Università degli Studi di Napoli Federico II
Scuola Politecnica e delle Scienze di Base
Dipartimento di Ingegneria Elettrica e Tecnologie dell'Informazione

Corso di Laurea in Informatica

Implementazione di una procedura di decisione per frammenti Binding in Vampire

Relatori
Prof. Massimo Benerecetti
Prof. Fabio Mogavero

Candidato
Matteo Richard Gaudino

Matricola
N86003226

Anno Accademico 2022 - 2023

Indice

Introduzione

Capitolo 1

Logica e automazione dei problemi di Decisione

In questo capitolo verranno descritte le nozioni di base necessarie per comprendere il lavoro svolto in questa tesi. In particolare, verranno introdotti i concetti di logica proposizionale e del primo ordine, definita come estensione della prima. Verrà anche introdotto il problema della decisione, ovvero il problema di stabilire se una data formula è soddisfacibile o meno e le principali tecniche ad alto livello che vengono utilizzate dai theorem prover moderni per risolvere questo problema. Nell'ultimo paragrafo del capitolo verrà descritto in che modo le formule di logica del primo ordine possono essere rappresentate in un formato di file, per poi essere processate come input da un theorem prover. Lo scopo di questo capitolo è quindi quello di accennare la teoria logica utilizzata nell'implementazione di vampire e della procedura di decisione per i Binding-Fragments. Non è tra gli obbiettivi dare una trattazione esaustiva sulla logica o in generale sulle varie teoria matematiche coinvolte, perciò verranno date per scontate nozioni di teoria degli insiemi, algebra, teoria della computazione e teoria dei linguaggi.

1.1 Logica Proposizionale

La logica proposizionale è un ramo della logica formale che si occupa dello studio e della manipolazione delle proposizioni, ovvero dichiarazioni che possono essere classificate come vere o false, ma non entrambe contemporaneamente (Principio di bivalenza). Essa fornisce un quadro formale per analizzare il ragionamento deduttivo basato su connettivi logici, come "e", "o", e "non". Questo strumento anche se incluso nella logica del primo ordine, rimane un pilastro fondamentale per lo studio svolto in questa tesi e quindi è necessario farne un'introduzione indipendente.

1.1.1 Formule

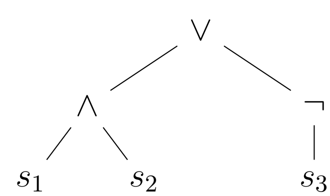
Sia $\Sigma_s = \{s_1, s_2, ...\}$ un insieme di simboli di costante, $\Sigma = \{\wedge, \vee, \neg, (,), \top, \bot\} \cup \Sigma_s$ è detto alfabeto della logica proposizionale. Con queste premesse si può definire come formule della logica proposizionale il linguaggio F generato dalla grammatica Context Free seguente:

$$\varphi := \top \mid \bot \mid S \mid \neg\varphi \mid (\varphi \wedge \varphi) \mid (\varphi \vee \varphi)$$

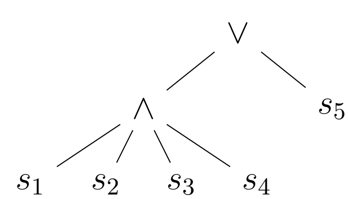
Dove $S \in \Sigma_s$ è un simbolo di costante. Con la funzione $const(\gamma) \rightarrow 2^{\Sigma_s}$ si indica la funzione che associa a ogni formula γ l'insieme dei suoi simboli di costante. Viene chiamato *Letterale*, ogni simbolo di costante s o la sua negazione $\neg s$. Vengono inoltre introdotti i seguenti simboli come abbreviazioni:

- $(\gamma \Rightarrow \kappa)$ per $(\neg\gamma \vee \kappa)$
- $(\gamma \Leftrightarrow \kappa)$ per $((\gamma \Rightarrow \kappa) \wedge (\kappa \Rightarrow \gamma))$
- $(\gamma \oplus \kappa)$ per $\neg(\gamma \Leftrightarrow \kappa)$

È possibile rappresentare una qualunque formula attraverso il proprio albero di derivazione. Questo albero verrà chiamato in seguito anche *albero sintattico* della formula. Ad esempio, la formula $(s_1 \wedge s_2) \vee \neg s_3$ può essere rappresentata dal seguente albero sintattico:



La radice dell'albero è detta *connettivo principale* e i sotto alberi della formula vengono dette *sottoformule*. Per compattezza, grazie alla proprietà associativa di $\wedge$ e $\vee$, è possibile omettere le parentesi, es. $(s_1 \wedge (s_2 \wedge (s_3 \wedge s_4))) \vee s_5$ può essere scritto come $(s_1 \wedge s_2 \wedge s_3 \wedge s_4) \vee s_5$. Allo stesso modo, nell'albero sintattico della formula è possibile compattare le catene di $\wedge$ e $\vee$ come figli di un unico nodo:



