], L) :- lista(L).
fondi([], L, L) :- L \== [], lista(L).
fondi([X | L1], [Y | L2], [X | L])    :- X < Y, fondi(L1, [Y | L2], L).
fondi([X | L1], [Y | L2], [X, Y | L]) :- X =:= Y, fondi(L1, L2, L).
fondi([X | L1], [Y | L2], [Y | L])    :- X > Y, fondi([X | L1], L2, L).
```

- Esempi relativi agli algoritmi di ordinamento:
  - Mergesort:

```
mergesort([], []).
mergesort([X], [X]).
mergesort([X1, X2 | L], LOrd) :- dimezza([X1, X2 | L], L1, L2),
                                 mergesort(L1, L1Ord), mergesort(L2, L2Ord),
                                 fondi(L1Ord, L2Ord, LOrd).
dimezza([], [], []).
dimezza([X], [X], []).
dimezza([X1, X2 | L], [X1 | L1], [X2 | L2]) :- dimezza(L, L1, L2).
```

  - Quicksort:

```
quicksort([], []).
quicksort([P | L], LOrd) :- partiziona(L, P, LInf, LSup),
                            quicksort(LInf, LInfOrd), quicksort(LSup, LSupOrd),
                            concatena(LInfOrd, [P | LSupOrd], LOrd).
partiziona([], _, [], []).
partiziona([X | L], P, [X | LInf], LSup) :- X < P, partiziona(L, P, LInf, LSup).
partiziona([X | L], P, LInf, [X | LSup]) :- X >= P, partiziona(L, P, LInf, LSup).
```

- Il Prolog mette a disposizione dei predicati predefiniti per discriminare tra differenti tipi di termini:
  - `var(T)` ha successo sse `T` è una variabile non istanziata, cioè una variabile non legata a un valore.
  - `nonvar(T)` ha successo sse `T` non è una variabile non istanziata.
  - `integer(T)` ha successo sse `T` è istanziato con una costante numerica intera.
  - `float(T)` ha successo sse `T` è istanziato con una costante numerica reale.
  - `number(T)` ha successo sse `T` è istanziato con una costante numerica.
  - `atom(T)` ha successo sse `T` è istanziato con una costante non numerica.
  - `compound(T)` ha successo sse `T` è istanziato con un termine composto:
    - `functor(T, F, N)` ha successo sse `T` è un termine composto il cui funtore ha nome `F` ed è applicato a `N` argomenti.
    - `arg(N, T, A)` ha successo sse `T` è un termine composto il cui `N`-esimo argomento è `A`.
- Ulteriori esempi relativi alle funzioni numeriche:
  - Predicato per calcolare quoziente e resto della divisione tra due numeri naturali:

```
divisione(X, Y, Q, R) :- integer(X), X >= 0, integer(Y), Y > 0,
                         Q is X / Y, R is X mod Y.
```

  - Predicato per calcolare la somma o la differenza a seconda di quali operandi risultano istanziati (l'uso del predicato `is` li costringe a essere istanziati con valori numerici):

```
somma_diff(X, Y, Z) :- nonvar(X), nonvar(Y), Z is X + Y.
somma_diff(X, Y, Z) :- nonvar(X), nonvar(Z), Y is Z - X.
somma_diff(X, Y, Z) :- nonvar(Y), nonvar(Z), X is Z - Y.
```

  - Predicato per calcolare il prodotto o il quoziente a seconda di quali operandi risultano istanziati (il valore numerico di `Y` nella terza regola e di `X` nella quinta regola è irrilevante ma deve essere presente, da cui il controllo finale `number` su quella variabile):

```
prod_quoz(X, Y, Z) :- nonvar(X), nonvar(Y), Z is X * Y.
prod_quoz(X, Y, Z) :- nonvar(X), nonvar(Z), X =\= 0, Y is Z / X.
prod_quoz(X, Y, Z) :- nonvar(X), nonvar(Z), X =:= 0, Z =:= 0, number(Y).
prod_quoz(X, Y, Z) :- nonvar(Y), nonvar(Z), Y =\= 0, X is Z / Y.
prod_quoz(X, Y, Z) :- nonvar(Y), nonvar(Z), Y =:= 0, Z =:= 0, number(X).
