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

Dediche

dedicato a ...

Prefazione

...

Indice

Capitolo 1

Introduzione

1.1 Panoramica generale

Questo elaborato riguarda l'ambito della logica matematica, concentrandosi sul metodo di risoluzione per la logica classica, ovvero la logica proposizionale. Nel capitolo 5 verrà trattato il metodo di risoluzione per le logiche modali non-normali, che rappresentano un'estensione di quella proposizionale. Di seguito viene presentata una breve introduzione della logica proposizionale, utile a capire il funzionamento del metodo di risoluzione, trattato nel capitolo 2.

1.2 Logica proposizionale

La logica proposizionale è un sistema formale per la rappresentazione e l'analisi del ragionamento, essa si basa su proposizioni. La sua sintassi comprende formule atomiche e formule composte. La semantica della logica proposizionale stabilisce come valutare le formule, associando loro valori di verità in base a interpretazioni che specificano lo stato di verità di ogni proposizione atomica.

1.2.1 Formule atomiche

Le formule atomiche rappresentano il caso più semplice di formula, nelle quali non vengono usati gli operatori 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.

1.2.2 Formule composte

Le formule composte sono costruite mediante operatori logici (detti anche "*connettivi*") 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**)

Consideriamo il seguente esempio di formula composta:

$$\neg p \wedge q$$

essa è ottenuta a partire dalla atomica "p" che viene negata tramite la negazione logica ($\neg$), e successivamente messa in congiunzione logica ($\wedge$) con l'atomica "q". Si possono creare formule composte più complicate, in tal caso si usano le parentesi tonde per evitare ambiguità tra i connettivi.

L'interpretazione di una formula composta dipende dai connettivi usati e dall'interpretazione delle sue atomiche.

1.2.3 Soddisfacibilità di una formula

Una formula F si dice **soddisfacibile** se e solo se ammette almeno un'interpretazione che la rende vera, ossia esiste almeno un assegnamento di valori di verità alle atomiche che rende vera F. Se tutte le interpretazioni possibili la rendono falsa, allora F si dice **insoddisfacibile**.

1.2.4 Tautologie e contraddizioni

Una formula è una **tautologia** se e solo se ogni interpretazione della formula la rende vera. Quindi una tautologia è una formula sempre vera, che come vedremo, può essere ridondante in alcuni casi. Ad esempio, la formula "$p \vee \neg p$" è il caso più semplice di tautologia.

Una formula è una **contraddizione** se e solo se è insoddisfacibile, ovvero tutte le sue interpretazioni la rendono falsa. Ad esempio, la formula "$p \wedge \neg p$" è una contraddizione. È bene notare che se si applica la negazione logica ad una contraddizione si ottiene una tautologia, e viceversa.

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.

Questo metodo consiste nell'applicare più volte una sola regola: la regola di risoluzione. Il difetto è che essa opera soltanto su formule espresse in **Forma normale congiuntiva (CNF)**, ovvero come congiunzione di clausole. 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

Un letterale è una formula atomica, oppure la sua negazione. Ad esempio "a" e "$\neg b$" sono dei letterali, essi rappresentano l'elemento fondamentale delle clausole.

Opposto di un letterale

Dato un letterale L, il suo *opposto* $\overline{L}$ è la sua negazione, ad esempio: se $L = p$ allora $\overline{L} = \neg p$ e se $L = \neg p$ allora $\overline{L} = p$.

2.1.2 Clausole

Una clausola è una disgiunzione di 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

Una clausola è una tautologia se e solo se contiene un letterale ed il suo opposto. Ad esempio: "$\{a, b, c, \neg a\}$" è una tautologia, perché contiene il letterale "a" ed il suo opposto. Infatti questo tipo di clausola risulta essere sempre vera, qualunque sia l'interpretazione dei suoi letterali (si veda la sezione 1.2.4).

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 1.2.4) e si può indicare con "{}".

2.1.3 Insiemi di clausole

La *CNF* consiste in 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

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

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 rende vere **tutte** le clausole appartenenti a S (a causa della congiunzione).

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, 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 prova che S è **insoddisfacibile**; se invece, dopo aver applicato *Res* su tutte le possibili coppie di clausole non la trova, allora il metodo prova che S è **soddisfacibile**.

2.3.1 Gestione delle clausole tautologiche

Nella *CNF* le clausole tautologiche rappresentano una ridondanza, perché 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 questo tipo di clausole, 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 tautologia, 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 tautologie, 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:

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

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

- **Step 1:**

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

- **Step 2:**

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

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 $S^{'}$:

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

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

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

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 quindi $S^{'}$ è **soddisfacibile**.