Questa è una caratteristica molto importante, in quanto non solo permette di risparmiare inchiostro, ma consente di vedere $\wedge$ e $\vee$ non più come operatori binari ma come operatori n-ari. A livello implementativo, ciò si traduce in un minor impatto in memoria, visite all'albero più veloci e algoritmi di manipolazione più semplici. Si consideri ad esempio di voler ricercare la foglia più a sinistra nell'albero di derivazione della seguente formula $((...(((s_1 \wedge s_2) \wedge s_3) \wedge s_4) \wedge ...) \wedge s_n)$. Senza compattazione, l'algoritmo di ricerca impiegherebbe $O(n)$ operazioni, mentre con la compattazione $O(1)$.

1.1.2 Assegnamenti

Un *assegnamento* è una qualunque funzione α da un insieme $S \subseteq \Sigma_s$ nell'insieme $\{1, 0\}$ (o $\{True, False\}$).

$$\alpha : S \to \{1, 0\}$$

Un assegnamento α è detto *appropriato* per una formula $\varphi \in F$ se e solo se $const(\varphi) \subseteq dom(\alpha)$.

Si definisce la relazione binaria di *Soddisfacibilità*:

$$\models \subseteq \{1, 0\}^S \times F$$

In modo tale che dato un assegnamento α appropriato a una formula φ, si dice che $\alpha \models \varphi$ (α soddisfa φ) o anche α è un assegnamento per φ o se e solo se:

- Se φ è una costante s allora $\alpha \models \varphi$ sse $\alpha(s) = 1$
- Se φ è $\top$ allora $\alpha \models \varphi$
- Se φ è $\bot$ allora $\alpha \not\models \varphi$
- Se φ è della forma $\neg\psi$ (dove ψ è una formula) allora $\alpha \models \varphi$ sse $\alpha \not\models \psi$
- Se φ è della forma $(\psi \wedge \chi)$ (con ψ e χ formule) allora $\alpha \models \varphi$ sse $\alpha \models \psi$ e $\alpha \models \chi$
- Se φ è della forma $(\psi \vee \chi)$ (con ψ e χ formule) allora $\alpha \models \varphi$ sse $\alpha \models \psi$ o $\alpha \models \chi$

Per convenzione si assume che $\alpha(\top) = 1$ e $\alpha(\bot) = 0$ per ogni assegnamento α. Una *Tautologia* è una formula φ tale che per ogni assegnamento α appropriato a φ, $\alpha \models \varphi$ (in simboli $\models \varphi$). Una formula è detta soddisfacibile se esiste un assegnamento appropriato che la soddisfa altrimenti è detta insoddisfacibile. Date due formule φ e ψ, si dice che ψ è *conseguenza logica* di φ (in simboli $\varphi \models \psi$) se e solo se per ogni assegnamento α appropriato a entrambe le formule, se $\alpha \models \varphi$ allora $\alpha \models \psi$. Due formule sono dette *equivalenti* sse $\varphi \models \psi$ e $\psi \models \varphi$ (in simboli $\varphi \equiv \psi$). Un'importante proprietà è che se $\varphi \models \psi$ allora la formula $\varphi \Rightarrow \psi$ è una tautologia ($\models \varphi \Rightarrow \psi$).

Due concetti molto simili a quello di equivalenza e conseguenza logica sono l'*equisoddisfacibilità* e la *soundness*. In pratica, due formule sono sound se e solo se, se la prima formula è soddisfacibile allora lo è anche la seconda. Due formule sono equisoddisfacibili se e solo se sono sound in entrambe le direzioni.

Quindi la conseguenza logica implica la soundness ma non il viceversa. Allo stesso modo l'equivalenza logica implica l'equisoddisfacibilità ma non il viceversa. Si consideri ad esempio le due formule $\varphi = s_1$ e $\psi = \neg s_1$. Ovviamente non può esserci conseguenza logica tra le due formule, ma sono equisoddisfacibili, infatti se α è un assegnamento per φ allora è possibile costruire un assegnamento β per ψ tale che $\beta(s_1) = 1 - \alpha(s_1)$ e viceversa.

Un'*inferenza* è una qualunque relazione da $F^n \times F$. Un'inferenza è detta *corretta* se conserva la soddisfacibilità, ovvero se a a partire da formule soddisfacibili non associa formule insoddisfacibili (soundness). Un esempio di inferenza è la regola del *Modus Ponens*. Date le premesse $\varphi, (\varphi \Rightarrow \psi)$ si può inferire ψ. Viene utilizzato il simbolo $P \vdash C$ per indicare l'applicazione di un'inferenza alla *Premessa* P per ottenere una *Conclusione* C. In notazione la regola del modus ponens viene espressa così: $[\varphi, \varphi \Rightarrow \psi] \vdash \psi$ oppure $\frac{\varphi, \varphi \Rightarrow \psi}{\psi}$.

