UNIVERSITÀ DEGLI STUDI DI MILANO
Facoltà di Scienze e Tecnologie
Corso di Laurea in Informatica

IMPLEMENTAZIONE IN JAVA DEL METODO DI RISOLUZIONE PER LA LOGICA CLASSICA, ED ESTENSIONE A LOGICHE MODALI

Relatore: Prof. Camillo FIORENTINI

Tesi di:
Nicolò IACCARINO
Matricola: 903870

Anno Accademico 2023-2024

Prefazione

Questo elaborato riguarda l'ambito della logica matematica, concentrandosi sul metodo di risoluzione e sulla sua implementazione mediante il linguaggio di programmazione Java. Inizialmente, affronteremo il metodo di risoluzione per la logica classica, ossia la logica proposizionale; successivamente, verrà trattato il metodo di risoluzione per le logiche modali non-normali, che rappresentano un'estensione della logica proposizionale.

Indice

Capitolo 1

Introduzione

Iniziamo presentando una breve introduzione alla logica proposizionale, che espone i concetti utili a comprendere il funzionamento del metodo di risoluzione, trattato nel capitolo 2. L'introduzione alle logiche modali sarà illustrata nel capitolo 5.

1.1 Logica proposizionale

La logica proposizionale [May11] è un sistema formale per la rappresentazione e l'analisi del ragionamento. Essa si basa su proposizioni e la sua sintassi comprende l'utilizzo di formule. Una formula è un'espressione sintattica che rappresenta una proposizione. Le formule sono costruite utilizzando gli operatori (o connettivi) logici e le variabili proposizionali, e sono utilizzate per rappresentare relazioni, proprietà e regole all'interno di un sistema formale. Nella sezione 1.1.1 è presente la definizione formale di *formula*. Le formule possono essere *atomiche* o *composte*.

Formule atomiche

Le formule atomiche rappresentano il caso più semplice di formula, in cui non vengono usati i connettivi logici. Una formula atomica è una variabile proposizionale, che può essere identificata tramite una lettera dell'alfabeto, ad esempio "p"; essa viene valutata come vera (*true*) o falsa (*false*) in base all'interpretazione considerata.

Definizione 1.1.1. *Sia P un insieme di variabili proposizionali. Un'**interpretazione** (o assegnamento) I per P è una funzione:*

$$I: \quad P \mapsto \{true, false\}$$

che assegna un valore di verità a ogni variabile proposizionale appartenente a P.

Esempio: La variabile p può rappresentare la proposizione "*piove*", ed essere interpretata come vera se effettivamente piove, o falsa in caso contrario.

Formule composte

Le formule composte sono ottenute mediante i connettivi logici a partire dalle formule atomiche. I connettivi della logica proposizionale sono:

- $\neg$ è la negazione logica (**not**)
- $\wedge$ è la congiunzione logica (**and**)
- $\vee$ è la disgiunzione logica (**or**)
- $\rightarrow$ è l'implicazione logica (**implica**)
- $\leftrightarrow$ è la doppia implicazione (**se e solo se**)

1.1.1 Definizione formale di formula

Ora abbiamo tutto l'occorrente per dare una definizione formale di *formula*, e della sua interpretazione.

Definizione 1.1.2. *Sia P un insieme di variabili proposizionali. L'insieme delle formule ottenute da P è definito induttivamente:*

- *ogni variabile proposizionale in P è una formula.*
- *se A è una formula, allora anche $\neg A$ è una formula.*
- *se A e B sono formule, allora anche $(A \wedge B)$, $(A \vee B)$, $(A \rightarrow B)$, $(A \leftrightarrow B)$ sono formule.*
- *nient'altro è una formula.*

Esempio: Consideriamo la seguente formula:

$$\neg p \wedge q$$

essa è una formula composta ottenuta a partire dall'atomica p che viene negata tramite la negazione logica ($\neg$), e successivamente messa in congiunzione logica ($\wedge$) con l'atomica q.

Nelle formule più complesse, si usano le parentesi tonde per evitare ambiguità tra i connettivi.

Interpretazione di formule

L'interpretazione di una formula dipende dall'interpretazione delle variabili proposizionali nell'insieme P che occorrono in essa. Una volta fissata un'interpretazione I per P, risulta comunque stabilita l'interpretazione di qualsiasi formula in base al significato delle lettere proposizionali che occorrono in essa e al significato dei connettivi logici. La funzione di interpretazione I che abbiamo visto per le variabili proposizionali si può estendere alle formule.

Definizione 1.1.3. *Sia L un insieme di formule. Un'**interpretazione** I per L è una funzione:*

$$I : \quad L \mapsto \{true, false\}$$

tale che, per ogni coppia di formule $A, B \in L$ valgono le seguenti condizioni:

- $I(\neg A) = true$*, se* $I(A) = false$
- $I(\neg A) = false$*, se* $I(A) = true$
- $I(A \wedge B) = true$*, se e solo se* $I(A) = true$ ***e*** $I(B) = true$
- $I(A \vee B) = true$*, se e solo se* $I(A) = true$ ***oppure*** $I(B) = true$
- $I(A \rightarrow B) = true$*, se e solo se* $I(A) = false$ ***oppure*** $I(B) = true$
- $I(A \leftrightarrow B) = true$*, se e solo se* $I(A) = true$ ***e*** $I(B) = true$*,* ***oppure*** $I(A) = false$ ***e*** $I(B) = false$

Questa definizione stabilisce il significato dei connettivi logici.

1.1.2 Soddisfacibilità di una formula

Definizione 1.1.4. *Una formula F si dice **soddisfacibile** se e solo se esiste almeno un'interpretazione che la soddisfa, ossia esiste almeno un'interpretazione I tale che $I(F) = true$. Se non esiste alcuna interpretazione del genere, allora F si dice **insoddisfacibile**.*

Esempio: la formula $p \wedge q$ è soddisfacibile, perché l'interpretazione I che assegna i valori $I(p) = true$ e $I(q) = true$ la soddisfa.

1.1.3 Tautologie e contraddizioni

Definizione 1.1.5. *Una formula F è una **tautologia** se e solo se ogni interpretazione di F la soddisfa.*

Quindi una tautologia è una formula sempre vera, che come vedremo, può essere ridondante in alcuni casi.

Esempio: la formula

$$p \vee \neg p$$

è il caso più semplice di tautologia, perché qualunque sia l'interpretazione che diamo a p, la formula risulta essere sempre vera.

Definizione 1.1.6. *Una formula è una* ***contraddizione*** *se e solo se è insoddisfacibile.*

Dunque, una contraddizione è una formula nella quale tutte le sue interpretazioni la rendono falsa. È bene notare che se si applica la negazione logica a una contraddizione si ottiene una tautologia, e viceversa.

Esempio: la formula

$$p \wedge \neg p$$

è una contraddizione, perché non è possibile renderla vera.

Per ulteriori approfondimenti sulla logica proposizionale si consulti [May11].

Capitolo 2

Il metodo di risoluzione per la logica classica

Il metodo di risoluzione è un sistema di calcolo logico per inferire la soddisfacibilità di una formula, esso ha avuto un impatto significativo in vari settori della matematica, dell'informatica e dell'ingegneria. È stato utilizzato per dimostrare teoremi importanti, risolvere problemi pratici e sviluppare algoritmi per l'intelligenza artificiale, la verifica formale e la progettazione dei circuiti [Lei97].

Questo metodo consiste nell'applicare (una o più volte) una sola regola: la regola di risoluzione. Il difetto è che essa opera soltanto su formule espresse in **Forma normale congiuntiva (CNF)**. Prima di vedere il metodo di risoluzione è bene capire in che cosa consiste la *CNF*.

2.1 Forma normale congiuntiva

2.1.1 Letterali

Definizione 2.1.1. *Un letterale è una formula atomica, oppure la sua negazione. Un letterale può quindi essere positivo o negativo.*

Esempio: "a" è un letterale positivo, "$\neg b$" è un letterale negativo.

Opposto di un letterale

Definizione 2.1.2. *Dato un letterale L, il suo* ***opposto*** *(o* ***complemento****) $\overline{L}$ è la sua negazione.*

Esempio: se $L = p$ allora $\overline{L} = \neg p$ e se $L = \neg p$ allora $\overline{L} = p$.

2.1.2 Clausole

Definizione 2.1.3. *Una clausola è una disgiunzione di uno o più letterali.*

Consideriamo il seguente esempio:

$$a \vee \neg b \vee \neg c \vee d$$

questa è una clausola formata dai letterali "a", "$\neg b$", "$\neg c$", "d".

Una clausola può anche essere rappresentata in notazione insiemistica, in questo modo diventa un insieme di letterali. La clausola dell'esempio precedente diventa:

$$\{a, \neg b, \neg c, d\}$$

che come si può notare, il simbolo di disgiunzione logica non è più presente, ma è sottinteso. D'ora in avanti useremo sempre la notazione insiemistica per rappresentare le clausole.

L'interpretazione delle clausole è semplice: una clausola è vera se e solo se almeno un letterale appartenente ad essa è vero in una data interpretazione (a causa della disgiunzione).

Clausola tautologica

Definizione 2.1.4. *Una clausola è tautologica se e solo se contiene almeno un letterale ed il suo opposto.*

Esempio: "$\{a, b, c, \neg a\}$" è una clausola tautologica, perché contiene il letterale a ed il suo opposto. Questo tipo di clausola risulta essere sempre vera qualunque sia l'interpretazione dei suoi letterali, quindi essa rappresenta una tautologia (si veda la sezione 1.1.3).

Clausola vuota

è importante notare che una clausola può non contenere alcun letterale, in tal caso si parla di *clausola vuota*. La clausola vuota rappresenta la **contraddizione** (si veda la sezione 1.1.3) e si può indicare con "$\{\}$".

2.1.3 Insiemi di clausole

Definizione 2.1.5. *La Forma Normale Congiuntiva (CNF) è una forma standard delle formule proposizionali in logica matematica. Una formula è in CNF se è una congiunzione di clausole.*

Seguendo lo stesso approccio per le clausole, la *CNF* può essere rappresentata anch'essa in notazione insiemistica come **insieme di clausole**, quindi ad esempio la seguente *CNF*:

$$(a \vee b) \wedge (\neg a \vee c) \wedge (\neg b \vee c)$$

diventa

$$\{ \quad \{a, b\}, \{\neg a, c\}, \{\neg b, c\} \quad \}$$

anche in questo caso la congiunzione è sottintesa. Possiamo quindi considerare la *CNF* e gli insiemi di clausole come lo stesso oggetto semantico.

Soddisfacibilità di un insieme di clausole

Un insieme di clausole S è **soddisfacibile** se e solo se esiste almeno un'interpretazione che soddisfa **tutte** le clausole appartenenti a S.

2.2 Regola di risoluzione

La regola di risoluzione, che chiamiamo Res, è una regola di inferenza che, a partire da due clausole di premessa, genera una clausola conclusione (detta **risolvente**). Per poter applicare *Res* su due clausole C_1 e C_2, è necessario che esista un letterale $L \in C_1$ ed il suo opposto $\overline{L} \in C_2$. Se questo non dovesse capitare, allora la regola non è applicabile su C_1 e C_2.

2.2.1 Clausola risolvente

Per poter ottenere la clausola risolvente (R) si rimuove il letterale L dalla clausola C_1 ed il suo opposto $\overline{L}$ dalla clausola C_2, infine si uniscono le due clausole (con l'operazione di **unione** insiemistica). Vediamo un esempio:

$$\frac{\overbrace{\{a, b\}}^{C_1} \qquad \overbrace{\{\neg b, c, d\}}^{C_2}}{\underbrace{\{a, c, d\}}_{R}} \text{Res}$$

la risolvente R è stata ottenuta cancellando il letterale "b" (da C_1) ed il suo opposto "$\neg b$" (da C_2), e facendo l'unione.

