Mathematical Logic

Progetto

Luca Milanesi 55053A
luca.milanesi1@studenti.unimi.it

2024-2025

Indice

1 Descrizione generale del progetto

Questo progetto si concentrerà sui capitoli 4, 5 e 6 rispetto alle dispense *"Risoluzione e Sovrapposizione"* di Silvio Ghilardi [1], che si incentrano su tre tipologie distinte di calcolo per la refutazione di un insieme di clausole della logica del primo ordine:

1. **Calcolo R**
2. **Risoluzione Ordinata**
3. **Calcolo S**

Obiettivo L'obiettivo del progetto è implementare in Java questi tre metodi di calcolo, seguendo le regole e le metodologie descritte nelle dispense.

Struttura del Progetto L'architettura del progetto è suddivisa in tre macro-sezioni:

1. **Strutture logiche**: contiene le classi che rappresentano le strutture logiche su cui verranno eseguiti i calcoli.
2. **Implementazione dei calcoli**: comprende le classi principali responsabili dell'implementazione dei tre metodi di calcolo elencati.
3. **Utility e supporto**: raccoglie le classi di supporto ai calcoli principali, tra cui quelle che implementano operazioni secondarie o ausiliarie necessarie per il corretto funzionamento dei calcoli principali.

Codice sorgente L'intero codice sorgente del progetto è disponibile nella repository GitHub personale [2].

2 Strutture logiche

Le strutture logiche sono le componenti principali che verranno manipolate per la risoluzione. Ogni struttura logica fornirà un parser per poter creare la struttura finale concreta partendo da una stringa di input.

Inoltre, ogni struttura logica implementa delle interfacce comuni che definiscono determinate operazioni, tra le quali:

- `LogicalStructure` che definisce dei metodi per collezionare tutti i simboli di una determinata struttura logica e per poter ritornare la struttura logica presa in considerazione applicando una funzione di sostituzione sui suoi argomenti (utile poi dal punto 3.1).
- `Copyable` che definisce un metodo per poter ritornare una copia della struttura logica presa in considerazione.

2.1 Termini

I termini dell'insieme sono identificati dalla classe `Term`. Questa classe rappresenta un termine dell'insieme $T_{\mathcal{L}}$ del linguaggio $\mathcal{L}$, che può essere:

- Una variabile contenuta nell'insieme delle variabili Var identificata da una lettera o una semplice stringa preceduta da un punto interrogativo, ad esempio $?x, ?y, ?z, ?var$.
- Una costante dell'insieme $\mathcal{F}$ di arietà nulla identificata da una lettera o una semplice stringa, ad esempio a, b, c, foo.
- Una funzione con argomenti identificata da $f(t_1, ..., t_n)$ dove $f \in \mathcal{F}$ è una funzione di arietà $n > 0$ mentre t_i per ogni $i \in \{1, ..., n\}$ è un termine dell'insieme $T_{\mathcal{L}}$.

È composta da un nome che può indicare una variabile o un simbolo di funzione, e una lista di termini che saranno gli argomenti della funzione (nel caso di una variabile o una costante la lista sarà vuota).

La classe fornisce metodi per determinare se un termine è una variabile, una costante o una funzione, per verificare se un termine è contenuto in un altro e un metodo per poter sostituire un argomento della lista con un altro termine (utile poi al punto 5.2).

2.2 Letterali

I letterali sono identificati dalla classe `Literal`. Questa classe rappresenta un letterale, ovvero una formula atomica del tipo:

$$P(t_1, ..., t_n)$$

con $P \in \mathcal{P}$ e $\forall i \in \{1, ..., n\} : t_i \in T_{\mathcal{L}}$, che può essere negata o non negata, dove P è un predicato di arietà $n \geq 0$ e tutti i t_i sono i termini del predicato. Per definire un letterale negativo viene usato il simbolo di negazione $\neg$ prima del predicato.

È composta da una flag booleana per poter indicare se il predicato è negato o meno, il simbolo del predicato e una lista di termini che saranno gli argomenti del predicato.