```

- Esempi relativi a termini e loro unificazione:
  - Predicato per stabilire se un termine è ground, cioè privo di variabili non istanziate:

```
ground(T) :- integer(T).
ground(T) :- float(T).
ground(T) :- atom(T).
ground(T) :- compound(T), functor(T, _, N), ground_tutti_arg(N, T).
ground_tutti_arg(0, _).
ground_tutti_arg(N, T) :- N > 0, arg(N, T, A), ground(A),
                          N1 is N - 1, ground_tutti_arg(N1, T).
```

  - Predicato per stabilire se un termine è sottotermine di un altro (`sottoterm(T, T)` non va bene come caso base quando uno dei due termini è una variabile perché verrebbe unificato con l'altro):

```
sottoterm(T1, T2) :- T1 == T2.
sottoterm(T1, T2) :- T1 \== T2, compound(T2),
                     functor(T2, _, N), sottoarg(T1, N, T2).
sottoarg(T1, N, T2) :- arg(N, T2, A), sottoterm(T1, A).
sottoarg(T1, N, T2) :- N > 1, N1 is N - 1, sottoarg(T1, N1, T2).
```

  - Algoritmo di unificazione di Robinson (stesso effetto del predicato = ma con occur check):

```
unifica(T1, T2, [])         :- integer(T1), integer(T2), T1 =:= T2.
unifica(T1, T2, [])         :- float(T1), float(T2), T1 =:= T2.
unifica(T1, T2, [])         :- atom(T1), atom(T2), T1 == T2.
unifica(T1, T2, [[T1, T2]]) :- var(T1), var(T2).
unifica(T1, T2, [[T2, T1]]) :- var(T1), integer(T2).
unifica(T1, T2, [[T2, T1]]) :- var(T1), float(T2).
unifica(T1, T2, [[T2, T1]]) :- var(T1), atom(T2).
unifica(T1, T2, [[T2, T1]]) :- var(T1), compound(T2), occur_check(T1, T2).
unifica(T1, T2, [[T1, T2]]) :- var(T2), integer(T1).
unifica(T1, T2, [[T1, T2]]) :- var(T2), float(T1).
unifica(T1, T2, [[T1, T2]]) :- var(T2), atom(T1).
unifica(T1, T2, [[T1, T2]]) :- var(T2), compound(T1), occur_check(T2, T1).
unifica(T1, T2, L)          :- compound(T1), compound(T2),
                               functor(T1, F, N), functor(T2, F, N),
                               unifica_tutti_arg(N, T1, T2, L).
unifica_tutti_arg(0, _, _, []).
unifica_tutti_arg(N, T1, T2, L) :- N > 0, arg(N, T1, A1), arg(N, T2, A2),
                                   unifica(A1, A2, LA),
                                   N1 is N - 1, unifica_tutti_arg(N1, T1, T2, L1),
                                   coerenti(L1, LA), concatena(L1, LA, L).
coerenti(_, []).
coerenti(L1, [[T, X] | L2]) :- no_stessa_v_diff_t(X, T, L1), coerenti(L1, L2).
no_stessa_v_diff_t(_, _, []).
no_stessa_v_diff_t(X, T, [[_, Y] | L]) :- X \== Y, no_stessa_v_diff_t(X, T, L).
no_stessa_v_diff_t(X, T, [[T, X] | L]) :- no_stessa_v_diff_t(X, T, L).
occur_check(_, T) :- integer(T).
occur_check(_, T) :- float(T).
occur_check(_, T) :- atom(T).
occur_check(X, T) :- var(T), X \== T.
occur_check(X, T) :- compound(T), functor(T, _, N), occur_check_tutti_arg(X, N, T).
occur_check_tutti_arg(_, 0, _).
occur_check_tutti_arg(X, N, T) :- N > 0, arg(N, T, A), occur_check(X, A),
                                  N1 is N - 1, occur_check_tutti_arg(X, N1, T).
```

8.4 Prolog: Negazione, Taglio, Input/Output, Predicati Avanzati

- Le formule atomiche presenti nel corpo delle regole di un programma Prolog e negli obiettivi sono tutte positive perché questo è il formato delle clausole di Horn con $\rightarrow$. A volte sarebbe tuttavia utile poter esprimere anche formule atomiche negative. A tale scopo il Prolog mette a disposizione il predicato `not` la cui interpretazione è conforme all'assunzione di mondo chiuso: è vero solo ciò che è stabilito dai fatti del programma o che può essere dedotto applicando le regole del programma, tutto il resto è falso.