2.3 Funzionamento del metodo di risoluzione

Il metodo di risoluzione opera su un insieme S di clausole, e applica (dove possibile) *Res* su tutte le coppie di clausole appartenenti a S con lo scopo di trovare la contraddizione (clausola vuota). Ogni volta che il metodo applica *Res*, la risolvente viene aggiunta a S (se non è già presente), la quale potrà poi essere considerata come clausola di premessa per una successiva applicazione di *Res*. Se il metodo riesce a trovare la clausola vuota, allora inferisce che S è **insoddisfacibile**; se invece, dopo aver applicato *Res* su tutte le possibili coppie di clausole non la trova, allora il metodo inferisce che S è **soddisfacibile**.

2.3.1 Gestione delle clausole tautologiche

Nella *CNF* le clausole tautologiche rappresentano una ridondanza, poiché essendo la *CNF* una congiunzione di clausole, esse non indicano alcun valore informativo. Questo si traduce col fatto che nel metodo di risoluzione si possono ignorare le clausole tautologiche, rendendo più semplice l'esecuzione del metodo anche da un punto di vista computazionale (come vedremo nel capitolo 3).

Un altro aspetto da tenere in considerazione è che applicando *Res* è possibile che la risolvente sia una clausola tautologica, in tal caso la risolvente viene scartata (non viene aggiunta a S). Vediamo un esempio di clausole C_1 e C_2 che generano una risolvente R tautologica:

$$\frac{\overbrace{\{a, \neg b\}}^{C_1} \qquad \overbrace{\{\neg a, b\}}^{C_2}}{\underbrace{\{\neg b, b\}}_{R}}\,\text{Res}$$

per ottenere R è stato rimosso il letterale a ed il suo opposto. Si può notare che in questo caso si poteva anche considerare il letterale $\neg b \in C_1$ ed il suo opposto $b \in C_2$ ed ottenere $R' = \{\neg a, a\}$, ma dal punto di vista del metodo di risoluzione non cambia nulla perché R e R' sono entrambe tautologiche, e quindi scartate.

2.3.2 Esempio pratico del metodo di risoluzione

Esempio su un insieme di clausole insoddisfacibile

Consideriamo il seguente insieme di clausole S:

$$S = \{ \quad \{a\}, \{\neg a, b\}, \{\neg b\} \quad \}$$

in questo caso la contraddizione si ricava applicando due volte *Res*:

- **Step 1:**

$$\frac{\overbrace{\{a\}}^{C_1} \qquad \overbrace{\{\neg a, b\}}^{C_2}}{\underbrace{\{b\}}_{R}}\ \mathrm{Res}$$

- **Step 2:**

$$\frac{\overbrace{\{b\}}^{C_1} \qquad \overbrace{\{\neg b\}}^{C_2}}{\underbrace{\{\}}_{R}}\ \mathrm{Res}$$

nello Step 2, la regola *Res* ha trovato la clausola vuota, questo dimostra che S è **insoddisfacibile**. Si noti che la clausola C_1 dello Step 2 è la risolvente R dello Step 1.

Esempio su un insieme di clausole soddisfacibile

Consideriamo ora il seguente insieme di clausole S':

$$S' = \{ \quad \{a\}, \{\neg a, b\}, \{\neg c\} \quad \}$$

in questo caso il metodo di risoluzione applica una sola volta *Res*:

$$\frac{\overbrace{\{a\}}^{C_1} \qquad \overbrace{\{\neg a, b\}}^{C_2}}{\underbrace{\{b\}}_{R}}\ \mathrm{Res}$$

stavolta il metodo non riesce più ad andare avanti, perché *Res* non è più applicabile in nessun'altra coppia di clausole presenti in S', anche tenendo conto della clausola R appena generata. Questo significa che la contraddizione non può essere ricavata, e che quindi S' è **soddisfacibile**.

Capitolo 3

Implementazione in Java del metodo di risoluzione

Questo capitolo si concentra sull'implementazione del metodo di risoluzione per la logica classica in linguaggio Java [Ora24]. Sfruttando la programmazione orientata agli oggetti del linguaggio Java, esploreremo come tradurre i concetti teorici esaminati nei capitoli precedenti in codice eseguibile, analizzando le classi, i metodi e le strutture dati necessari per realizzare efficacemente il metodo di risoluzione. Per approfondimenti sulla programmazione orientata agli oggetti si consulti [LG01].

Partiremo con una panoramica generale dell'architettura del progetto Java, identificando le principali classi per rappresentare le strutture dati necessarie per il metodo di risoluzione. Successivamente, affronteremo la classe che implementa l'algoritmo di risoluzione. Nel capitolo 4 vedremo l'implementazione delle formule della logica proposizionale e la loro conversione in *CNF*.

3.1 Struttura del progetto Java

Di seguito viene mostrato l'elenco dei package contenuti nella directory *src* del progetto, e i file java contenuti in ognuno di essi:

- **literal**
  - *Literal.java*
  - *Atom.java*
  - *NegAtom.java*
- **cnf**
  - *Clause.java*

  - *ClauseSet.java*

- **resolution**
  - *Resolution.java*
  - *Step.java*

- **formula**
  - *Formula.java*
  - *AtomicFormula.java*
  - *CompoundFormula.java*

- **connective**
  - *Connective.java*

- **antlr4**
  - *FormulaExpression.g4*
  - *FormulaExpressionListener.java*
  - *FormulaExpressionBaseListener.java*
  - *FormulaExpressionLexer.java*
  - *FormulaExpressionParser.java*
  - *FormulaListenerImplementation.java*
  - *ParseFormula.java*

- **test**
  - *ResolutionTest.java*
  - (file *txt* per il test)

- *App.java*

In questo capitolo ci concentriamo sui primi tre package dell'elenco, che sono i principali per l'implementazione del metodo di risoluzione. Gli altri package contengono le classi per rappresentare le formule, eseguire il parsing delle formule, effettuare il testing, ed infine è presente file *App.java* che contiene il metodo *main* (si noti che la classe `App` non è contenuta in alcun package). I package *formula*, *connective* e *antlr4* verranno trattati nel capitolo 4, la classe `App` verrà discussa nel capitolo 6, ed infine il package *test* verrà esaminato nel capitolo 7.

3.2 Strutture dati per la CNF

I package *literal* e *cnf* contengono le classi per rappresentare la *CNF*, fondamentale per il metodo di risoluzione.

3.2.1 La classe astratta `Literal` e le classi `Atom` e `NegAtom`

La classe `Literal` permette di rappresentare i letterali. Essa è una classe astratta che fornisce un'interfaccia comune per le classi `Atom` e `NegAtom`, che la estendono per rappresentare rispettivamente un letterale positivo e uno negativo. Questo approccio consente una gestione modulare dei letterali, facilitando l'estensione e il mantenimento del codice. La classe `Literal` contiene una stringa come campo privato che identifica il nome del letterale, e il corrispettivo metodo `getName()` che lo restituisce; inoltre, ha un metodo astratto `getOpposite()` sovrascritto dalle due classi che la estendono, che permette di restituire l'opposto del letterale sul quale viene chiamato (se chiamato su un'istanza di `Atom` restituisce l'istanza corrispondente di `NegAtom`, e viceversa). Il metodo `toString` della classe `NegAtom` usa il carattere "$\sim$" per rappresentare testualmente la negazione logica in un letterale negativo (ad esempio "$\sim p$"). Il codice 3.1 mostra il metodo `getOpposite()` nella classe `Atom`.

Codice 3.1: metodo astratto `getOpposite()` sovrascritto dalla classe `Atom`

```java
@Override
public Literal getOpposite() {
    return new NegAtom(this.getName());
}
```

3.2.2 La classe `Clause`

Questa classe permette di rappresentare le clausole della *CNF*, tenendo conto della notazione insiemistica. La classe contiene il campo `literals` di tipo Set<Literal> che consiste in un insieme di letterali, inoltre contiene anche il campo `index` che specifica un indice numerico che identifica la clausola istanziata.

All'interno della classe `Clause` sono presenti i classici metodi per gli insiemi (`add`, `remove`, `contains`, ecc.), in aggiunta al metodo `union` che permette di eseguire l'unione insiemistica con un'altra clausola specificata come parametro. Un altro importante metodo è `isTautology` che restituisce `true` se e solo se la clausola è una tautologia (si veda il codice 3.2). La classe `Clause` segue il design pattern *iterator*, che permette di iterare facilmente sui letterali di una clausola tramite il ciclo *for-each* di Java.

Codice 3.2: Metodo `isTautology` della classe `Clause`

```
public boolean isTautology() {
    for (Literal l1 : this.literals) {
        for (Literal l2 : this.literals) {
            if (l1.equals(l2.getOpposite())) return true;
        }
    }
    return false;
}
```

Questo codice mostrato esegue un doppio loop sulla clausola (utilizzando il campo `literals`) per verificare la presenza di un letterale ed il suo opposto all'interno di essa. Se questo dovesse capitare, allora la clausola è tautologica e il metodo restituisce `true`; altrimenti restituisce `false` dopo aver terminato il doppio loop.

3.2.3 La classe `ClauseSet`

Questa classe rappresenta la *CNF* in notazione insiemistica, ovvero gli insiemi di clausole. Nella classe è presente il campo `clauses` di tipo `Set<Clause>`, che contiene le clausole dell'istanza di `ClauseSet`. Anche in questo caso ci sono i metodi per gestire gli elementi dell'insieme come nella classe `Clause`, ed il metodo `union` per fare l'unione insiemistica dell'oggetto con un'altra istanza di `ClauseSet`. Importante è il metodo `removeTautologies` che rimuove le clausole tautologiche dall'oggetto, in questo modo si tolgono le ridondanze, rendendo più semplice la *CNF* (si veda il codice 3.3). Anche la classe `ClauseSet` segue il design pattern "*iterator*".

Codice 3.3: Metodo `removeTautologies` della classe `ClauseSet`

```
public void removeTautologies() {
    List<Clause> tautologies = new ArrayList<>();
    for (Clause c : this.clauses) {
        if (c.isTautology()) {
            tautologies.add(c);
        }
    }
    for (Clause taut : tautologies) {
        this.clauses.remove(taut);
    }
}
```

Questo codice esegue la rimozione delle clausole tautologiche: prepara una lista `tautologies` vuota, esegue un loop sul campo `clauses` per aggiungere alla lista le tautologie, ed infine esegue un loop sulla lista `tautologies` per rimuovere le tautologie contenute in `clauses`.

3.3 La classe `Resolution`

La classe `Resolution` è una classe senza costruttori che contiene alcuni campi statici e metodi statici per l'implementazione del metodo di risoluzione. I campi della classe sono tre:

- `visited`: è di tipo `Map<Integer, Set<Integer>>` e consiste in una mappa che associa una clausola ad un insieme di clausole (utilizzando i loro indici). Essa memorizza tutte le clausole alle quali è stata applicata la regola di risoluzione con la clausola rappresentata dalla chiave della mappa (si veda la sottosezione 3.3.2).
- `enableSteps`: è un campo booleano che, se impostato a `true`, permette di abilitare gli step del metodo di risoluzione quando viene eseguito (si veda la sottosezione 3.3.4).
- `trace`: è una lista di `Step`, che memorizza tutti i passaggi del metodo di risoluzione se il campo `enableSteps` è impostato a `true` (si veda la sottosezione 3.3.4).