La classe offre metodi per negare un letterale, ottenere la vista del multiinsieme dei termini (utile poi nel punto 4.2 per determinare se un letterale è massimale in una clausola) e un metodo per poter trasformare il letterale corrente con il predicato d'identità, se non lo è già (utile poi al punto 5.2).

2.3 Clausole

Le clausole sono identificati dalla classe `Clause`. Questa classe rappresenta una clausola, che è una disgiunzione di letterali rappresentata dalla vista:

$$A_1, ..., A_n \Rightarrow B_1, ..., B_m$$

con $\forall i \in \{1, ..., n\}, j \in \{1, ..., m\} : A_i, B_j \in \mathcal{P}$, dove gli A_i rappresentano i letterali negativi e B_j quelli positivi. È composta quindi da due insiemi di letterali: uno per i letterali negati e uno per i letterali positivi.

La classe fornisce metodi per determinare se una clausola è vuota o è una tautologia, per confrontare le clausole in base al numero di simboli e un metodo per poter trasformare tutti i letterali della clausola corrente con il predicato d'identità, se non lo sono già. Inoltre, verranno identificati anche i letterali positivi e negativi massimali durante la costruzione della clausola (che serviranno poi dal punto 4.3 in avanti).

3 Risoluzione

In questo capitolo vengono descritte le componenti implementate per poter implementare il calcolo della Risoluzione su linguaggi senza identità. Inoltre, verrà descritta la struttura e l'interfaccia del dimostratore automatico.

3.1 Unificazione

L'unificazione e il matching sono processi fondamentali per la refutazione di un insieme di clausole. Questi due processi hanno scopo di trovare una sostituzione:

$$\sigma : Var \to T_{\mathcal{L}}$$

che ha come dominio le variabili e codominio i termini del linguaggio $\mathcal{L}$, che se applicata a due letterali li rende uguali (nel caso dell'unificazione la sostituzione si applica a entrambi i letterali, mentre per il matching solo ad uno di essi).

La classe `Unification` è responsabile del processo di unificazione e matching tra letterali. Essa fornisce metodi per eseguire l'unificazione o il matching tra due letterali, oltre che per la verifica della correttezza delle sostituzioni trovate.

I metodi `unify` e `match` implementano gli algoritmi di unificazione e matching tra due letterali, seguendo una serie di regole per trasformare e semplificare le equazioni derivate dai termini dei letterali identificate dalla classe `Equation` (che definisce una equazione del tipo $t \stackrel{?}{=} u$ con $t, u \in T_{\mathcal{L}}$), al fine di trovare una sostituzione che li unifichi o li faccia corrispondere. Le regole del ciclo di istruzioni sono le seguenti:

1. Cancellare le equazioni del tipo $t \stackrel{?}{=} t$.
2. Sostituire un'equazione del tipo $f(t_1, ..., t_n) \stackrel{?}{=} f(u_1, ..., u_n)$ con le n equazioni $t_1 \stackrel{?}{=} u_1, ..., t_n \stackrel{?}{=} u_n$.
3. Sostituire un'equazione del tipo $t \stackrel{?}{=} ?x$ con $?x \stackrel{?}{=} t$, qualora $t \notin Var$.
4. Sostituire, qualora si incontri un'equazione del tipo $?x \stackrel{?}{=} t$ con t che non contiene occorrenze di $?x$, tutte le occorrenze di $?x$ nelle rimanenti equazioni con t.
5. Terminare in stato di fallimento qualora si incontri un'equazione del tipo $f(t_1, ..., t_n) \stackrel{?}{=} g(u_1, ..., u_m)$ con $f \not\equiv g$.
6. Terminare in stato di fallimento qualora si incontri un'equazione del tipo $x \stackrel{?}{=} t$ con t che è diverso da x ma contiene un'occorrenza di x.

L'operazione di matching è trattata come l'operazione di unificazione considerando le variabili dei membri destri come fossero delle costanti.