- `not(T)`, a volte indicato con `\+(T)`, ha successo sse `T` fallisce, cioè il Prolog implementa la negazione come fallimento. A causa della possibile presenza di cammini infiniti nell'albero e-o di `T`, la regola di negazione come fallimento è un'approssimazione dell'assunzione di mondo chiuso e la negazione così implementata non coincide con la negazione della logica matematica. La regola di negazione come fallimento è corretta e completa quando l'argomento `T` è ground, cioè privo di variabili non istanziate.

- Esempi:

  – Dati il seguente programma Prolog:

    ```
    studente(franco).
    sposato(roberto).
    studente_celibe(X) :- studente(X), not(sposato(X)).
    ```

    e il seguente obiettivo col quale si chiede se esiste uno studente celibe:

    ```
    ?- studente_celibe(X).
    ```

    abbiamo successo con `X` istanziata a `franco`. Osserviamo che quando viene considerata la formula atomica `not(sposato(X))` durante l'esecuzione, la variabile `X` è già stata istanziata e quindi l'argomento del `not` è assimilabile a un termine ground.
    Se invece la regola del programma fosse stata espressa nel seguente modo:

    ```
    studente_celibe(X) :- not(sposato(X)), studente(X).
    ```

    allora l'argomento del `not` non sarebbe stato assimilabile a un termine ground. Poiché `X` sarebbe stata istanziata a `roberto`, avremmo erroneamente avuto un fallimento. Ciò mostra che la regola della negazione come fallimento non è sempre corretta e completa quando l'argomento del `not` non è ground.

  – L'occur check può essere riformulato tramite la singola clausola:

    ```
    occur_check(X, T) :- not(sottoterm(X, T)).
    ```

    a patto di invocarla con `T` ground.

- Dati un programma Prolog e un obiettivo, la costruzione in profondità del corrispondente albero e-o viene sospesa quando si raggiunge la foglia di un cammino di successo. A quel punto l'interprete chiede all'utente se vuole terminare l'esecuzione oppure se vuole continuare l'esecuzione in cerca di altre soluzioni del problema. In quest'ultimo caso, l'esecuzione riprende risalendo dalla foglia verso la radice fino a incontrare il primo nodo la cui formula atomica più a sinistra ammette un'unificazione alternativa a quelle che avevano condotto ai cammini precedentemente intrapresi.

- Poiché non è noto a priori se l'esecuzione di un programma Prolog rispetto a un obiettivo avrà successo, la costruzione in profondità del corrispondente albero e-o equivale ad agire per tentativi e revoche (backtrack). Prima si tenta col cammino più a sinistra. Se questo è un cammino di fallimento oppure è un cammino di successo ma l'utente vuole trovare ulteriori soluzioni al problema, allora si torna indietro sui passi appena compiuti annullando gli eventuali legami creati per unificazione e poi si tenta col cammino successivo e così via per i rimanenti cammini. L'esecuzione fallisce sse tutti i cammini sono cammini di fallimento.

- Il Prolog mette a disposizione i seguenti predicati predefiniti di natura logica che hanno un impatto sulla costruzione dell'albero e-o:
  - `true` ha sempre successo e quindi fa avanzare la costruzione del cammino.
  - `fail` non ha mai successo e quindi blocca la costruzione del cammino.
  - `!`, detto taglio, ha successo una e una sola volta e ha l'effetto collaterale di bloccare alcuni cammini. Il taglio ha lo scopo di ottenere prestazioni migliori e di evitare soluzioni ridondanti, ma va usato con attenzione perché la sua introduzione nelle clausole di un programma potrebbe alterare l'insieme delle soluzioni di un problema rappresentato da un certo obiettivo.