I metodi statici della classe sono:

- `isSatisfiable` (spiegato nella sottosezione 3.3.1)
- `getComplementaryLiteral` (spiegato nella sottosezione 3.3.1)
- `alreadyVisited` (spiegato nella sottosezione 3.3.2)
- `resolRule` (spiegato nella sottosezione 3.3.3)
- `setEnableSteps` (spiegato nella sottosezione 3.3.4)
- `printTrace` (spiegato nella sottosezione 3.3.4)

3.3.1 Il metodo `isSatisfiable`

Questo è il metodo più importante della classe `Resolution`. Esso ha come parametro in input un oggetto `ClauseSet s` e restituisce `true` se `s` è soddisfacibile, `false` altrimenti. Il metodo `isSatisfiable` dopo aver controllato che `s` non sia `null` o un insieme vuoto, elimina tutte le clausole tautologiche appartenenti a `s` richiamando il metodo `removeTautologies` sull'oggetto `s`. Successivamente controlla se è vuoto (in tal caso viene restituito `true`), e in caso negativo continua l'esecuzione inizializzando i campi `visited` e `trace`; inoltre, tutte le clausole in `s` vengono inserite nella lista di clausole `listCl`, in questo modo le clausole risolventi ottenute con la regola `Res` saranno aggiunte a `listCl`, così da evitare che l'oggetto in input `s` venga modificato.

A questo punto vengono eseguiti due cicli for innestati su `listCl`; il ciclo esterno itera la lista utilizzando la clausola `c1`, il ciclo interno utilizza la clausola `c2`. Il codice 3.4 mostra l'esecuzione dei due cicli.

Codice 3.4: Esecuzione dei cicli `for` innestati, all'interno del metodo `isSatisfiable` della classe `Resolution`

```java
for (int i = 0; i < listCl.size(); i++) {
    Clause c1 = listCl.get(i);
    int index1 = c1.getIndex();
    for (int j = 0; j < listCl.size(); j++) {
        Clause c2 = listCl.get(j);
        int index2 = c2.getIndex();
        if ((i != j) && !alreadyVisited(c1, c2)) {
            Literal complemLit = getComplementaryLiteral(c1, c2);
            if (complemLit != null) {
                if (index1 < index2) {
                    (visited.get(index1)).add(index2);
                } else {
                    (visited.get(index2)).add(index1);
                }
                Clause resolvent = resolRule(c1, c2, complemLit);
                Step step = null;
                if (enableSteps) {
                    step = new Step(c1, c2, resolvent,
                       complemLit);
                    trace.add(step);
                }
                if (resolvent.isEmpty()) {
                    if (enableSteps) printTrace();
                    return false;
                }
                if (resolvent.isTautology()) {
                    if (enableSteps)
                        step.setTautology();
                } else if (listCl.contains(resolvent)) {
                    if (enableSteps)
                        step.setAlreadyPresent();
                } else {
                    visited.put(resolvent.getIndex(), new
                       HashSet<>());
                    listCl.add(resolvent);
                }
            }
        }
    }
}
if (enableSteps) printTrace();
return true;
```

Una volta entrato nel secondo ciclo, il metodo esegue la regola di risoluzione (metodo resolRule) su c1 e c2, se e solo se le due clausole non sono uguali, non sono già state visitate in precedenza, e contengono almeno un letterale complementare in comune (metodo getComplementaryLiteral). Una volta eseguita la regola sulle due clausole, viene controllato se la clausola resolvent è vuota (metodo isEmpty); in tal caso il metodo restituisce false e termina la sua esecuzione, altrimenti la continua e aggiunge la risolvente a listCl se e solo se resolvent non è tautologica e non è già presente nella lista. A questo punto si esegue la successiva iterazione del for interno.

Il metodo getComplementaryLiteral

Questo metodo controlla se nella clausola c1 è presente un letterale l1, ed il suo opposto l2 nella clausola c2. Se lo trova lo restituisce, altrimenti restituisce null. Il codice 3.5 mostra questo metodo.

Codice 3.5: Metodo getComplementaryLiteral della classe Resolution

```java
private static Literal getComplementaryLiteral(Clause c1, Clause
    c2) {
    for (Literal l1 : c1) {
        for (Literal l2 : c2) {
            if (l1.equals(l2.getOpposite())) return l1;
        }
    }
    return null;
}
```

3.3.2 Memorizzazione delle coppie di clausole visitate

Per evitare che si esegua la regola di risoluzione più di una volta su una stessa coppia di clausole, è opportuno memorizzare la coppia sulla mappa visited. Essa utilizza gli indici delle clausole, ed ha come chiave un Integer, e come valore un insieme di interi (Set<Integer>). Ogni volta che viene eseguita la regola di risoluzione su c1 e c2, viene aggiunto l'indice più grande alla mappa ottenuta come valore a partire dalla chiave corrispondente all'indice più piccolo. In questo modo viene gestita più semplicemente la simmetria, infatti se c1 e c2 sono già state visitate, allora vale la stessa cosa anche per c2 e c1. Nel codice 3.4 L'inserimento degli indici nella mappa viene eseguito tra la riga 10 e la riga 14.

Esempio di funzionamento della mappa

Consideriamo il seguente insieme di clausole con associati gli indici ad ognuna di essa:

Clausole	$\{a, \neg b, c\}$	$\{\neg a, d\}$	$\{\neg c\}$
Indici	0	1	2

In questo caso, il metodo `isSatisfiable` richiama il metodo `resolRule` sulle clausole **0** e **1**, e crea la risolvente con indice **3**. Nella mappa viene aggiunto **1** nell'insieme corrispondente alla chiave **0** (perché $0 < 1$). Poi va avanti eseguendo altre volte la regola; le clausole che vengono aggiunte all'insieme di partenza sono le seguenti:

Clausole	$\{\neg b, c, d\}$	$\{a, \neg b\}$	$\{d, \neg b\}$
Indici	3	4	5

Nella situazione finale la mappa visited avrà la seguente struttura:

Chiavi	**Valori**
0	$\{1, 2\}$
1	$\{4\}$
2	$\{3\}$

La prima riga dice che la clausola **0** è stata visitata con le clausole **1** e **2** (infatti le clausole **3** e **4** sono state ottenute dalle coppie **0 - 1** e **0 - 2**). La seconda riga dice che la clausola **1** è stata visitata con la clausola **4** (per ottenere la clausola **5**); infine, la terza riga dice che la clausola **2** è stata visitata con la clausola **3** (in quest'ultimo caso viene generata la clausola **6** che è uguale alla clausola **5**, quindi viene scartata).

Il metodo `alreadyVisited`

Questo metodo utilizza la mappa `visited` per verificare se una coppia di clausole è gia stata visitata in precedenza, ossia la regola *Res* è già stata applicata su di essa. Per farlo controlla se nell'insieme ottenuto a partire dall'indice più piccolo è presente l'indice più grande (utilizzando il metodo `contains`). Questo garantisce che l'algoritmo possa terminare. Il codice 3.6 mostra questo metodo.

Codice 3.6: Metodo `alreadyVisited` della classe `Resolution`

```
private static boolean alreadyVisited(Clause c1, Clause c2) {
    int i1 = c1.getIndex();
    int i2 = c2.getIndex();
    if (i1 < i2) {
        return (visited.get(i1)).contains(i2);
    }
    return (visited.get(i2)).contains(i1);
}
```

3.3.3 Implementazione della regola di risoluzione *Res*

La regola *Res* viene implementata dal metodo `resolRule`, esso prende in input le clausole `c1` e `c2`, insieme al letterale `lit`. Il metodo esegue l'unione delle due clausole (tramite il metodo `union` della classe `Clause`), e successivamente rimuove il letterale `lit` ed il suo opposto dalla clausola risultante (con il metodo `remove`); infine restituisce il risultato. Il codice 3.7 mostra questo metodo.

Codice 3.7: Metodo `resolRule` della classe Resolution

```
private static Clause resolRule(Clause c1, Clause c2,
    Literal lit) {
    Clause result = c1.union(c2);
    result.remove(lit);
    result.remove(lit.getOpposite());
    return result;
}
```

3.3.4 Gestione degli step

Per verificare la correttezza dell'implementazione del metodo di risoluzione è possibile tenere traccia di tutti gli step che vengono eseguiti dal metodo `isSatisfiable`, ovvero di tutte le applicazioni della regola *Res*. Per farlo è stata scritta la classe `Step`, le cui istanze memorizzano il numero di step attuale, le due clausole di premessa, la clausola risolvente, il letterale da considerare per la regola, ed infine le informazioni che dicono che la risolvente viene scartata perché è tautologica oppure già presente nella lista di clausole.

Nella classe `Resolution` è presente il metodo `setEnableSteps` che prende in input un valore booleano che viene impostato sul campo statico `enableSteps`. Se questo campo è impostato a `true`, verrà stampata su Standard Output (*Stdout*) la lista degli step quando viene chiamato il metodo `isSatisfiable`.

Gli step vengono memorizzati nel campo statico `trace` (di tipo `List<Step>`). Nel codice 3.4, nelle righe 16-19 viene creata una nuova istanza di `Step` e aggiunta a `trace` soltanto nel caso in cui il campo `enableSteps` è `true`. Nelle righe 25-30 viene controllato se la clausola risolvente è tautologica oppure è già presente in `listCl`, e vengono chiamati i metodi `setTautology` o `setAlreadyPresent` sullo step, per indicare che in quello step la risolvente viene scartata, specificandone il suo motivo. Nella riga 22 e nella riga 39 viene richiamato il metodo `printTrace`; esso itera sulla lista `trace`, stampando tutti gli step su *Stdout*.

Esempio di funzionamento degli step

Supponendo che il campo `enableSteps` sia impostato a `true`, consideriamo l'esecuzione del metodo `isSatisfiable` con il seguente insieme di clausole `s` in input:

$$\texttt{s} = \{ \quad \{a\}; \{\neg a, b\}; \{\neg b\} \quad \}$$

La lista degli step che verrà stampata su *Stdout* è la seguente:

```
STEP NUMBER 1:
First premise: {a}
Second premise: {~a, b}
Conclusion: {b}
obtained by removing the literal "a" and its opposite.
----------------------------------------------------------

STEP NUMBER 2:
First premise: {~a, b}
Second premise: {~b}
Conclusion: {~a}
obtained by removing the literal "b" and its opposite.
----------------------------------------------------------

STEP NUMBER 3:
First premise: {~b}
Second premise: {b}
Conclusion: {}
obtained by removing the literal "~b" and its opposite.
----------------------------------------------------------
```

Come si può notare, nello step 3 viene trovata la clausola vuota, quindi in questo caso il metodo restituisce `false` (`s` è **insoddisfacibile**).

Capitolo 4

Implementazione in Java di formule generiche

Finora ci siamo concentrati soltanto su formule espresse in *CNF* rappresentate mediante la classe `ClauseSet`. Se volessimo implementare una formula qualsiasi della logica proposizionale sono necessari altri elementi. I package *formula* e *connective* del progetto Java visto nella sezione 3.1 contengono i file necessari per rappresentare le formule della logica proposizionale. Se si volesse verificare la soddisfacibilità di una formula, è necessario convertire prima la formula in *CNF* (ottenendo così l'istanza di `ClauseSet` associata), e poi richiamare il metodo `isSatisfiable` della classe `Resolution` dando in input quella istanza. Questo aspetto viene affrontato nella sezione 4.5.

4.1 La classe astratta `Formula`

Come visto nel capitolo 1, una formula può essere *atomica* o *composta*, perciò nell'implementazione in Java è stata scritta la classe astratta `Formula`, che viene estesa dalle classi concrete `AtomicFormula` e `CompoundFormula`.

La classe `Formula` contiene soltanto il metodo astratto `toCnf` che permette di convertire la formula in *CNF*, restituendo l'istanza di `ClauseSet` corrispondente (si veda la sezione 4.5).

4.2 La classe `AtomicFormula`

Questa classe estende `Formula` ed istanzia una formula atomica. Essa contiene un campo `atm` di tipo `Atom` che lo identifica, e alcuni semplici metodi per la sua gestione:

il metodo per ottenere il nome (`getName`), il metodo per ottenere il letterale associato (`toLiteral`), e i classici metodi `equals` e `toString`.