3.2 Calcolo R

Le regole di inferenza del **calcolo R** sono state implementate nella classe `CalculusR`. In questa classe sono quindi presenti due metodi rispetto alle due regole di inferenza del **calcolo R**:

- Un metodo per poter applicare la **Regola di Risoluzione**. Dati in input un letterale positivo A, un letterale negativo B e due clausole del tipo $\Gamma \Rightarrow \Delta, A$, $\Gamma', B \Rightarrow \Delta'$, ritorna, se possibile, la clausola risolta

  $$\Gamma\mu, \Gamma'\mu \Rightarrow \Delta\mu, \Delta'\mu$$

  dove μ è l'unificatore più generale del problema di unificazione $A \stackrel{?}{=} B$.

- Un metodo per poter applicare la **Regola di Fattorizzazione Destra**. Dati in input due letterali positivi A, B e una clausola del tipo $\Gamma \Rightarrow \Delta, A, B$, ritorna, se possibile, la clausola fattorizzata

  $$\Gamma\mu \Rightarrow \Delta\mu, A\mu$$

  dove μ è l'unificatore più generale del problema di unificazione $A \stackrel{?}{=} B$.

Questa classe implementa anche dei metodi che verranno poi estesi e sovrascritti dall'implementazione della **Risoluzione Ordinata** al punto 4.3. Questi metodi sono:

- Dei metodi che, data in input una clausola, ritornano l'insieme di letterali sulla quale poter applicare le regole di fattorizzazione e risoluzione. Nel caso del **calcolo R**, non ci saranno restrizioni sugli insiemi di letterali delle clausole che si utilizzeranno.

- Dei metodi che restituiscono se le regole di fattorizzazione e risoluzione possono essere applicate secondo determinate regole rispetto al tipo di calcolo adottato. Nel caso del **calcolo R**, se possibile, vengono sempre applicate senza eseguire nessun controllo a posteriori.

Per poter ridurre lo spazio di ricerca della refutazione nel **calcolo R**, sono stati implementati nella classe `Reduction` i metodi per l'applicazione delle regole di riduzione su insiemi di clausole, che in questo caso saranno:

- Rimozione delle **tautologie** da un insieme di clausole.
- Rimozione delle clausole **sussunte**, per la quale è stata creata una classe apposita `Subsumption`, che fornisce le operazioni necessarie a capire se una data clausola è sussunta da una seconda clausola. La clausola $\Gamma \Rightarrow \Delta$ sussume la clausola $\Gamma' \Rightarrow \Delta'$ se e solo se per un matcher σ abbiamo $\Gamma\sigma \subseteq \Gamma'$ e $\Delta\sigma \subseteq \Delta'$.
- Rimozione delle clausole tramite **Matching Replacement Resolution**, per la quale è stata creata una classe apposita `MatchingReplacementResolution` che implementa la logica necessaria. Date due clausole già dedotte $\Gamma_1 \Rightarrow \Delta_1, A_1$, $\Gamma_2, A_2 \Rightarrow \Delta_2$ supponiamo anche che esista un matcher σ soddisfacente le seguenti condizioni:

  1. $A_1\sigma \equiv A_2$
  2. $\Gamma_1\sigma \subseteq \Gamma_2$
  3. $\Delta_1\sigma \subseteq \Delta_2$

  In tali condizioni possiamo cancellare la clausola $\Gamma_2, A_2 \Rightarrow \Delta_2$ e sostituirla con la clausola più semplice $\Gamma_2 \Rightarrow \Delta_2$.

Inoltre, la classe estende l'interfaccia comune per un dimostratore automatico definita dalla classe astratta `AutomaticCalculus`.

3.3 Struttura del dimostratore automatico

La classe astratta `AutomaticCalculus` implementa un'interfaccia comune per un dimostratore automatico. Questa classe segue un processo iterativo di saturazione delle clausole, garantendo che tutte le regole di inferenza vengano applicate efficentemenete.