- Supponiamo che un nodo di un albero e-o contenga l'obiettivo `Q_1, ..., Q_h, !, Q_h+1, ..., Q_k`, che il taglio evidenziato nell'obiettivo sia stato introdotto per via dell'ultima risoluzione effettuata e che da questo nodo parta un cammino che porta al nodo contenente l'obiettivo `!, Q_h+1, ..., Q_k`. Giunti a quest'ultimo nodo, il taglio ha successo e vengono bloccati tutti gli eventuali cammini che iniziano dai fratelli destri dei nodi che si trovano lungo il cammino tra i due nodi che contengono i due obiettivi considerati. Poi si procede con l'obiettivo rimanente `Q_h+1, ..., Q_k`. Quando l'esecuzione torna indietro al nodo contenente l'obiettivo `!, Q_h+1, ..., Q_k` a causa di fallimento o della ricerca di ulteriori soluzioni al problema, l'esecuzione passa direttamente al nonno del nodo contenente l'obiettivo `Q_1, ..., Q_h, !, Q_h+1, ..., Q_k` in virtù del fatto che il taglio ha già avuto successo e che tutti gli eventuali cammini che iniziano dai fratelli destri dei nodi che si trovano lungo il cammino tra i due nodi che contengono i due obiettivi considerati sono stati bloccati (v. Fig. 8.1).
- Esempi:
  - Il predicato che stabilisce se un elemento appartiene a una lista può essere reso più efficiente (nel senso di evitare di unificare anche con `membro(X, [Y | L])` e subito dopo scoprire che `X \== Y` fallisce) ridefinendolo come segue:

    ```
    membro(X, [X | _]) :- !.
    membro(X, [_ | L]) :- membro(X, L).
    ```

    Tuttavia osserviamo che l'obiettivo:

    ```
    ?- membro(Z, [1, 3, 5]).
    ```

    dà luogo a un solo cammino di successo (in cui `Z` viene istanziata a `1`) a causa del taglio, quindi il predicato non può più essere usato per attraversare tutti gli elementi di una lista.
  - Predicato per implementare l'istruzione if-then-else (l'ordine tra le due regole non è importante):

    ```
    if_then_else(C, I1, _) :- C, !, I1.
    if_then_else(C, _, I2) :- not(C), !, I2.
    ```

  - Il predicato predefinito `not`, che ha successo sse il suo argomento fallisce, è definito come segue (sono rilevanti sia l'ordine tra `!` e `fail` nella prima regola che l'ordine tra le due regole):

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

    Essa equivale a:

    ```
    not(T) :- T, !, fail; true.
    ```

    dove il punto e virgola rappresenta una disgiunzione, ha meno priorità della virgola ed è comodo per accorpare in una sola più clausole aventi la stessa testa.

- Esempi relativi agli alberi binari (rappresentati attraverso un funtore `ab` dove il primo argomento è un numero intero associato alla radice, il secondo argomento è il sottoalbero sinistro e il terzo argomento è il sottoalbero destro; l'atomo `nil` rappresenta l'albero vuoto):
  - Predicato per stabilire se una sequenza di caratteri è un albero binario:
    ```
    albero_bin(nil).
    albero_bin(ab(N, Sx, Dx)) :- integer(N), albero_bin(Sx), albero_bin(Dx).
    ```
  - Predicato per stabilire se un elemento appartiene a un albero binario visitando quest'ultimo in ordine anticipato:
    ```
    cerca_albero_bin_ant(N, ab(N, _, _))  :- !.
    cerca_albero_bin_ant(N, ab(_, Sx, _)) :- cerca_albero_bin_ant(N, Sx).
    cerca_albero_bin_ant(N, ab(_, _, Dx)) :- cerca_albero_bin_ant(N, Dx).
    ```
  - Predicato per stabilire se un elemento appartiene a un albero binario visitando quest'ultimo in ordine posticipato:
    ```
    cerca_albero_bin_post(N, ab(_, Sx, _)) :- cerca_albero_bin_post(N, Sx).
    cerca_albero_bin_post(N, ab(_, _, Dx)) :- cerca_albero_bin_post(N, Dx).
    cerca_albero_bin_post(N, ab(N, _, _))  :- !.
    ```
  - Predicato per stabilire se un elemento appartiene a un albero binario visitando quest'ultimo in ordine simmetrico:
    ```
    cerca_albero_bin_simm(N, ab(_, Sx, _)) :- cerca_albero_bin_simm(N, Sx).
    cerca_albero_bin_simm(N, ab(N, _, _))  :- !.
    cerca_albero_bin_simm(N, ab(_, _, Dx)) :- cerca_albero_bin_simm(N, Dx).