4.3 L'enumerazione `Connective`

Prima di considerare la classe `CompoundFormula` è bene vedere prima l'enumerazione `Connective`. Essa si trova nel package *connective* e definisce delle costanti enumerative che descrivono i connettivi della logica proposizionale usati dalle formule composte. La seguente tabella mostra le costanti associate alla loro rappresentazione testuale (ottenute dal metodo `toString`):

Costante enumerativa	**Rappresentazione testuale**
NOT	$\sim$
AND	&
OR	$\mid$
IMPLIES	$->$
IFF	$<->$

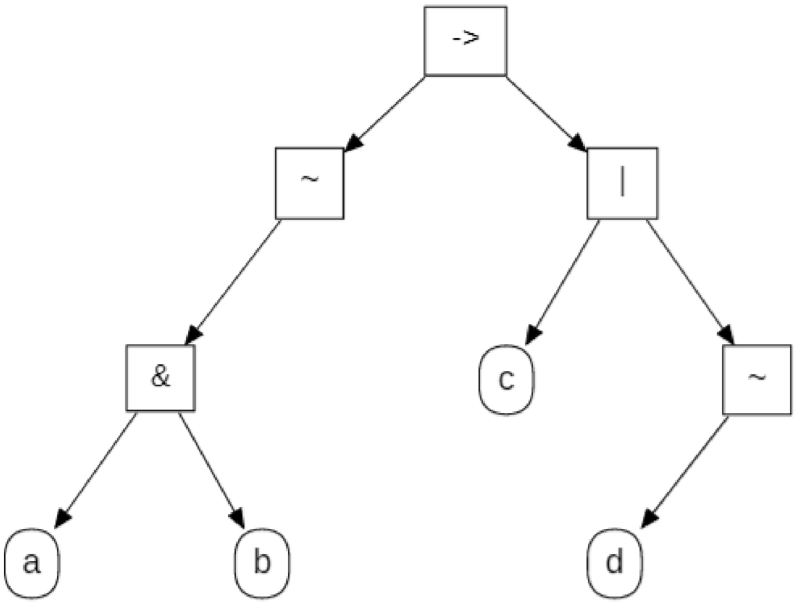
4.4 La classe `CompoundFormula`

Questa classe estende `Formula` ed istanzia una formula composta, utilizzando le costanti dell'enumerazione `Connective`. Per la rappresentazione interna di una formula composta si usa un albero binario, che definisce la sua struttura ricorsiva. La radice dell'albero contiene il connettivo principale della formula, i nodi interni contengono i connettivi delle sue sottoformule, ed infine le foglie contengono le sue formule atomiche. I nodi che contengono i connettivi binari hanno due figli, mentre i nodi che contengono i connettivi unari hanno un solo figlio (il figlio sinistro).

Consideriamo il seguente esempio di formula:

$$\neg(a \wedge b) \rightarrow (c \vee \neg d)$$

La sua rappresentazione ad albero è la seguente:



In questo albero, la radice è il nodo "$->$", perché è il connettivo principale della formula dell'esempio. I nodi "$\sim$" hanno solo il figlio sinistro perché rappresentano la negazione, e le foglie sono le atomiche *a, b, c, d*.

La classe `CompoundFormula` ha quindi come attributi il campo `mainConnective` che definisce il connettivo principale, e il campo `subFormulas` che consiste in un array di formule che contiene i nodi figli. La classe contiene dei metodi getter per ottenere il connettivo, il figlio destro ed il figlio sinistro, e sovrascrive i metodi `toString` e `toCnf`.

4.5 Clausificazione di una formula

Come spiegato all'inizio di questo capitolo, se volessimo verificare la soddisfacibilità di una formula F tramite il metodo di risoluzione, è fondamentale effettuare prima la *clausificazione* di F, ovvero la sua conversione in *CNF*, in modo tale da ottenere l'insieme di clausole necessario per il metodo di risoluzione. Per fare ciò bisogna applicare alcune regole di inferenza:

- **Eliminazione della doppia implicazione**:

$$(p \leftrightarrow q) \quad \equiv \quad ((p \rightarrow q) \land (q \rightarrow p))$$

- **Eliminazione dell'implicazione**:

$$(p \rightarrow q) \quad \equiv \quad (\neg p \vee q)$$

- **Proprietà distributiva di or su and**:

$$(p \vee (q \wedge r)) \quad \equiv \quad ((p \vee q) \wedge (p \vee r))$$

- **Doppia negazione**:

$$\neg\neg p \quad \equiv \quad p$$

- **Leggi di De Morgan**:

$$\neg(p \wedge q) \quad \equiv \quad (\neg p \vee \neg q)$$

$$\neg(p \vee q) \quad \equiv \quad (\neg p \wedge \neg q)$$

Le regole appena mostrate vengono applicate in una delle implementazioni del metodo astratto `toCnf` della classe `Formula`. Questo metodo restituisce un `ClauseSet` e ha due diverse implementazioni: una per le formule atomiche e una per le formule composte.

L'implementazione del metodo `toCnf` nella classe `AtomicFormula` è molto semplice: esso restituisce un `ClauseSet` contenente una sola clausola, tale clausola contiene soltanto il letterale associato a quella formula atomica.

L'implementazione del metodo nella classe `CompoundFormula` invece applica sulla formula le regole di inferenza sopra citate, seguendo un approccio ricorsivo. Per la creazione del `ClauseSet` da restituire, utilizza i metodi `union` delle classi `Clause` e `ClauseSet`.

4.5.1 Esempio di clausificazione

Consideriamo la seguente formula:

$$(a \vee b) \rightarrow c$$

la clausificazione viene fatta in tre passaggi:

- Eliminazione implicazione:

$$\neg(a \vee b) \vee c$$

- De Morgan:

$$(\neg a \wedge \neg b) \vee c$$

- Proprietà distributiva:

$$(\neg a \vee c) \wedge (\neg b \vee c)$$

La formula ricavata nell'ultimo passaggio è in *CNF*, ed essa corrisponde all'insieme di clausole:

$$\{ \quad \{\neg a, c\}, \{\neg b, c\} \quad \}$$

4.6 Parsing di formule

Se volessimo utilizzare un programma che legge da un flusso di input una formula in formato testuale (ad esempio da *Stdin* o da un file di testo), è necessario interpretare la stringa di testo in modo tale da ottenere la formula. La trasformazione di rappresentazioni testuali in strutture dati utilizzabili da un programma è un processo noto come *parsing*. Nel nostro sistema, implementiamo un parser per convertire le formule da stringhe di testo in oggetti di tipo `Formula`. Per eseguire il parsing, bisogna definire prima di tutto una grammatica che descrive il linguaggio utilizzato per le formule. A partire dalla grammatica vengono generati il *lexer*, il *parser* e il *listener*.

4.6.1 Utilizzo del parser ANTLR4

Il parser utilizzato nel nostro sistema è *ANTLR4* [Pro24, Par13]. Nella struttura del progetto presente nella sezione 3.1, nel package *antlr4* è presente il file *FormulaExpression.g4* che contiene la grammatica. Inoltre, sono presenti quattro file che sono stati generati automaticamente da *ANTLR4* a partire dalla grammatica:

- **FormulaExpressionLexer.java**: esso consiste nel lexer, ovvero l'analizzatore lessicale; il suo scopo è quello di analizzare la stringa in input ed ottenere da esso un flusso di *token*. In questo caso i token sono i connettivi, le variabili atomiche, e i caratteri di *white space*.

- **FormulaExpressionParser.java**: esso consiste nel parser, che esegue l'analisi sintattica della stringa in input generando un **albero sintattico** (*AST*), fondamentale per la costruzione della formula.

- **FormulaExpressionListener.java**: è un'interfaccia che definisce i metodi che vengono richiamati durante l'attraversamento dell'*AST*.

- **FormulaExpressionBaseListener.java**: è una classe che implementa l'interfaccia `FormulaExpressionListener`, dove ogni metodo sovrascritto non esegue alcuna operazione.

Oltre a questi file, ne sono presenti altri due:

- **FormulaListenerImplementation.java**: è una classe che estende il Base Listener. Utilizza uno *stack* di formule come campo per la creazione delle sottoformule, e sovrascrive alcuni metodi del Base Listener.
- **ParseFormula.java**: è una classe che contiene soltanto il metodo `parse`. Esso è il metodo principale che prende in input una stringa, e mette insieme tutte le componenti viste precedentemente, creando tutto l'occorrente per il parsing. Infine restituisce la formula ottenuta a partire dalla stringa in input.

La grammatica utilizzata

Nel codice 4.1 è presente il contenuto del file *FormulaExpression.g4* che descrive la grammatica.

Codice 4.1: file FormulaExpression.g4

```
grammar FormulaExpression;

start : formula EOF;

formula : atomic_formula
        | '(' formula ')'
        | unary_connective formula
        | formula binary_connective formula
        ;

atomic_formula : LITERAL ;

unary_connective : NOT ;

binary_connective : AND | OR | IMPLIES | IFF;

//token
AND : '&' ;
OR : '|' ;
IMPLIES : '->' ;
NOT : '~' ;
IFF : '<->';

LITERAL : [a-z]+ ;
WS : [ \t\r\n]+ -> skip ;
```

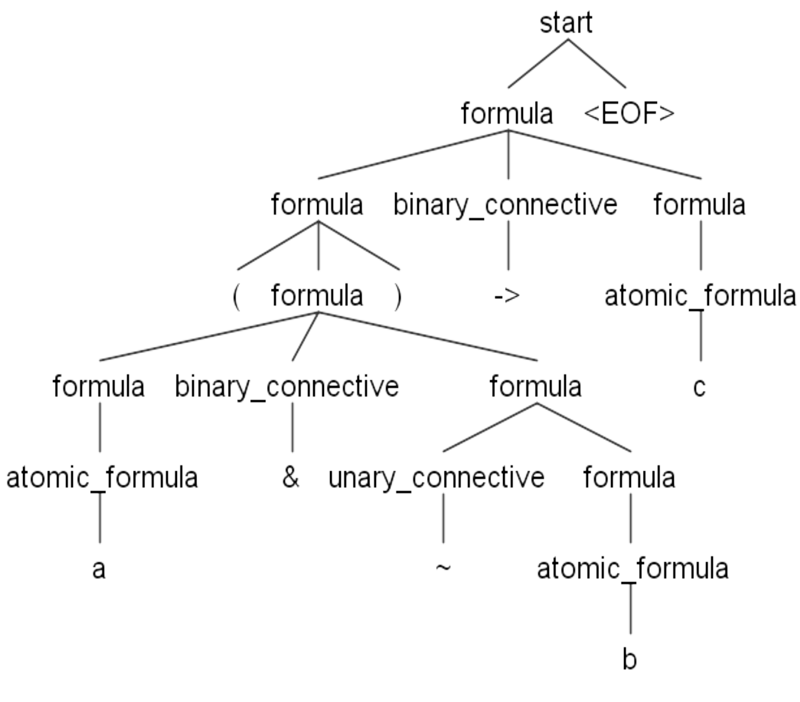
In questa grammatica ci sono le regole che definiscono le formule atomiche e composte, e le regole che descrivono i connettivi unari e binari. Le sottoformule sono distinguibili grazie alle parentesi tonde, che specificano anche la precedenza tra i vari connettivi della formula. in fondo al file è presente la definizione dei token. È bene notare che la rappresentazione testuale dei connettivi in questa grammatica è la medesima utilizzata dall'enumerazione `Connective`.