Capitolo 3

Implementazione in Java del metodo di risoluzione

Questo capitolo si concentra sull'implementazione pratica del metodo di risoluzione per la logica classica in 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. 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 questa classe non è contenuta in alcun package).

3.2 Strutture dati per la CNF

I package "**literal**" e "**cnf**" contengono le classi per rappresentare la CNF, fondamentali 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 una variabile atomica e la negazione di essa. 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 atomo negato (ad esempio "$\sim p$").

metodo getOpposite sovrascritto dalla classe Atom

```
1        @Override
2        public Literal getOpposite() {
3            return new NegAtom(this.getName());
4        }
```

3.2.2 La classe Clause

Questa classe permette di rappresentare le clausole della *CNF*, tenendo conto della loro notazione insiemistica. La classe contiene il campo `literals` di tipo `Set<Literal>` che consiste nell'insieme di letterali, inoltre contiene anche il campo `index` che rappresenta 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 3.1). 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 3.1: Metodo isTautology della classe Clause

```
1        public boolean isTautology() {
2            for (Literal l1 : this.literals) {
3                for (Literal l2 : this.literals) {
4                    if (l1.equals(l2.getOpposite())) return
                         true;
5                }
6            }
7            return false;
8        }
```

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 una tautologia 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 3.2). Anche la classe *ClauseSet* segue il design pattern "iterator".

Codice 3.2: Metodo "removeTautologies" della classe ClauseSet

```
1        public void removeTautologies() {
2            List<Clause> tautologies = new ArrayList<>();
3            for (Clause c : this.clauses) {
4                if (c.isTautology()) {
5                    tautologies.add(c);
6                }
7            }
8            for (Clause taut : tautologies) {
9                this.clauses.remove(taut);
10           }
11       }
```

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 i passaggi 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. Di seguito viene mostrato il codice. 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 che sia vuoto (in tal caso viene restituito `true` perché `s` è una tautologia), e in caso negativo continua l'esecuzione inizializzando i campi `visited` e `trace`; inoltre, le clausole in `s` vengono inserite nella lista di clausole `listCl` per poter consentire la modifica della lista durante il suo scorrimento (eseguito dal ciclo *for*).

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 3.3 mostra l'esecuzione dei due cicli.

Codice 3.3: Metodo "isSatisfiable" della classe Resolution

```
1         for (int i = 0; i < listCl.size(); i++) {
2             Clause c1 = listCl.get(i);
3             int index1 = c1.getIndex();
4             for (int j = 0; j < listCl.size(); j++) {
5                 Clause c2 = listCl.get(j);
6                 int index2 = c2.getIndex();
7                 if ((i != j) && !alreadyVisited(c1, c2)) {
8                     Literal complemLit =
                          getComplementaryLiteral(c1, c2);
9                     if (complemLit != null) {
10                        if (index1 < index2) {
11                            (visited.get(index1)).add(index2);
12                        } else {
13                            (visited.get(index2)).add(index1);
14                        }
15                        Clause resolvent = resolRule(c1, c2,
                              complemLit);
16                        Step step = null;
17                        if (enableSteps) {
18                            step = new Step(c1, c2, resolvent,
                                  complemLit);
19                            trace.add(step);
20                        }
21                        if (resolvent.isEmpty()) {
22                            if (enableSteps) printTrace();
23                            return false;
24                        }
25                        if (resolvent.isTautology()) {
26                            if (enableSteps)
27                                step.setTautology();
28                        } else if (listCl.contains(resolvent)) {
29                            if (enableSteps)
30                                step.setAlreadyPresent();
31                        } else {
32                            visited.put(resolvent.getIndex(),
                                  new HashSet<>());
33                            listCl.add(resolvent);
34                        }
35                    }
36                }
37            }
38        }
39        if (enableSteps) printTrace();
40        return true;
41    }
```

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 3.4 mostra questo metodo.

Codice 3.4: Metodo "getComplementaryLiteral" della classe Resolution

```
1        private static Literal
             getComplementaryLiteral(Clause c1, Clause c2) {
2            for (Literal l1 : c1) {
3                for (Literal l2 : c2) {
4                    if (l1.equals(l2.getOpposite())) return
                         l1;
5                }
6            }
7            return null;
8        }
```

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 3.3 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 di risoluzione è già stata applicata su di essa. Per farlo controlla se nell'insieme ottenuto dall'indice più piccolo è presente l'indice più grande (utilizzando il metodo `contains`). Questo garantisce che l'algoritmo possa terminare. Il codice 3.5 mostra questo metodo.