    ```
  - Predicato per stabilire se una sequenza di caratteri è un albero binario di ricerca:
    ```
    albero_bin_ric(nil).
    albero_bin_ric(ab(N, Sx, Dx)) :- albero_bin(ab(N, Sx, Dx)),
                                     minore_ug(Sx, N), maggiore_ug(Dx, N).
    minore_ug(nil, _).
    minore_ug(ab(M, Sx, Dx), N) :- M =< N, minore_ug(Sx, N), minore_ug(Dx, N).
    maggiore_ug(nil, _).
    maggiore_ug(ab(M, Sx, Dx), N) :- M >= N, maggiore_ug(Sx, N), maggiore_ug(Dx, N).
    ```
  - Predicato per stabilire se un elemento appartiene a un albero binario di ricerca:
    ```
    cerca_albero_bin_ric(N, ab(N, _, _))  :- !.
    cerca_albero_bin_ric(N, ab(M, Sx, _)) :- N < M, cerca_albero_bin_ric(N, Sx).
    cerca_albero_bin_ric(N, ab(M, _, Dx)) :- N > M, cerca_albero_bin_ric(N, Dx).
    ```
- Il Prolog mette a disposizione dei predicati predefiniti per effettuare operazioni di input/output:
  - `read(T)` ha successo sse è possibile acquisire tramite tastiera una sequenza di caratteri che finisce con il punto (l'acquisizione termina premendo il tasto invio) e questa sequenza escluso il punto è unificabile con `T`.
  - `write(T)` ha successo stampando `T` sullo schermo.
  - `nl` ha successo stampando un'andata a capo sullo schermo.
- Esempi:
  - Predicato per stampare un valore e andare a capo sullo schermo:
    ```
    write_nl(T) :- write(T), nl.
    ```
  - Predicato per risolvere il problema delle torri di Hanoi (dischi, partenza, arrivo, intermedia):
    ```
    hanoi(1, P, A, _) :- stampa_mossa(P, A).
    hanoi(N, P, A, I) :- N > 1, N1 is N - 1,
                         hanoi(N1, P, I, A), stampa_mossa(P, A), hanoi(N1, I, A, P).
    stampa_mossa(P, A) :- write('sposta da '), write(P), write(' a '), write_nl(A).
    ```

- Il Prolog mette a disposizione dei predicati predefiniti di secondo ordine per raccogliere tutte le soluzioni di un problema rappresentato da un obiettivo (anziché calcolarle una alla volta):
  - `findall(T, O, L)` ha successo sse `L` è unificabile con la lista con eventuali ripetizioni di tutte le istanze del termine `T` per le quali l'obiettivo `O` ha successo.
  - `bagof(T, O, L)` ha successo sse `L` è unificabile con la lista non vuota con eventuali ripetizioni di tutte le istanze del termine `T` per le quali l'obiettivo `O` ha successo, dove le istanze sono organizzate in base alle variabili eventualmente presenti in `O`.
  - `setof(T, O, L)` ha successo sse `L` è unificabile con la lista non vuota senza ripetizioni di tutte le istanze del termine `T` per le quali l'obiettivo `O` ha successo, dove le istanze sono organizzate in base alle variabili eventualmente presenti in `O`.
- Esempi relativi ai grafi diretti (rappresentati attraverso un funtore `gd` dove il primo argomento è una lista senza ripetizioni di atomi che denotano i vertici mentre il secondo argomento è una lista senza ripetizioni di termini basati sul funtore `arco`, i cui due argomenti sono rispettivamente il vertice da cui l'arco esce e il vertice in cui l'arco entra):
  - Predicato per stabilire se una sequenza di caratteri è un grafo diretto:

```
grafo_dir(gd(LV, LA)) :- controlla_v(LV), controlla_a(LA, LV).
controlla_v([]).
controlla_v([V | LV]) :- not(membro(V, LV)), controlla_v(LV).
controlla_a([], _).
controlla_a([arco(V1, V2) | LA], LV) :- membro(V1, LV), membro(V2, LV),
                                        not(membro(arco(V1, V2), LA)),
                                        controlla_a(LA, LV).