Esempio di *AST*

Consideriamo la seguente formula rappresentata mediante una stringa:

$$(a \ \& \ \sim b) \quad -> \quad c$$

se eseguiamo il parsing di questa stringa otteniamo il seguente *AST*:



Durante l'attraversamento di questo albero vengono eseguiti i metodi `ExitAtomic_formula` e `exitFormula` della classe `FormulaListenerImplementation`; essi vengono richiamati rispettivamente quando si esce da un nodo *atomic_formula* e da un nodo *formula*. Nel primo caso si crea una formula atomica e la si aggiunge allo *stack* (tramite l'operazione di *push*), nel secondo caso si controlla la presenza di un figlio che contiene un connettivo unario o binario, e si crea una formula composta applicando quel connettivo alla formula in cima allo stack (nel caso di connettivo unario) oppure alle prime due formule in cima allo stack (nel caso di connettivo binario). Infine viene inserita nello stack la formula appena creata. Per ottenere l'elemento in cima allo stack viene usata l'operazione di *pop* su di esso.

Alla fine della visita dell'*AST*, in cima allo stack ci sarà la formula associata alla stringa di partenza, che viene ottenuta dal metodo `getFormula` e a sua volta restituita dal metodo `parse` (si veda il codice 4.2). Nell'esempio visto in precedenza la formula ottenuta è:

$$(a \wedge \neg b) \rightarrow c$$

Codice 4.2: Metodo `parse` della classe `ParseFormula`

```java
public static Formula parse(String formulaStr) {
    CharStream input = CharStreams.fromString(formulaStr);
    FormulaExpressionLexer lexer = new
        FormulaExpressionLexer(input);
    CommonTokenStream tokens = new CommonTokenStream(lexer);
    FormulaExpressionParser parser = new
        FormulaExpressionParser(tokens);

    //create listener
    FormulaListenerImplementation listener = new
        FormulaListenerImplementation();

    //adds the listener to the parser
    parser.addParseListener(listener);

    try {
        //analyze the input and get the corresponding formula
        parser.start();
        return listener.getFormula();
    } catch (Exception e) {
        return null;
    }
}
```

Capitolo 5

Logiche modali non-normali ed estensione del metodo di risoluzione

Questo capitolo si concentra sulle logiche modali, in particolare sulle logiche non-normali presentate nell'articolo [PON23]; principalmente, mostreremo l'implementazione di una di queste logiche ottenuta estendendo il metodo di risoluzione della logica classica. La prima parte del capitolo illustra le nozioni teoriche, la seconda parte riguarda l'implementazione in Java di ciò che viene spiegato nella parte teorica.

Definizione 5.0.1. *Una logica modale è un'estensione della logica classica che introduce connettivi modali per esprimere modalità come necessità e possibilità. Questi connettivi permettono di ragionare non solo sulle proposizioni, ma anche su ciò che è possibile o necessario in vari mondi possibili.*

Il connettivo modale che noi consideriamo è il *box* ($\Box$); esso viene utilizzato nelle formule delle logiche modali, insieme agli altri connettivi della logica classica. Questo connettivo è fondamentale nella formalizzazione di concetti come la necessità, l'obbligatorietà, la conoscenza o la verità assoluta, a seconda del contesto in cui viene utilizzato.

L'interpretazione delle formule in questo tipo di logiche è più complicata rispetto all'interpretazione nella logica classica che abbiamo visto nel capitolo 1; per approfondire questo argomento si consulti [PON23].

5.1 Panoramica sulle logiche modali non-normali

Le logiche modali *non-normali* (*NNML*) sono una classe di sistemi logici che estendono la logica modale normale introducendo concetti e assiomi che non sono presenti nella forma più basilare della logica modale. Queste logiche *non-normali* si discostano dalle proprietà della logica modale *normale* minima, **K**, che soddisfa tutti gli assiomi basilari della logica modale. Queste logiche sono organizzate gerarchicamente in un "cubo" di sistemi logici, che comprende la logica modale **E** come la più semplice e altre logiche ottenute aggiungendo assiomi come **C**, **M** e **N**. Il metodo di risoluzione che vediamo in questo capitolo si concentra soltanto sulla logica modale non-normale minimale **E**.

Gli assiomi della logica minimale **E** comprendono gli assiomi della logica classica e la regola di congruenza **RE**: da $\phi \leftrightarrow \psi$ si inferisce $\Box\phi \leftrightarrow \Box\psi$, dove ϕ e ψ sono due formule.

5.2 Forma clausale

La forma clausale di una formula ϕ consiste nell'esprimere ϕ come un insieme di clausole, come nel caso della logica classica.

Letterali

I letterali possono essere di due tipi: *proposizionali* o *modali.* I letterali proposizionali sono i medesimi della logica classica (si veda sezione 2.1); i letterali modali sono della forma $\Box p$ oppure $\neg\Box p$, dove p è una variabile proposizionale.

Clausole

Le clausole, come nella logica classica, sono una disgiunzione di letterali. Questa volta però le clausole possono essere di due tipi:

- **Clausola locale**:
$$l_1 \vee l_2 \vee \ldots \vee l_n$$
- **Clausola globale**:
$$G(l_1 \vee l_2 \vee \ldots \vee l_n)$$

dove l_i è un letterale proposizionale o modale. Una clausola globale è vera in tutti i mondi di un modello, una clausola locale invece è vera soltanto in un mondo del modello. Per approfondire il concetto di "mondo" di un modello si faccia riferimento a [PON23].

Per rappresentare le clausole è possibile utilizzare la notazione insiemistica, come nella logica classica.

5.3 Clausificazione di formule modali

Per poter effettuare la clausificazione di una formula ϕ è necessario utilizzare la funzione η. Essa è una funzione iniettiva di rinominazione, il cui scopo è quello di mappare una nuova variabile proposizionale ad ogni sottoformula di ϕ; è importante che questa nuova variabile non sia presente in ϕ. La funzione η viene utilizzata all'interno di un'altra funzione, chiamata **R**, ossia la funzione di *riduzione*. Queste funzioni vengono descritte in [PON23]. La clausificazione della formula ϕ viene fatta nel seguente modo:

$$\{\eta(\phi)\} \cup R(G(\eta(\phi) \leftrightarrow \phi))$$

dove **R** è così definita (si consideri t e p come variabili proposizionali):

$$\begin{aligned}
R(G(t \leftrightarrow p)) &= \{G(\neg t \vee p), G(t \vee \neg p)\} \\
R(G(t \leftrightarrow \neg\psi)) &= \{G(\neg t \vee \neg\eta(\psi)), G(t \vee \eta(\psi))\} \cup R(G(\eta(\psi) \leftrightarrow \psi)) \\
R(G(t \leftrightarrow \psi \vee \psi')) &= \{G(\neg t \vee \eta(\psi) \vee \eta(\psi')), G(t \vee \neg\eta(\psi)), G(t \vee \neg\eta(\psi'))\} \\
&\quad \cup R(G(\eta(\psi) \leftrightarrow \psi)) \cup R(G(\eta(\psi') \leftrightarrow \psi')) \\
R(G(t \leftrightarrow \Box\psi)) &= \{G(\neg t \vee \Box\eta(\psi)), G(t \vee \neg\Box\eta(\psi))\} \cup R(G(\eta(\psi) \leftrightarrow \psi))
\end{aligned}$$

Essa è una funzione ricorsiva, in cui il caso base è rappresentato da $R(G(t \leftrightarrow p))$. La clausola $\{\eta(\phi)\}$ è una clausola locale che contiene soltanto la variabile proposizionale associata alla formula ϕ, ossia $\eta(\phi)$. L'insieme di clausole che contiene solo questa clausola viene unito ad un altro insieme di clausole ottenuto ricorsivamente dalla funzione **R** applicata alla formula globale $G(\eta(\phi) \leftrightarrow \phi)$. Si noti che questa funzione è definita solamente per formule ϕ che contengono i connettivi *not*, *or* e *box*. Gli altri connettivi binari possono essere ricavati utilizzando i connettivi *or* e *not*.

Esempio di clausificazione

Consideriamo la seguente formula ϕ:

$$\phi = \Box(\neg a \vee b)$$

Prima di procedere con la clausificazione dobbiamo vedere quali sono le nuove variabili proposizionali ottenute dalla funzione di rinominazione η su ogni sottoformula ψ. La tabella 1 mostra questa mappatura. Si noti che ϕ è una sottoformula di se stessa.

ψ	$\eta(\psi)$
$\Box(\neg a \vee b)$	$\$p_0$
$\neg a \vee b$	$\$p_1$
$\neg a$	$\$p_2$
a	$\$p_3$
b	$\$p_4$

Tabella 1: Funzione η applicata su tutte le sottoformule ψ di ϕ

Di seguito viene mostrato l'elenco degli step per effettuare la clausificazione.

1. Si ottiene la clausola locale $\{\eta(\phi)\}$:

$$\{\$p_0\}$$

2. A questa clausola si aggiungono le clausole globali ottenute da $R(G(\eta(\phi) \leftrightarrow \phi))$:

$$G(\{\neg\$p_0, \Box\$p_1\})$$

$$G(\{\$p_0, \neg\Box\$p_1\})$$

3. Si aggiungono le clausole ottenute da $R(G(\eta(\psi) \leftrightarrow \psi))$, dove $\psi = \neg a \vee b$:

$$G(\{\neg\$p_1, \$p_2, \$p_4\})$$

$$G(\{\$p_1, \neg\$p_2\})$$
$$G(\{\$p_1, \neg\$p_4\})$$

4. Si aggiungono le clausole ottenute da $R(G(\eta(\psi) \leftrightarrow \psi))$ e da $R(G(\eta(\psi') \leftrightarrow \psi'))$ dove $\psi = \neg a$ e $\psi' = b$. Consideriamo $R(G(\eta(\psi) \leftrightarrow \psi))$, si ottengono le seguenti clausole:

$$G(\{\neg\$p_2, \neg\$p_3\})$$
$$G(\{\$p_2, \$p_3\})$$

5. Si aggiungono le clausole ottenute da $R(G(\eta(\psi) \leftrightarrow \psi))$, dove $\psi = a$ (**caso base**):

$$G(\{\neg\$p_3, a\})$$
$$G(\{\$p_3, \neg a\})$$

6. Ritorniamo allo step 4, e consideriamo $R(G(\eta(\psi') \leftrightarrow \psi'))$ (**caso base**). Si aggiungono le seguenti clausole:

$$G(\{\neg\$p_4, b\})$$

$$G(\{\$p_4, \neg b\})$$

Nella sezione 5.6.1, che riguarda l'implementazione della clausificazione, vedremo che è possibile semplificare la clausificazione di una formula in modo da ottenere meno clausole rispetto a quelle che si otterrebbero applicando la clausificazione come sopra riportato.

5.4 Metodo di Risoluzione per logiche modali non-normali

Il metodo di risoluzione per le logiche modali non-normali è un approccio sistematico per determinare la soddisfacibilità delle formule (espresse in forma clausale) di queste logiche. Esso rappresenta un'estensione del metodo di risoluzione per la logica classica, che considera l'utilizzo di clausole locali e globali, e l'applicazione di regole di risoluzione adattate ad una logica L presa in esame (RES_L). Come nella logica classica, si considerano tutte le coppie di clausole e si cerca di trovare la contraddizione applicando le regole. Qui ci concentriamo solo sulle regole per la logica modale minimale **E**, ovvero RES_E. Le regole di risoluzione per le altre logiche non-normali sono presentate nell'articolo [PON23].