Codice 3.5: Metodo "alreadyVisited" della classe Resolution

```
1        private static boolean alreadyVisited(Clause c1,
             Clause c2) {
2            int i1 = c1.getIndex();
3            int i2 = c2.getIndex();
4            if (i1 < i2) {
5                return (visited.get(i1)).contains(i2);
6            }
7            return (visited.get(i2)).contains(i1);
8        }
```

3.3.3 Implementazione della regola di risoluzione

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; infine restituisce il risultato. Il codice 3.6 mostra questo metodo.

Codice 3.6: Metodo "resolRule" della classe Resolution

```
1        private static Clause resolRule(Clause c1, Clause
             c2, Literal lit) {
2            Clause result = c1.union(c2);
3            result.remove(lit);
4            result.remove(lit.getOpposite());
5            return result;
6        }
```

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 di risoluzione. 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 3.3, 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:

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

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	|
IMPLIES	$->$
IFF	$<->$

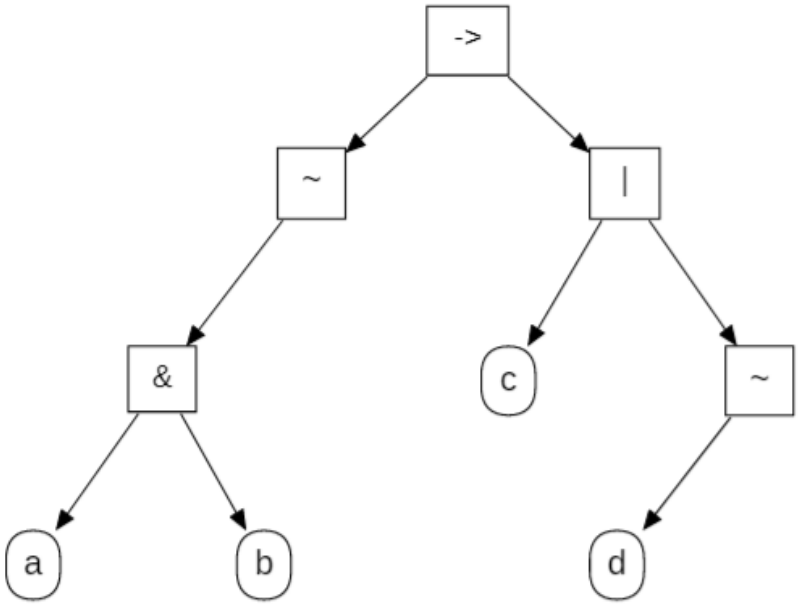
4.4 La classe CompoundFormula

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**:

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

- **Eliminazione dell'implicazione**:

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

- **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 nel metodo astratto `toCnf` della classe `Formula`, che restituisce un `ClauseSet`. Questo metodo ha quindi 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:

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

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:

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

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 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*. 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

```
1  grammar FormulaExpression;
2
3  start : formula EOF;
4
5  formula : atomic_formula
6          | '(' formula ')'
7          | unary_connective formula
8          | formula binary_connective formula
9          ;
10
11 atomic_formula : LITERAL ;
12
13 unary_connective : NOT ;
14
15 binary_connective : AND | OR | IMPLIES | IFF;
16
17 //token
18 AND : '&' ;
19 OR : '|' ;
20 IMPLIES : '->' ;
21 NOT : '~' ;
22 IFF : '<->';
23
24 LITERAL : [a-z]+ ;
25 WS : [ \t\r\n]+ -> skip ;
```

In questa grammatica ci sono le regole che definiscono le formule atomiche e composte,