```

  - Predicato per stabilire se un elemento appartiene a un grafo diretto visitando quest'ultimo in ampiezza a partire da un certo vertice iniziale (il secondo e il quarto argomento di `cerca_amp` rappresentano rispettivamente la lista dei vertici da visitare e la lista dei vertici già visitati):

```
cerca_grafo_dir_amp(V, gd(LV, LA), I) :- membro(I, LV), cerca_amp(V, [I], LA, []).
cerca_amp(V, [V | _], _, _)        :- !.
cerca_amp(V, [V1 | LV], LA, LVVis) :- V \== V1, membro(V1, LVVis),
                                      cerca_amp(V, LV, LA, LVVis).
cerca_amp(V, [V1 | LV], LA, LVVis) :- V \== V1, not(membro(V1, LVVis)),
                                      findall(V2, membro(arco(V1, V2), LA), LVAdiac),
                                      concatena(LV, LVAdiac, NuovaLV),
                                      cerca_amp(V, NuovaLV, LA, [V1 | LVVis]).
```

  - Predicato per stabilire se un elemento appartiene a un grafo diretto visitando quest'ultimo in profondità a partire da un certo vertice iniziale (il secondo e il quarto argomento di `cerca_prof` rappresentano rispettivamente la lista dei vertici da visitare e la lista dei vertici già visitati):

```
cerca_grafo_dir_prof(V, gd(LV, LA), I) :- membro(I, LV), cerca_prof(V, [I], LA, []).
cerca_prof(V, [V | _], _, _)        :- !.
cerca_prof(V, [V1 | LV], LA, LVVis) :- V \== V1, membro(V1, LVVis),
                                       cerca_prof(V, LV, LA, LVVis).
cerca_prof(V, [V1 | LV], LA, LVVis) :- V \== V1, not(membro(V1, LVVis)),
                                       findall(V2, membro(arco(V1, V2), LA), LVAdiac),
                                       concatena(LVAdiac, LV, NuovaLV),
                                       cerca_prof(V, NuovaLV, LA, [V1 | LVVis]).
```

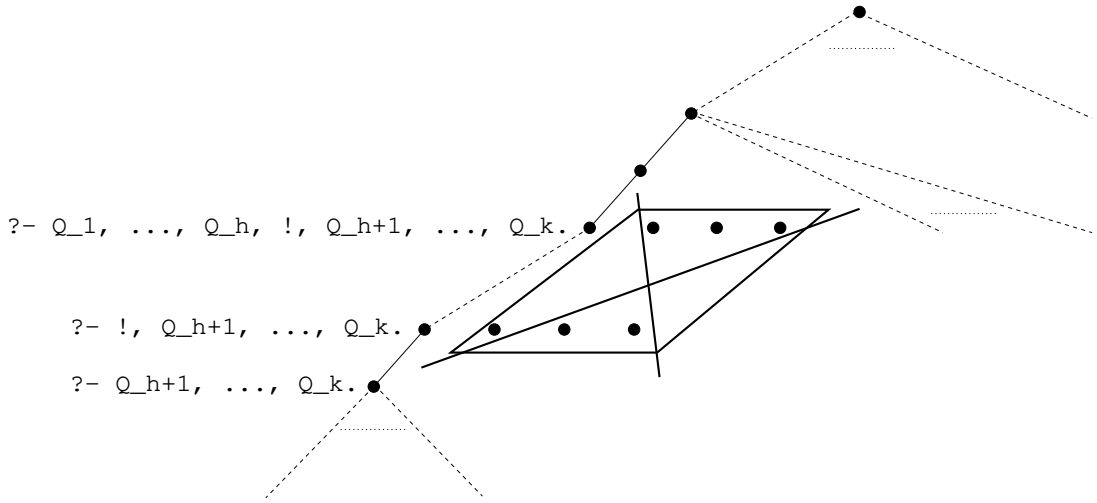


Figura 8.1: Effetto del taglio sull'albero e-o

- Il Prolog mette a disposizione dei predicati predefiniti per leggere, aggiungere o eliminare clausole di un programma così come per invocare obiettivi, grazie al fatto che le clausole e gli obiettivi stessi sono visti come termini in cui simboli come `:-` e la virgola sono visti come predicati in notazione infissa (in Prolog non c'è distinzione tra istruzioni e dati):
  - `clause(H, B)` ha successo sse il programma contiene una clausola la cui testa è unificabile con `H` e il cui corpo è unificabile con `B` (il corpo è `true` nel caso di un fatto).