5.4.1 Regole di risoluzione per la logica modale *E*

Nelle regole RES_E che vediamo ora, C e D sono clausole, l è un letterale, e p è una variabile proposizionale. Le regole RES_E sono le seguenti:

$$\textbf{LRES} \quad \frac{(D \vee l) \quad (D' \vee \neg l)}{(D \vee D')}$$

$$\textbf{GRES} \quad \frac{G(D \vee l) \quad G(D' \vee \neg l)}{G(D \vee D')}$$

$$\textbf{G2L} \quad \frac{G(D)}{D}$$

$$\textbf{LERES} \quad \frac{(D \vee \Box p) \quad (D' \vee \neg \Box p') \quad G(C) \quad G(C')}{(D \vee D')}$$

$$\textbf{GERES} \quad \frac{G(D \vee \Box p) \quad G(D' \vee \neg \Box p') \quad G(C) \quad G(C')}{G(D \vee D')}$$

dove $C \subseteq (\neg p \vee p')$ e $C' \subseteq (p \vee \neg p')$

Di seguito viene riportata la descrizione di ogni regola:

- **LRES**: Questa regola è la medesima della regola *Res* della logica classica (si veda sezione 2.2), con la differenza che le due premesse e la risolvente sono clausole locali, e che l presente nelle premesse può essere un letterale modale.
- **GRES**: Come *LRES*, ma le due premesse e la risolvente sono clausole globali.
- **G2L**: Questa regola permette di ricavare una clausola locale a partire dalla sua controparte globale (la soddisfacibilità locale è conseguenza di quella globale).
- **LERES**: Questa regola afferma che i letterali modali $\Box p$ e $\neg\Box p'$ possono essere considerati come l'uno l'opposto dell'altro ogni volta che i letterali proposizionali p e p' sono globalmente equivalenti. Questo avviene in tre casi:
  1. quando $G(C) = G(\neg p \vee p')$ e $G(C') = G(p \vee \neg p')$, questo significa che p e p' sono semanticamente equivalenti (ossia $p \leftrightarrow p'$).
  2. quando $G(C) = G(\neg p)$ e $G(C') = G(\neg p')$, in questo caso p e p' sono entrambi globalmente falsi, e quindi semanticamente equivalenti.
  3. quando $G(C) = G(p')$ e $G(C') = G(p)$, in questo caso p e p' sono entrambi globalmente veri, e quindi semanticamente equivalenti.

  Ogni volta che si rientra in uno di questi casi, si considera l'approccio classico della regola di risoluzione, ovvero si genera la risolvente cancellando i letterali modali $\Box p$ e $\neg\Box p'$ dalle prime due premesse, e facendo l'unione. In questo caso le prime due premesse e la risolvente sono clausole locali (mentre le ultime due premesse devono essere sempre globali).

- **GERES**: Come *LERES*, ma le prime due premesse e la risolvente sono clausole globali.

Implementazione in Java

Ora vediamo la seconda parte di questo capitolo, che si concentra sull'implementare in Java i concetti che abbiamo visto finora nel capitolo. La struttura del progetto Java è simile a quella del progetto per la logica classica (si veda sezione 3.1), e anche alcune classi sono simili a quelle della logica classica.

5.5 Implementazione della forma clausale

Le classi per i letterali

Come nell'implementazione per la logica classica, è presente la classe astratta `Literal` (si veda sezione 3.2.1). Questa classe viene estesa dalle classi `PropAtom` e `NegPropAtom` che rappresentano i letterali proposizionali, e dalle classi `ModalAtom` e `NegModalAtom` che rappresentano i letterali modali. Per la rappresentazione testuale dei letterali modali si usa il carattere "#", ad esempio "$\#p$" o "$\sim\#q$".

Le classi per le clausole

Per quanto riguarda le clausole, è presente la classe astratta `Clause`. Questa classe è simile a quella utilizzata nella logica classica (si veda sezione 3.2.2), con la differenza che in questo caso si tratta di una classe astratta che definisce il metodo astratto `union`; gli altri metodi sono gli stessi della classe `Clause` per la logica classica.

La classe `Clause` viene estesa dalle due classi `LocalClause` e `GlobalClause`, che rappresentano rispettivamente le clausole locali e globali. Queste due classi forniscono un'implementazione del metodo `union`, infatti una clausola di un tipo (locale o globale) può essere unita soltanto con un'altra clausola dello stesso tipo. Inoltre le due classi sovrascrivono i metodi `equals` e `toString` (nella rappresentazione testuale delle clausole globali viene usato "G($\cdot$)").

La classe `ClauseSet`

La classe `ClauseSet` rappresenta la forma clausale di una formula, espressa come insieme di clausole. Questa classe è la medesima utilizzata nella logica classica (si veda sezione 3.2.3). La differenza sostanziale è che le clausole all'interno di un'istanza di `ClauseSet` possono essere indistintamente locali o globali, e i letterali contenuti in ciascuna clausola possono essere parimenti proposizionali o modali.

5.6 implementazione delle formule modali

Per l'implementazione delle formule modali generiche si segue lo stesso approccio usato per la logica classica (si veda l'inizio del capitolo 4); quindi si usa la classe astratta `Formula` che viene estesa dalle classi `AtomicFormula` e `CompoundFormula`. La classe `AtomicFormula` è identica a quella presentata nella sezione 4.2. La classe `CompoundFormula` utilizza l'enumerazione `Connective` estesa al connettivo unario *box* (si utilizza la costante enumerativa `BOX`, la cui rappresentazione testuale usa il carattere "#"). La classe `AtomicFormula` implementa le formule seguendo lo stesso approccio con alberi binari che abbiamo già visto; i nodi che descrivono il connettivo *box* hanno solo il figlio sinistro, come nel caso del *not* (perché è unario).

5.6.1 Implementazione della clausificazione

La classe `Eta`

La clausificazione viene fatta utilizzando la classe `Eta`, essa gestisce l'esecuzione della funzione di rinominazione η. Questa classe ha come campo una mappa che associa istanze di `Formula` a istanze di `PropAtom`; il costruttore della classe prende in input un oggetto `Formula` f e crea ricorsivamente la mappa associando ad ogni sottoformula di f una nuova variabile proposizionale (`PropAtom`), i nomi che vengono dati alle variabili atomiche sono $\$p1, \$p2, \ldots, \$pi$.

La classe *Eta* ha il metodo `getPropVariable` che prende in input un oggetto `Formula` f e restituisce il valore $\eta(f)$. Un'istanza di `Eta` viene usata come campo statico all'interno della classe `Formula`.

I metodi per la clausificazione

La classe `CompoundFormula` permette di eseguire la clausificazione tramite i metodi `toClauseSet`, `R` e `classicClausification`. La clausificazione viene implementata in maniera leggermente diversa rispetto a quanto riportato in questo capitolo (nella sezione 5.3), infatti è possibile considerare $\eta(p) = p$ nel caso in cui p sia una formula atomica (`AtomicFormula`); in questo modo si evita il caso $R(G(\eta(p) \leftrightarrow p))$. Inoltre, è possibile applicare la clausificazione della logica classica quando la funzione **R** arriva a calcolare $R(G(\eta(F) \leftrightarrow F))$, dove F è una formula della logica classica (ossia non contiene il connettivo *box*); in questo caso si fa la clausificazione classica della formula $\eta(F) \leftrightarrow F$, che consiste in un caso base della funzione **R**. Si può applicare la clausificazione classica anche quando la formula F contiene il connettivo *box* ma solo se esso viene applicato alle atomiche della formula. Ad esempio, se avessimo la formula $(\Box a \wedge b) \rightarrow \Box c$, possiamo applicare su di essa la clausificazione classica, senza usare la funzione η.

Vediamo ora una descrizione dei metodi usati per la clausificazione.

Il metodo `toClauseSet`

Questo metodo è il primo ad essere richiamato per la clausificazione, ed è l'unico dei tre ad essere pubblico. Esso controlla se la formula `this` da clausificare è classica, in tal caso richiama il metodo `classicClausification`. Se invece non è classica, procede con la parte iniziale della clausificazione con η: crea una nuova istanza del campo statico `Eta` (della *superclasse* `Formula`), e crea un nuovo `ClauseSet` vuoto nel quale aggiunge la clausola locale $\{\eta(\mathit{this})\}$, ed infine restituisce l'unione con il `ClauseSet` ottenuto da $R(G(\eta(\mathit{this}) \leftrightarrow \mathit{this}))$. Il codice 5.1 mostra questo metodo.

Il metodo `R`

Il metodo `R` prende in input l'oggetto `PropAtom t` corrispondente alla rinominazione della formula sulla quale viene richiamato il metodo, ad esempio: scrivere `this.R(t)` equivale ad applicare $R(G(t \leftrightarrow \mathit{this}))$, dove $t = \eta(\mathit{this})$. Il metodo applica la funzione **R** in maniera ricorsiva come abbiamo visto, tenendo conto anche dei casi di doppia negazione (questo capita quando si calcola $R(G(t \leftrightarrow \neg\neg\psi)))$, ed evita di calcolare $R(G(p \leftrightarrow p))$ (dove p è atomica) fermandosi alla chiamata ricorsiva precedente. Inoltre considera come caso base il calcolo di $R(G(t \leftrightarrow F))$ dove F è una formula della logica classica (richiamando in questo caso il metodo `classicClausification` su $t \leftrightarrow F$). Siccome questo metodo è lungo, il codice 5.2 riporta soltanto il caso in cui viene calcolato $R(G(t \leftrightarrow \neg\psi))$; in questo codice la variabile `psi` è il figlio sinistro della formula `this` (si noti che `this` è $\neg\psi$), la variabile `etaPsi` è il valore η(`psi`) ed infine la variabile `res` è un `ClauseSet` inizialmente vuoto.

Il metodo `classicClausification`

Questo metodo esegue la clausificazione classica, applicando le regole di inferenza riportate nella sezione 4.5. Quindi il metodo `classicClausification` è uguale al metodo `toCnf` per la logica classica, con la differenza che tutte le clausole restituite sono globali; inoltre, gestisce anche le formule che contengono il connettivo *box* solo se compare davanti alle atomiche (ad esempio se contengono $\Box p$ e/o $\neg\Box q$).

Ad esempio, se dobbiamo clausificare la formula $\neg(\Box p \lor \neg\Box q)$ viene eseguito il metodo `classicClausification`, che restituisce il seguente insieme di clausole:

$$\{\ G(\neg\Box p),\ G(\Box q)\ \}$$

In questo modo si evita di utilizzare le funzioni η e R, rendendo più semplice la clausificazione.

Codice 5.1: Metodo `toClauseSet` della classe `CompoundFormula`

```
@Override
public ClauseSet toClauseSet() {
    if (this.isClassic()) {
        return this.classicClausification();
    } else {
        eta = new Eta(this); //eta e' il campo statico della
            classe Formula.
        PropAtom t = eta.getPropVariable(this);
        ClauseSet cs = new ClauseSet(new LocalClause(t));
        return cs.union(this.R(t));
    }
}
```

Codice 5.2: Parte del metodo `R` della classe `CompoundFormula`

```
switch (this.mainConnective) {
    case NOT:
        GlobalClause gc1 = new GlobalClause();
        gc1.add(t.getOpposite());
        gc1.add(etaPsi.getOpposite());
        res.add(gc1);
        GlobalClause gc2 = new GlobalClause();
        gc2.add(t);
        gc2.add(etaPsi);
        res.add(gc2);
        if (psi instanceof CompoundFormula) {
            CompoundFormula cf_psi = (CompoundFormula) psi;
            if (cf_psi.isClassic()) {
                CompoundFormula f = new CompoundFormula(IFF, new
                    AtomicFormula(etaPsi), psi);
                return res.union(f.classicClausification());
            }
            return res.union(cf_psi.R(etaPsi));
        }
        return res;
    case OR:
        ...
}
```

Parsing di formule modali

Il parsing di formule modali viene fatto nello stesso modo che abbiamo visto nella sezione 4.6, in questo caso però si deve considerare il connettivo *box*. Per farlo, bisogna modificare la grammatica presente nel codice 4.1, aggiungendo nella definizione dei token la riga:

BOX : '#';

e modificando la regola *unary_connective* nel seguente modo:

unary_connective : NOT | BOX;

In questo modo l'AST terrà conto anche del connettivo *box*.