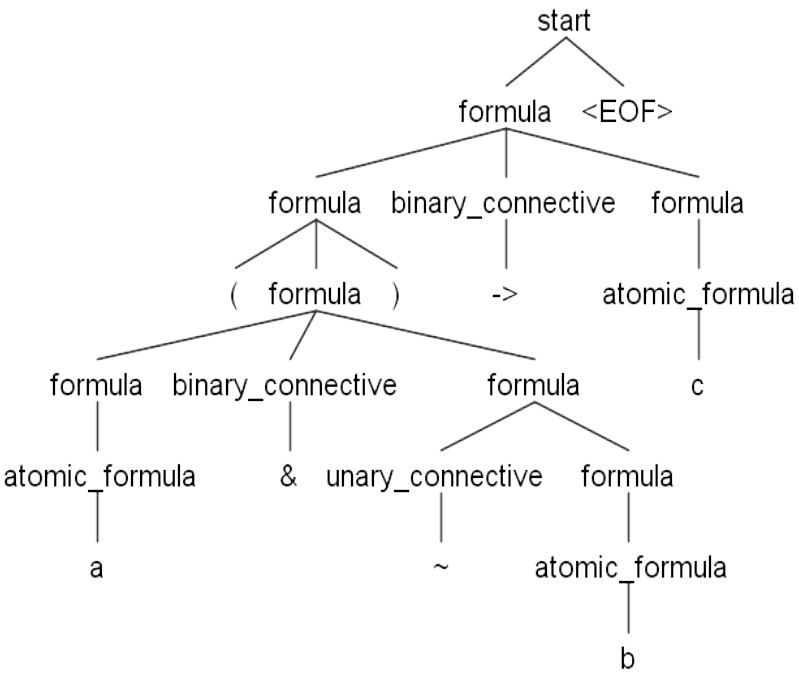
e le regole che descrivono i connettivi unari e binari; le sottoformule sono distinguibili grazie alle parentesi tonde, che specificano anche la precedenza tra i vari connettivi della formula. in fondo al file è presente la definizione dei token. È bene notare che la rappresentazione testuale dei connettivi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

Esempio di *AST*

Consideriamo la seguente 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 che estende il Base Listener; 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`

```
1  public static Formula parse(String formulaStr) {
2      CharStream input =
           CharStreams.fromString(formulaStr);
3      FormulaExpressionLexer lexer = new
           FormulaExpressionLexer(input);
4      CommonTokenStream tokens = new
           CommonTokenStream(lexer);
5      FormulaExpressionParser parser = new
           FormulaExpressionParser(tokens);
6
7      //create listener
8      FormulaListenerImplementation listener = new
           FormulaListenerImplementation();
9
10     //adds the listener to the parser
11     parser.addParseListener(listener);
12
13     try {
14         //analyze the input and get the
               corresponding formula
15         parser.start();
16         return listener.getFormula();
17     } catch (Exception e) {
18         return null;
19     }
20 }
```

Capitolo 5

Logiche modali non-normali ed estensione del metodo di risoluzione

Questo capitolo si concentra sulle logiche modali e su come estendere il metodo di risoluzione per questo tipo di logiche. La prima parte del capitolo illustra le nozioni teoriche, la seconda parte riguarda l'implementazione in Java di quello che viene spiegato nella parte teorica.

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**.

Le formule delle logiche modali (normali e non-normali) utilizzano, oltre ai connettivi della logica classica, il connettivo *box* ($\Box$); 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.

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.

5.2.1 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.

5.2.2 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. Anche in questo caso è possibile utilizzare la notazione insiemistica per rappresentare le clausole.

5.3 Clausificazione di formule modali

Per poter effettuare la clausificazione di una formula ϕ è necessario utilizzare una 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 $\mathbf{R}$, ossia la funzione di *riduzione*. 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 ottenuti utilizzando i connettivi *or* e *not*.

5.3.1 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 5.7.1, che riguarda l'implementazione, 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 alla logica da considerare (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.

5.4.1 Regole di risoluzione per la logica modale *E*

Nelle regole RES_E che vediamo ora, C e D sono clausole, l sono letterali, e p sono variabili proposizionali. 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.

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

Nella sezione 5.8 vedremo un esempio di applicazione di queste regole.

5.5 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.

5.6 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(·)").

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 indistintamente proposizionali o modali.

5.7 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).

5.7.1 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 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 $\{\eta(this)\}$, ed infine restituisce l'unione con il `ClauseSet` ottenuto da $R(G(\eta(this) \leftrightarrow this))$. Il codice 5.1 mostra questo metodo.

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

```
1          @Override
2          public ClauseSet toClauseSet() {
3              if (this.isClassic()) {
4                  return this.classicClausification();
5              } else {
6                  eta = new Eta(this); //eta e' il campo
                       statico della classe Formula.
7                  PropAtom t = eta.getPropVariable(this);
8                  ClauseSet cs = new ClauseSet(new
                       LocalClause(t));
9                  return cs.union(this.R(t));
10             }
11         }
```

Il metodo `R`

Il metodo `classicClausification`

Parsing di formule modali

5.8 Implementazione del metodo di risoluzione

Capitolo 6

Applicazioni pratiche del metodo di risoluzione

6.1 Verificare la validità di una formula

6.2 Dimostrazione di conseguenze logiche

6.3 Dimostrazione di equivalenze logiche

Capitolo 7

Testing

7.1 Struttura del file di test

Capitolo 8

Conclusioni

Bibliografia

[1] ...

Ringraziamenti

...