  - `asserta(C)` aggiunge la clausola `C` all'inizio del programma.
  - `assertz(C)` aggiunge la clausola `C` alla fine del programma.
  - `retract(C)` ha successo sse il programma contiene una clausola unificabile con `C` e in tal caso elimina dal programma la prima clausola unificabile con `C`.
  - `retractall(H)` ha successo sse il programma contiene una clausola la cui testa è unificabile con `H` e in tal caso elimina dal programma tutte le clausole la cui testa è unificabile con `H`.
  - `call(G)` invoca `G` come obiettivo corrente e ha successo sse `G` ha successo.

  Questi predicati sono utilizzabili sia all'interno dei programmi (i quali possono quindi autoispezionarsi e automodificarsi) che al volo dentro gli obiettivi. Nel primo caso, le aggiunte o eliminazioni che essi determinano hanno luogo a tempo di esecuzione nell'interprete in cui i programmi sono stati caricati, non nei sorgenti dei programmi.
- Un metainterprete è un interprete scritto nello stesso linguaggio nel quale sono scritti i programmi da interpretare. Grazie ai predicati predefiniti per accedere alle clausole, è piuttosto semplice scrivere metainterpreti in Prolog:
  - Metainterprete per il Prolog puro:

```
raggiungi(true).
raggiungi((G1, G2)) :- raggiungi(G1), raggiungi(G2).
raggiungi(G)        :- clause(G, B), raggiungi(B).
```

  - Metainterprete per il Prolog puro esteso con taglio e negazione:

```
raggiungi(true).
raggiungi((G1, G2)) :- raggiungi(G1), raggiungi(G2).
raggiungi(!)        :- !.
raggiungi(not(G))   :- not(raggiungi(G)).
raggiungi(G)        :- clause(G, B), raggiungi(B).
```

■fltplf_20

Capitolo 9

Attività di Laboratorio in Linux

9.1 Il Compilatore/Interprete `ghci`

- Un programma Haskell viene scritto con un editor come `gvim`, viene memorizzato in un file sorgente con estensione `.hs` e viene eseguito attraverso un interprete Haskell.
- Comando per lanciare in esecuzione l'interprete Haskell:
  `ghci`
- L'interprete Haskell consente all'utente di usare funzioni predefinite e di caricare, eseguire e modificare programmi Haskell mediante un ciclo leggi-valuta-stampa in cui l'interprete:
  - Aspetta di acquisire da tastiera un identificatore e gli argomenti a cui esso deve essere applicato (l'acquisizione termina premendo il tasto invio). L'espressione complessiva deve essere preceduta da `:type` qualora l'utente voglia inferire il tipo dell'espressione anziché calcolare il suo valore.
  - Applica l'identificatore a quegli argomenti. Nel caso di funzioni definite in un file sorgente Haskell, quest'ultimo deve essere stato preventivamente caricato nell'interprete col comando:
    `:load` ◁*file sorgente*▷
    passando così dal modulo `Prelude` al modulo `Main` o al modulo comunque definito nel file sorgente.
  - Stampa sullo schermo il valore o il tipo dell'espressione oppure un messaggio d'errore in caso di problemi lessicali, sintattici o semantici.
- Usare `:help` per ottenere informazioni sui comandi dell'interprete e `<ctrl>d` per uscire dall'interprete.
- Comando per compilare un programma Haskell:
  `ghc` ◁*file sorgente*▷ `-o` ◁*file eseguibile*▷
  nel qual caso il file sorgente deve contenere l'azione `main`.
- Comando per lanciare in esecuzione il file eseguibile di un programma Haskell:
  `./`◁*file eseguibile*▷
  poi sulle linee successive bisogna introdurre i dati di ingresso del programma e dopo premere il tasto invio oppure `<ctrl>d`, a seconda dell'azione di input, per ottenere il risultato dell'esecuzione.