5.7 Implementazione del metodo di risoluzione

L'implementazione del metodo di risoluzione viene svolta, come nel caso della logica classica, all'interno della classe `Resolution`; essa è una classe senza costruttori dove sono presenti dei campi statici e metodi statici. I campi statici sono quelli descritti nella sezione 3.3, ossia `visited`, `enableSteps` e `trace`. i metodi della classe sono i seguenti:

- `isSatisfiable`
- `getComplementaryLiteral`
- `getModalLiterals`
- `addCoupleInMap`
- `alreadyVisited`
- `setEnableSteps`
- `printTrace`
- `G2L`
- `LRES`
- `GRES`
- `LERES`
- `GERES`

Il metodo `isSatisfiable`

Il metodo `isSatisfiable` è il metodo principale che implementa l'algoritmo di risoluzione ed è l'unico ad essere pubblico. Questo metodo prende in input un oggetto `ClauseSet s` e restituisce `true` se `s` è soddisfacibile, `false` altrimenti. Inizialmente, Dopo aver controllato che `s` non sia `null` e non sia vuoto, elimina tutte le eventuali clausole tautologiche e crea la lista di clausole `listCl` dove aggiunge tutte le clausole rimaste in `s`. Lo svolgimento del metodo `isSatisfiable` può essere diviso in tre parti:

1. **Applicazione della regola G2L**: ad ogni clausola globale contenuta in `listCl` viene applicata la regola *G2L* (tramite il metodo `G2L`), e le nuove clausole ottenute vengono aggiunte a `listCl`.

2. **Applicazione delle regole LRES e GRES**: vengono eseguiti due cicli *for* innestati sulla lista `listCl` in modo da applicare (dove possibile) la regola *LRES* su tutte le coppie di clausole locali, e la regola *GRES* su tutte le coppie di clausole globali (tramite i metodi `LRES` e `GRES`). Per verificare il tipo di clausola si utilizza l'operatore `instanceof`, per ottenere il letterale l (oppure $\neg l$), si usa il metodo `getComplementaryLiteral`, il cui funzionamento è il medesimo dell'omonimo metodo spiegato nella sezione 3.3.1. Ogni volta che vengono eseguite le regole *LRES* e *GRES* viene fatto un controllo sulla risolvente: se è vuota, il

metodo restituisce `false`, altrimenti la risolvente viene aggiunta a `listCl` (se e solo se non è tautologica oppure non è già presente nella lista) e il metodo continua la sua esecuzione.

3. **Applicazione delle regole LERES e GERES**: viene eseguito un altro paio di *for* innestati sulla lista `listCl`, questa volta cercando di applicare (dove possibile) la regola *LERES* su tutte le coppie di clausole locali, e la regola *GERES* su tutte le coppie di clausole globali; infine esegue sulla risolvente le stesse operazioni del caso precedente ogni volta che viene applicata una delle due regole. Per poter ottenere i letterali modali $\Box p$ e $\neg\Box p'$ necessari per attuare le regole *LERES* e *GERES*, si utilizza il metodo `getModalLiterals`. Il codice 5.5 mostra la parte del metodo `isSatisfiable` dove sono presenti i due cicli `for` innestati per effettuare le regole *LERES* e *GERES*.

Il metodo `getModalLiterals`

Questo metodo verifica se è possibile applicare le regole *LERES* e *GERES* all'interno del metodo `isSatisfiable`, ed in caso affermativo restituisce i due letterali modali $\Box p$ e $\neg\Box p'$. Esso prende in input due clausole, una lista di clausole (nel nostro caso `listCl`) e restituisce un array di letterali (`Literal[]`). Il metodo controlla se in una delle due clausole in input sia presente un letterale modale positivo (`ModalAtom`), ed un letterale modale negativo (`NegModalAtom`); se ciò dovesse capitare, allora ispeziona la lista di clausole fornita in input (`listCl`) con lo scopo di ricercare le clausole $G(C)$ e $G(C')$, ovvero le ultime due premesse delle regole *LERES* e *GERES*. Se vengono trovate, allora il metodo `getModalLiterals` restituisce l'array contenente i due letterali modali, in caso contrario restituisce un array vuoto.

5.7.1 Implementazione delle regole di risoluzione

Le regole di risoluzione vengono implementate mediante gli omonimi metodi presenti nella classe `Resolution`.

Il metodo `G2L`

Il metodo `G2L` prende in input una clausola globale `gc`, e restituisce la corrispondente locale. Per farlo, istanzia una clausola locale vuota, e la riempie aggiungendo tutti i letterali presenti in `gc`.

I metodi `LRES` e `GRES`

Questi metodi sono implementati nello stesso modo del metodo `resolRule` della logica classica, con la differenza che nel caso di `LRES`, le due clausole in input (le clausole di premessa) e la clausola restituita (risolvente) sono locali, e nel caso di `GRES` le clausole in input e la clausola restituita sono globali. il codice 5.3 mostra l'implementazione del metodo `LRES`.

I metodi `LERES` e `GERES`

Il metodo `LERES` prende in input due clausole locali e due letterali, e restituisce una clausola locale. Il metodo effettua l'unione delle due clausole, cancella i due letterali dal risultato, ed infine lo restituisce. Il metodo `GERES` effettua le stesse operazioni, ma le due clausole in input e la clausola restituita sono globali. il codice 5.4 mostra l'implementazione del metodo `LERES`. Si noti che il metodo non controlla se i due letterali in input siano modali, questo perché ci si assicura che il metodo `isSatisfiable` richiami questo metodo con gli input corretti; inoltre, le due clausole $G(C)$ e $G(C')$ non vengono considerate, perché il metodo viene richiamato soltanto nel caso in cui `getModalLiterals` non restituisce un array vuoto.

Codice 5.3: Metodo `LRES` della classe `Resolution`

```
private static LocalClause LRES(LocalClause lc1, LocalClause lc2, Literal lit) {
    LocalClause result = (LocalClause) lc1.union(lc2);
    result.remove(lit);
    result.remove(lit.getOpposite());
    return result;
}
```

Codice 5.4: Metodo `LERES` della classe `Resolution`

```
private static LocalClause LERES(LocalClause lc1, LocalClause lc2, Literal ma, Literal nma) {
    LocalClause result = (LocalClause) lc1.union(lc2);
    result.remove(ma);
    result.remove(nma);
    return result;
}
```

5.7.2 Memorizzazione delle coppie di clausole visitate

La gestione delle coppie di clausole visitate viene fatta nello stesso modo in cui veniva svolto nell'implementazione per la logica classica (si veda la sezione 3.3.2). Si utilizza quindi la mappa `visited`, che memorizza inizialmente le coppie di clausole sulle quali vengono applicate le regole *LRES* e *GRES* tramite i loro indici. Dopo che sono state applicate queste due regole su tutte le coppie possibili, la mappa `visited` viene svuotata, in questo modo sarà possibile eseguire le regole *LERES* e *GERES* anche sulle coppie di clausole alle quali sono state applicate in precedenza le altre due regole. Si noti che l'applicazione della regola *G2L* non viene memorizzata.

Come nell'implementazione per la logica classica, viene utilizzato il metodo `alreadyVisited` per verificare se una coppia di clausole è già stata visitata in precedenza; inoltre, per aggiungere una coppia alla mappa `visited` viene utilizzato il metodo `addCoupleInMap`

5.7.3 Gestione degli step

La gestione degli step viene svolta come nel caso della logica classica (si veda la sezione 3.3.4, nella quale vengono spiegati anche i metodi `setEnableSteps` e `printTrace`). In questo caso nella classe `Step` viene aggiunto, tra gli altri, un campo che specifica il nome della regola di risoluzione applicata nello step, e poi nel caso delle regole *LERES* e *GERES*, vengono aggiunte anche le premesse $G(C)$ e $G(C')$. La regola *G2L* non viene presa in esame dagli step, perché è considerata sottintesa.

Codice 5.5: cicli for innestati per l'esecuzione delle regole LERES e GERES nel metodo `isSatisfiable`

```
for (int i = 0; i < listCl.size(); i++) {
    Clause c1 = listCl.get(i);
    int index1 = c1.getIndex();
    for (int j = 0; j < listCl.size(); j++) {
        Clause c2 = listCl.get(j);
        int index2 = c2.getIndex();
        if ((i != j) && (c1.getClass().equals(c2.getClass())) && !alreadyVisited(c1, c2)) {
            Literal[] literals = getModalLiterals(c1, c2, listCl);
            if (literals.length != 0) {
                addCoupleInMap(index1, index2);
                Clause resolvent = null;
                if ((c1 instanceof GlobalClause) && (c2 instanceof GlobalClause)) {
                    resolvent = GERES((GlobalClause) c1, (GlobalClause) c2, literals[0], literals[1]);
                    //gestione step
                } else {
                    resolvent = LERES((LocalClause) c1, (LocalClause) c2, literals[0], literals[1]);
                    //gestione step
                }
                if (resolvent.isEmpty()) {
                    return false;
                }
                if (resolvent.isTautology()) {
                    //gestione step
                } else if (listCl.contains(resolvent)) {
                    //gestione step
                } else {
                    visited.put(resolvent.getIndex(), new HashSet<>());
                    listCl.add(resolvent);
                }
            }
        }
    }
}
```

Capitolo 6

Applicazioni pratiche del metodo di risoluzione

Il metodo di risoluzione ha numerose applicazioni pratiche, tra cui:

- **Verifica della validità di formule**: Determinare se una formula è valida all'interno di un dato sistema logico.
- **Dimostrazione automatica di teoremi**: Utilizzare algoritmi di risoluzione per automatizzare il processo di dimostrazione di teoremi.
- **Verifica formale**: Applicare metodi di risoluzione per verificare formalmente proprietà di sistemi hardware e software.
- **Soddisfacibilità booleana (SAT)**: Utilizzare il metodo di risoluzione per risolvere problemi di soddisfacibilità booleana, che hanno applicazioni in vari campi, tra cui l'intelligenza artificiale, la teoria dei giochi e la verifica del software.

L'utilizzo pratico che viene implementato nel nostro caso è quello della verifica della validità di formule logiche.

Definizione 6.0.1. *Una formula F è **valida** se e solo se è una tautologia, ossia è vera in ogni interpretazione di F.*

6.1 Verificare la validità di una formula

Per verificare la validità di una qualsiasi formula F, si considera la sua negazione logica $\neg F$ e si controlla se essa è soddisfacibile o meno; se $\neg F$ è **insoddisfacibile**, allora F è valida. Questo perché, come spiegato nel capitolo 1, se applichiamo la negazione logica ad una tautologia otteniamo una contraddizione, e viceversa.

6.1.1 Dimostrazione di conseguenze logiche

Definizione 6.1.1. *Siano F_1 e F_2 due formule. Si dice che F_2 è conseguenza logica di F_1 se e solo se per ogni interpretazione I, se I soddisfa F_1, allora soddisfa anche F_2. La conseguenza logica si può scrivere con $F_1 \models F_2$.*

Applicando la verifica della validità di formule, è possibile dimostrare la conseguenza logica di due formule F_1 e F_2 utilizzando l'implicazione logica, ovvero il connettivo $\rightarrow$. Infatti, se F_2 è conseguenza logica di F_1, allora la formula:

$$F_1 \rightarrow F_2$$

è valida.

Esempio di conseguenza logica

Un esempio che possiamo considerare è il *modus ponens*: siano a, b due formule, se la formula $a \rightarrow b$ è vera, e anche a è vera, allora la conseguenza b è vera.

$$\frac{a \qquad a \rightarrow b}{b}\ \text{Modus ponens}$$

Per dimostrare il modus ponens, creiamo una formula F utilizzando la congiunzione logica per unire le due premesse (a e $a \rightarrow b$), e l'implicazione logica per rappresentare la conseguenza logica (con b). La formula F ottenuta è la seguente:

$$F \ = \ (a \wedge (a \rightarrow b)) \rightarrow b$$

A questo punto dimostriamo la validità di F. Bisogna quindi verificare che la formula $\neg F$ è insoddisfacibile. Per farlo, dobbiamo clausificare $\neg F$ ed eseguire il metodo di risoluzione sull'insieme di clausole ottenuto. La clausificazione di $\neg F$ produce il seguente insieme di clausole:

$$\{ \quad \{a\}; \{\neg a, b\}; \{\neg b\} \quad \}$$

applicando due volte la regola *Res* si trova facilmente la clausola vuota. Questo dimostra che il modus ponens è valido nella logica classica.