È composta dai due insiemi di clausole *Us* (usable), che all'inizio contiene tutte le clausole inter-ridotte da refutare, e *Wo* (worked), inizialmente vuoto.

Il metodo principale di questa classe è `refute`, che date in input un insieme di clausole, prova a refutarle, restituendo un dato positivo nel caso in cui si arrivi ad una refutazione, oppure un dato negativo in caso contrario[1].

Questo metodo implementa le istruzioni del ciclo di risoluzione di un dimostratore automatico globale descritto di seguito:

[1]Non è detto che il metodo termini sempre. Infatti, se un insieme di clausole è soddisfacibile, potrebbe portare il calcolo refutazionale in un loop infinito.

1. Si sceglie all'interno di *Us* una clausola, detta clausola data, che viene aggiunta all'insieme *Wo*. Questa clausola sarà la clausola con il minor numero di simboli che la compongono.

2. Si applicano tutte le inferenze possibili fra la clausola data, se stessa e le clausole in *Wo*, tramite i metodi astratti opportuni (che saranno poi implementati concretamente nelle classi dei risolutori concreti rispetto alla tipologia di calcolo che si vuole attuare), generando un insieme *New* di clausole nuove. Durante questa fase, per ogni coppia di clausole da inferire, viene applicata, prima dell'inferenza, la rinominazione per assicurarsi che le due clausole abbiano variabili disgiunte tra loro (ciò è possibile grazie alla classe `Renaming` che implementa questa logica).

3. Si applica la riduzione in avanti per semplificare l'insieme *New* tramite le clausole in *New* stesse, tramite *Wo* ed infine tramite *Us*.

4. Si applica la riduzione all'indietro per semplificare le clausole in *Wo* e *Us* tramite le clausole in *New*.

5. Si aggiungono le clausole in *New* alle clausole in *Us*, "svuotando" l'insieme *New*.

6. Se *Us* è vuoto, l'insieme di clausole è soddisfacibile, e quindi si restituisce un dato negativo, altrimenti si torna al punto 1.

I metodi di riduzione delle clausole iniziali, della riduzione in avanti e della riduzione all'indietro saranno dichiarati all'interno di `AutomaticCalculus` per poi essere implementati concretamente nelle classi finali che estenderanno questa classe rispetto alla tipologia di calcolo adottata.

4 Ordinamenti

In questo capitolo verranno discusse le classi implementate per l'ordinamento dei termini, i letterali massimali e l'estensione del **calcolo R**, ovvero la **Risoluzione Ordinata**.

4.1 Ordinamento dei termini

Per l'ordinamento dei termini è stata implementata la classe `LpoComparator`, che altro non è che un comparatore di termini che implementa la logica di ordinamento **"Lexicographic Path Order"**. Questa classe estende l'interfaccia `Comparator` sulle classi `Term` ed è composta da un comparatore, che definisce la precedenza per l'ordinamento dei simboli dei termini (di default l'ordinamento dei simboli è alfabetico).

4.2 Letterali massimali in una clausola

Prima di iniziare, è da ricordare che per determinare i letterali massimali in una clausola verrà usata la vista del multiinsieme dei termini di un letterale.

Dunque, l'obiettivo primario è la contruzione di un comparatore di multiinsiemi di termini. Per fare ciò, è stata implementata la classe `MultisetComparator`, che appunto è un comparatore di multiinsiemi di termini che si appoggierà al comparatore `LpoComparator` spiegato al punto precedente.

Per semplicità, il problema di comparazione tra due multiinsiemi di termini è stato **ridotto** al problema di comparazione tra due multiinsiemi di interi, assegnando ad ogni termine dei multiinsiemi un intero tramite una funzione $\pi : T_{\mathcal{L}} \to \mathbb{N}$ data dall'ordine dei termini dei multiinsiemi, i termini maggiori saranno mappati ad interi maggiori e viceversa. Dati due multiinsiemi M e N, diciamo $M > N$ se riusciamo ad ottenere N da M rimpiazzando le occorrenze degli elementi di M con un multiinsieme di elementi tutti minori.