9.2 Implementazione e Modifica di Programmi Haskell

- Esercizi: Per ognuno dei seguenti esempi introdotti durante le lezioni teoriche, scrivere il file sorgente usando `gvim` e poi caricarlo in `ghci` al fine di testarne il corretto funzionamento:

  1. Programma contenente le funzioni relative al calcolo del fattoriale e di Fibonacci (Sez. 4.4).
  2. Programma contenente le funzioni relative alle liste (Sez. 4.4).
  3. Programma contenente le funzioni relative agli algoritmi di ordinamento (Sez. 4.4).
  4. Programma contenente le funzioni relative al problema delle torri di Hanoi (Sez. 4.6).
  5. Programma contenente le funzioni relative agli alberi binari (Sez. 4.6).
  6. Programma contenente le funzioni relative ai grafi diretti (Sez. 4.6). ■fegplf_1_2

9.3 Il Compilatore/Interprete `gprolog`

- Un programma Prolog viene scritto con un editor come `gvim`, viene memorizzato in un file sorgente con estensione `.pl` o `.pro` e viene eseguito attraverso un interprete Prolog.

- Comando per lanciare in esecuzione l'interprete Prolog:
  `gprolog`

- L'interprete Prolog consente all'utente di fare interrogazioni mediante predicati predefiniti e di caricare, eseguire e modificare programmi Prolog tramite un ciclo leggi-valuta-stampa in cui l'interprete:

  – Aspetta di acquisire da tastiera un obiettivo dopo aver visualizzato `?-` sullo schermo, dove l'obiettivo finisce con il punto (l'acquisizione termina premendo il tasto invio).

  – Tenta di raggiungere l'obiettivo sulla base dei predicati predefiniti e di eventuali predicati definiti in un file sorgente Prolog, che deve essere stato preventivamente caricato nell'interprete utilizzando l'obiettivo:
  `consult('◁`*file sorgente*`▷').`

  – Riporta l'esito dell'esecuzione (che può anche essere un messaggio d'errore in caso di problemi lessicali, sintattici o semantici):

    * Se l'obiettivo non può essere raggiunto, l'interprete stampa sullo schermo `no`.
    * Se l'obiettivo può essere raggiunto ed è ground, l'interprete stampa sullo schermo `yes` o `true`.
    * Se l'obiettivo può essere raggiunto ma non è ground, l'interprete stampa sullo schermo anche i legami creati tramite unificazione per le variabili non istanziate presenti nell'obiettivo.

    Negli ultimi due casi, dopo aver trovato quella soluzione l'interprete stampa sullo schermo `?` e aspetta di acquisire da tastiera una decisione (premere `h` per vedere le opzioni disponibili):

    * Se l'utente preme il tasto invio, l'esecuzione termina.
    * Se l'utente preme `;`, l'esecuzione continua in cerca di un'altra soluzione.
    * Se l'utente preme `a`, l'esecuzione continua in cerca di tutte le altre soluzioni.

- Usare `<ctrl>c` seguito da `h` per ottenere informazioni sui comandi dell'interprete e `<ctrl>d` per uscire dall'interprete.

- Comando per compilare un programma Prolog:
  `gplc ◁`*file sorgente*`▷ -o ◁`*file eseguibile*`▷`

- Comando per lanciare in esecuzione il file eseguibile di un programma Prolog:
  `./◁`*file eseguibile*`▷`
  che equivale a lanciare in esecuzione l'interprete Prolog esteso con i predicati definiti in quel programma.

9.4 Implementazione e Modifica di Programmi Prolog

- Esercizi: Per ognuno dei seguenti esempi introdotti durante le lezioni teoriche, scrivere il file sorgente usando `gvim` e poi caricarlo in `gprolog` al fine di testarne il corretto funzionamento:

  1. Programma contenente i predicati relativi al calcolo del fattoriale e di Fibonacci (Sez. 8.3).
  2. Programma contenente i predicati relativi alle liste (Sez. 8.3).
  3. Programma contenente i predicati relativi agli algoritmi di ordinamento (Sez. 8.3).
  4. Programma contenente i predicati relativi all'algoritmo di unificazione di Robinson (Sez. 8.3).
  5. Programma contenente i predicati relativi al problema delle torri di Hanoi (Sez. 8.4).
  6. Programma contenente i predicati relativi agli alberi binari (Sez. 8.4).
  7. Programma contenente i predicati relativi ai grafi diretti (Sez. 8.4). ■fegplf_3_4