6.1.2 Dimostrazione di equivalenze logiche

Definizione 6.1.2. *Due formule F_1 e F_2 sono logicamente equivalenti se e solo se $F_1 \models F_2$ e $F_2 \models F_1$, ossia entrambe sono conseguenza logica l'una dell'altra.*

In questo caso, per dimostrare l'equivalenza logica tra due formule F_1 e F_2 si usa la doppia implicazione, ovvero il connettivo $\leftrightarrow$. Infatti se F_1 e F_2 sono logicamente equivalenti, allora la seguente formula:

$$F_1 \leftrightarrow F_2$$

è valida.

Esempio di equivalenza logica

Consideriamo la dimostrazione di una delle leggi di De Morgan, in altre parole, dimostriamo che la formula $\neg(a \wedge b)$ è logicamente equivalente alla formula $\neg a \vee \neg b$. Per farlo dobbiamo verificare la validità della seguente formula F:

$$F \ = \ \neg(a \wedge b) \leftrightarrow (\neg a \vee \neg b)$$

Procediamo ora con la clausificazione di $\neg F$ ed ottieniamo il seguente insieme di clausole S:

$$S \ = \ \{ \quad \{a, b\}; \{\neg a, \neg b\}; \{a\}; \{b\} \quad \}$$

A questo punto applichiamo il metodo di risoluzione su S, che ricava facilmente la clausola vuota. Questo dimostra che $\neg F$ è insoddisfacibile, e che quindi F è valida.

6.2 La classe `App`

L'implementazione di quanto visto in questo capitolo, viene svolta nella classe `App`. Questa classe è presente sia nel sistema per la logica classica, che in quello per le logiche modali. Come spiegato nella sezione 3.1, la classe `App` contiene soltanto il metodo `main`; esso calcola la validità di una formula letta da *Standard input*. Il codice 6.1 mostra questo metodo.

Il metodo `main` esegue le seguenti operazioni:

- Legge da *Stdin* una formula in formato testuale, ed effettua il *parsing* su di essa, ottenendo così l'istanza di `Formula f`.
- Applica la negazione logica su `f`, ottenendo l'istanza di `CompoundFormula not_f`.
- Esegue la clausificazione della formula `not_f`, ottenendo l'istanza di `ClauseSet cnf`, e la stampa su *Standard Output*.
- Se sulla riga di comando è presente il comando "**-v**", abilita la stampa degli step di risoluzione, richiamando il metodo `setEnableSteps` della classe `Resolution`, passandogli come parametro `true`.

- Applica il metodo di risoluzione sull'insieme di clausole `cnf`, tramite il metodo `isSatisfiable` della classe `Resolution`. Se il metodo restituisce `true`, allora stampa su *Stdout* la stringa "*The formula is NOT a tautology*"; se restituisce `false`, stampa la stringa "*The formula is a tautology*".

Codice 6.1: Metodo `main` della classe `App`

```java
public static void main(String[] args) {
    Scanner sc = new Scanner(System.in);
    String formulaStr = sc.nextLine();
    sc.close();
    Formula f = ParseFormula.parse(formulaStr);
    if (f == null) {
        System.out.println("\nYour formula in input is not a well-formed formula");
        return;
    }
    System.out.println("\nYour formula in input:");
    System.out.println(f);
    Formula not_f = new CompoundFormula(NOT, f);
    ClauseSet cnf = not_f.toCnf();
    System.out.println("\nThe corresponding clause set of the negation is:");
    System.out.println(cnf);
    System.out.println();
    if (args.length != 0 && args[0].equals("-v")) {
        Resolution.setEnableSteps(true);
    }
    if (Resolution.isSatisfiable(cnf)) {
        System.out.println("The formula is not a tautology");
    } else {
        System.out.println("The formula is a tautology");
    }
}
```

Capitolo 7

Testing

Per effettuare Il testing del metodo di risoluzione, è stato utilizzato il framework *Junit* [BG24], che consente di implementare test unitari in Java. Nella struttura del progetto presente nella sezione 3.1, si trova il package *test* che contiene la classe `ResolutionTest` e vari file di testo (*txt*) che contengono formule, una per ogni file. I file di testo nel package *test* si dividono in formule valide (*taut*) e in formule non valide (*non-taut*); ad esempio il file *taut1.txt* contiene una formula valida, ed il file *non-taut1.txt* contiene invece una formula non valida.

7.1 La classe `ResolutionTest`

La classe `ResolutionTest` contiene i metodi di test, ed il metodo `readFile`.

Il metodo `readFile` prende in input una stringa che identifica il nome di un file *txt* e restituisce il contenuto di quel file.

I metodi di test verificano che le formule valide contenute nei file *txt*, siano effettivamente valide, e che le formule non valide siano effettivamente tali. Il codice 7.1 mostra il metodo che testa la validità della formula presente nel file *taut1.txt*. Il metodo legge il file *taut1.txt* ed esegue il *parsing* sul suo contenuto, in modo da ottenere la formula `f`, successivamente applica la negazione logica su `f` memorizzandola sulla variabile `not_f`, ed infine asserisce che il metodo `isSatisfiable` applicato sulla clausificazione di `not_f` restituisca `false`. Per fare questa asserzione viene utilizzato il metodo `assertFalse` di *Junit*; se nel metodo `assertFalse` viene passato il parametro `true`, il test fallisce, altrimenti il metodo di test ha successo. Il test della classe `ResolutionTest` ha successo se e solo se tutti i suoi metodi di test hanno successo. Il codice 7.2 mostra invece il metodo che testa la non validità della formula contenuta nel file *non-taut1.txt* (si noti che in questo caso viene usato il metodo `assertTrue`).

Codice 7.1: Metodo che testa la validità della formula nel file *taut1.txt*

```
@Test
public void testTaut1() {
    String taut1 = readFile("taut1.txt");
    if (taut1 == null) {
        fail("The file doesn't exist");
    }
    Formula f = ParseFormula.parse(taut1);
    if (f == null) {
        fail("Parsing error of formula f");
    }
    Formula not_f = new CompoundFormula(NOT, f);
    assertFalse(Resolution.isSatisfiable(not_f.toCnf()));
}
```

Codice 7.2: Metodo che testa la non validità della formula nel file *non-taut1.txt*

```
@Test
public void testNonTaut1() {
    String nt = readFile("non-taut1.txt");
    if (nt == null) {
        fail("The file doesn't exist");
    }
    Formula f = ParseFormula.parse(nt);
    if (f == null) {
        fail("Parsing error of formula f");
    }
    Formula not_f = new CompoundFormula(NOT, f);
    assertTrue(Resolution.isSatisfiable(not_f.toCnf()));
}
```

Capitolo 8

Conclusioni

In questo elaborato, abbiamo esplorato la logica proposizionale e le logiche modali non-normali, concentrandoci sui metodi di risoluzione e alla loro implementazione in Java. Abbiamo anche esaminato ed implementato la conversione di formule logiche in forma normale congiuntiva (CNF) e il parsing. In questo capitolo finale, riassumiamo i principali contributi del lavoro svolto e suggeriamo possibili direzioni future di ricerca.

8.1 Riassunto del lavoro svolto

Abbiamo iniziato con una panoramica della logica proposizionale, introducendo le formule atomiche e composte. Successivamente, abbiamo descritto il metodo di risoluzione per la logica classica, spiegando la forma normale congiuntiva e l'applicazione della regola di risoluzione *Res* per determinare la soddisfacibilità di insiemi di clausole.

L'implementazione in Java del metodo di risoluzione è stata dettagliata nei capitoli successivi, dove abbiamo descritto le strutture dati utilizzate per rappresentare letterali, clausole e insiemi di clausole. Abbiamo anche discusso l'implementazione della regola di risoluzione e la memorizzazione delle coppie di clausole visitate.

Un'altra parte importante del nostro lavoro è stata l'implementazione delle formule generiche in Java, includendo la loro conversione in *CNF* e l'utilizzo del parser *ANTLR4* per il parsing delle formule. Infine, abbiamo esteso il metodo di risoluzione alle logiche modali non-normali, discutendo anche la clausificazione di formule in questi tipi di logiche. Abbiamo poi visto le applicazioni pratiche ed infine il testing.

8.2 Direzioni future di ricerca

Il lavoro svolto in questo elaborato apre diverse direzioni per future ricerche:

- **Ottimizzazione degli algoritmi di risoluzione**: Migliorare l'efficienza degli algoritmi di risoluzione, sviluppando nuove tecniche di ottimizzazione.
- **Estensione a logiche più complesse**: Applicare i metodi di risoluzione a logiche modali non-normali più complesse rispetto alla logica minimale E che abbiamo visto.
- **Integrazione con altri strumenti**: Integrare il nostro sistema di risoluzione con altri strumenti di verifica formale e dimostrazione automatica.
- **Applicazioni pratiche**: Sperimentare l'applicazione dei metodi di risoluzione in contesti reali, come la verifica di sistemi critici e la risoluzione di problemi di ottimizzazione.

In conclusione, il metodo di risoluzione rimane un potente strumento per l'analisi e la dimostrazione di proprietà logiche. La sua implementazione in Java e l'estensione alle logiche modali rappresentano passi significativi verso l'automazione della logica e l'applicazione pratica di questi concetti teorici.

Bibliografia

[BG24] Kent Beck and Erich Gamma, *Junit 5*, `https://junit.org/junit5/`, 2024.

[Lei97] Alexander Leitsch, *The resolution calculus*, Texts in Theoretical Computer Science. An EATCS Series, Springer, 1997.

[LG01] Barbara Liskov and John V. Guttag, *Program development in java - abstraction, specification, and object-oriented design*, Addison-Wesley, 2001.

[May11] Marta Cialdea Mayer, *Introduzione alla logica proposizionale*, `https://www.inf.uniroma3.it/~cialdea/teaching/pf/materiale/logica.pdf`, 2010/2011.

[Ora24] Oracle Corporation, *Java programming language*, `https://www.oracle.com/java/`, 2024.

[Par13] Terence Parr, *The definitive antlr 4 reference*, 2 ed., Pragmatic Bookshelf, Raleigh, NC, 2013.

[PON23] Dirk Pattinson, Nicola Olivetti, and Cláudia Nalon, *Resolution calculi for non-normal modal logics*, Automated Reasoning with Analytic Tableaux and Related Methods - 32nd International Conference, TABLEAUX 2023, Prague, Czech Republic, September 18-21, 2023, Proceedings (Revantha Ramanayake and Josef Urban, eds.), Lecture Notes in Computer Science, vol. 14278, Springer, 2023, pp. 322–341.

[Pro24] ANTLR Project, *Antlr (another tool for language recognition)*, `https://www.antlr.org/`, 2024.

Ringraziamenti

Dedico questo spazio finale della mia tesi di Laurea, alla mia famiglia e a tutte le persone che durante il mio percorso universitario mi hanno sostenuto e compreso.

Un grazie ai miei amici di Università, Manuele, Samuele, Davide, Stefano, Michele, Filippo, Enrico, che hanno reso il mio percorso universitario più spensierato.

Un grazie al mio relatore, Camillo Fiorentini che mi ha seguito per la realizzazione di questo elaborato finale.

E un grazie speciale ai miei genitori, a mio fratello Lorenzo e mia nonna, che con la loro comprensione, il loro sostegno e il loro affetto mi hanno aiutato a superare momenti più difficili e ad arrivare alla fine di questo percorso stanco ma soddisfatto.

Ringrazio anche me stesso che sono riuscito a superare difficoltà, ma con tenacia e costanza sono arrivato al traguardo che mi ero prefissato.