Inoltre, è presente la classe `MaximalLiteral` che offre dei metodi per poter poter ottenere l'insieme dei letterali massimali di una data clausola e per poter calcolare se un determinato letterale è (strettamente) massimale in una clausola. Nel caso di determinazione di massimalità **stretta** su un insieme di letterali, tutte le variabili vengono viste allo stesso modo senza distinzione e viene tenuta in considerazione la **riflessività** dei letterali con identità.

4.3 Risoluzione ordinata

La **Risoluzione Ordinata** è un'estensione del **calcolo R** descritto al punto 3.2. Infatti, per poter implementare questo tipo di calcolo, è stata implementata la classe `SortedCalculus` che estende la classe `CalculusR`. Questa classe sovrascrive determinati elementi del **calcolo R**, tra i quali:

- I metodi che, dati in input una clausola ritornano l'insieme di letterali sulla quale poter applicare le regole di fattorizzazione e risoluzione. In questa tipologia di calcolo verranno presi in considerazione solo i letterali massimali della clausola.
- I metodi che restituiscono se le regole di fattorizzazione e risoluzione possono essere applicate secondo determinate regole rispetto al tipo di calcolo adottato. In questa tipologia di calcolo, a differenza del **calcolo R**, i letterali della nuova clausola dedotta devono rispettare determinati vincoli di massimalità:
  - La **Regola di Risoluzione** si applica solo qualora il letterale $A\mu$ è strettamente massimale in $\Gamma\mu \Rightarrow \Delta\mu, A\mu$ e $B\mu$ è massimale in $\Gamma'\mu, B\mu \Rightarrow \Delta'\mu$, dove μ è l'unificatore più generale del problema di unificazione $A \stackrel{?}{=} B$.
  - La **Regola di Fattorizzazione Destra** si applica solo qualora il letterale $A\mu$ è massimale in $\Gamma\mu \Rightarrow \Delta\mu, A\mu, B\mu$, dove μ è l'unificatore più generale del problema di unificazione $A \stackrel{?}{=} B$.

5 Paramodulazione e Sovrapposizione

5.1 Alberi e Posizioni

I termini implementati dalla classe `Term`, strutturalmente, sono già rappresentati come alberi: il nodo radice è il nome (che può indicare una variabile o un simbolo di funzione) e i figli sono gli argomenti, anch'essi termini, che ricorsivamente comporranno un albero. Le posizioni, pur non essendo gestite esplicitamente, saranno identificate dall'indice del termine nella lista degli argomenti rispetto un determinato termine.

Ad esempio con il termine $f(?x, g(c, ?y))$ si avrà un primo oggetto `Term` identificato da s con:

$$s.name = f, s.arguments = [t_1, t_2]$$

dove ricorsivamente t_1 e t_2 saranno:

$$t_1.name = ?x, t_1.arguments = []$$

$$t_2.name = g, t_2.arguments = [h_1, h_2]$$

e infine, sempre ricorsivamente, h_1 e h_2 saranno:

$$h_1.name = ?c, h_1.arguments = []$$

$$h_2.name = ?y, h_2.arguments = []$$

5.2 Calcolo S

Come per il **calcolo R**, nella classe `CalculusS` sono stati implementati i metodi per l'applicazione delle regole di inferenza del **Calcolo S** che verranno applicate nel calcolo della risoluzione automatica, ovvero:

- Un metodo per poter applicare la **Regola di Sovrapposizione Destra**. Dati in input due letterali positivi $l = r, s = t$ e due clausole $\Gamma' \Rightarrow \Delta', l = r$, $\Gamma \Rightarrow \Delta, s = t$ ritorna, se possibile, la clausola

  $$\Gamma\mu, \Gamma'\mu \Rightarrow (s[r]_p)\mu = t\mu, \Delta\mu, \Delta'\mu$$

  dove $s_{|p}$ non è una variabile e μ è l'unificatore più generale del problema di unificazione $s_{|p} \stackrel{?}{=} l$ se e solo se:

  - $l\mu \not\leq r\mu$ e $s\mu \not\leq t\mu$

- $l\mu = r\mu$ è **strettamente** massimale in $\Gamma'\mu \Rightarrow \Delta'\mu, l\mu = r\mu$
- $s\mu = t\mu$ è **strettamente** massimale in $\Gamma\mu \Rightarrow \Delta\mu, s\mu = t\mu$

- Un metodo per poter applicare la **Regola di Sovrapposizione Sinistra**. Dati in input un letterale positivo $l = r$, uno negativo $s = t$ e due clausole $\Gamma' \Rightarrow \Delta', l = r$, $\Gamma, s = t \Rightarrow \Delta$ ritorna, se possibile, la clausola

$$\Gamma\mu, \Gamma'\mu, (s[r]_p)\mu = t\mu \Rightarrow \Delta\mu, \Delta'\mu$$

dove $s_{|p}$ non è una variabile e μ è l'unificatore più generale del problema di unificazione $s_{|p} \stackrel{?}{=} l$ se e solo se:

- $l\mu \not\leq r\mu$ e $s\mu \not\leq t\mu$
- $l\mu = r\mu$ è **strettamente** massimale in $\Gamma'\mu, \Rightarrow \Delta'\mu, l\mu = r\mu$
- $s\mu = t\mu$ è massimale in $\Gamma\mu, s\mu = t\mu \Rightarrow \Delta\mu$

- Un metodo per poter applicare la **Regola di Risoluzione per l'Uguaglianza**. Dati in input un letterale negativo $s = t$ e una clausola $\Gamma, s = t \Rightarrow \Delta$ ritorna, se possibile, la clausola

$$\Gamma\mu \Rightarrow \Delta\mu$$

dove μ è l'unificatore più generale del problema di unificazione $s \stackrel{?}{=} t$ se e solo se:

- $s\mu = t\mu$ è massimale in $\Gamma\mu, s\mu = t\mu \Rightarrow \Delta\mu$

Verrà tenuta in considerazione la simmetria dell'identità, tenendo in considerazione tutte le combinazioni tra gli argomenti.

- Un metodo per poter applicare la **Regola di Fattorizzazione per l'Uguaglianza**. Dati in input due letterali positivi $s = t$, $s' = t'$ e una clausola $\Gamma \Rightarrow s = t, s' = t', \Delta$ ritorna, se possibile, la clausola

$$\Gamma\mu, t\mu = t'\mu \Rightarrow s\mu = t'\mu, \Delta\mu$$

dove μ è l'unificatore più generale del problema di unificazione $s \stackrel{?}{=} s'$ se e solo se:

- $s\mu = t\mu$ è massimale in $\Gamma\mu \Rightarrow s\mu = t\mu, s'\mu = t'\mu, \Delta\mu$

Verrà tenuta in considerazione la simmetria dell'identità, tenendo in considerazione tutte le combinazioni tra gli argomenti.

Anche questa classe estende l'interfaccia comune per un dimostratore automatico definita dalla classe astratta `AutomaticCalculus`, che si occuperà di eseguire le operazioni standard per la risoluzione automatica utilizzando le regole di inferenza implementate nella classe concreta.

Nel calcolo S però, a differenza del calcolo R, al momento dell'inizializzazione dell'insieme di clausole da refutare, tutti i letterali delle clausole date in input vengono trasformate, per comodità, in letterali con identità, scrivendo le formule atomiche del tipo $P(t_1, ..., t_n)$ nella forma $P(t_1, ..., t_n) = true$, dove P viene considerato simbolo di funzione e $true$ una costante (che si suppone minima nell'ordinamento dei termini).

Riferimenti bibliografici

[1] Silvio Ghilardi. *Risoluzione e Sovrapposizione*. 2007. URL: https://aguzzoli.di.unimi.it/didattica/logica/secondaparte.pdf.

[2] Luca Milanesi. *Repository GitHub del progetto*. 2025. URL: https://github.com/Luca-02/math-logic-project.