Infine, si definisce *Implicante* di una formula φ un insieme I di letterali di φ che rendono vera φ. Cioè, costruendo una assegnazione α tale che $\alpha \models c$ per ogni letterale $c \in I$, si ha che $\alpha \models \varphi$. In altre parole la formula costruita dalla congiunzione di tutti i letterali di I implica logicamente φ. Spesso con abuso di terminologia gli elementi di I vengono chiamati anch'essi implicanti, di solito è facile intuire dal contesto se si sta parlando dell'insieme o dei letterali. È possibile anche costruire un Implicante a partire da una assegnazione. È sufficiente prendere l'insieme dei letterali della formula soddisfatti dall'assegnamento e si ottiene così un implicante.

1.1.3 Forme Normali

Una delle strategie più utilizzate dai dimostratori di teoremi automatici è la *normalizzazione* delle formule. Una *forma normale* è essenzialmente un sottoinsieme di F che rispetta determinate proprietà. Una *normalizzazione* invece è il processo di trasformazione di una formula tramite una successione d'inferenze (corrette) in una forma normale. In questo paragrafo verranno descritte le tre forme normali che sono state utilizzate per il preprocessing dell'algoritmo. In questo caso, tutte e tre le forme presentate preservano la relazione di equivalenza logica, quindi è sempre possibile trasformare una formula in un'altra equivalente in uno di questi tre formati. La prima e l'ultima ossia le forme NNF e CNF sono le più famose e utilizzate, mentre la seconda, la ENNF, non è abbastanza conosciuta da essere definita standard e viene utilizzata per bypassare alcuni problemi di efficienza causati dalla CNF grazie all'utilizzo di tecniche di Naming, che però verranno discusse nella prossima sezione.

La prima tra queste è la *NNF* ossia *Negated Normal Form* (Forma normale negata). Una formula è in formato NNF sse non contiene connettivi semplificati

($\Rightarrow$, $\Leftrightarrow$, $\oplus$) e la negazione è applicata solo a letterali. La classe di formule NNF è generata dalla seguente grammatica:

$$\eta := \top \mid \bot \mid S \mid \neg S \mid (\eta \wedge \eta) \mid (\eta \vee \eta)$$

Dove $S \in \Sigma_s$ è un simbolo di costante. La normalizzazione di una formula in NNF è un processo semplice che consiste nell'applicare opportunamente le regole di De Morgan e le regole di semplificazione dei connettivi.

La seconda forma normale è la *ENNF* ossia *Extended Negated Normal Form* (Forma normale negata estesa). Il formato ENNF è essenzialmente una classe più permissiva della NNF, in quanto conserva il vincolo sulla negazione ma vieta esclusivamente l'uso di '$\Rightarrow$'. La classe di formule ENNF è generata dalla seguente grammatica:

$$\overline{\eta} := \top \mid \bot \mid S \mid \neg S \mid (\overline{\eta} \wedge \overline{\eta}) \mid (\overline{\eta} \vee \overline{\eta}) \mid (\overline{\eta} \Leftrightarrow \overline{\eta}) \mid (\overline{\eta} \oplus \overline{\eta})$$

La terza e ultima forma normale è la *CNF* ossia *Conjunctive Normal Form* (Forma normale congiuntiva). Una formula è in formato CNF sse è una congiunzione di disgiunzioni di letterali. La classe di formule CNF è generata dalla seguente grammatica:

$$\zeta := \xi \mid (\xi \wedge \zeta)$$
$$\xi := \top \mid \bot \mid S \mid \neg S \mid (\xi \vee \xi)$$

La classe CNF è storicamente la più famosa e utilizzata, in quanto è la più semplice da implementare e da manipolare. È possibile vedere le clausole come insiemi di letterali mentre la formula principale è vista come un insieme di clausole. Ad esempio, la CNF $(s_1 \vee \neg s_2) \wedge (s_3)$ può essere rappresentata in termini insiemistici come $\{\{s_1, \neg s_2\}, \{s_3\}\}$. La clausola vuota $\{\}$ è una clausola speciale che rappresenta la formula $\bot$, viene spesso raffigurata dal simbolo $\square$. La normalizzazione di una formula in CNF è un processo più complesso rispetto alle altre due forme normali. Non esiste un unica tecnica di normalizzazione, ma una strategia comune è questa:

1. Si trasforma la formula in NNF.
2. Se la formula è del tipo $\varphi_1 \wedge ... \wedge \varphi_n$ allora la struttura principale è già una congiunzione di formule, quindi si procede applicando l'algoritmo sulle sottoformule $\varphi_1, ..., \varphi_n$.
3. Se la formula è del tipo $(\varphi_1 \wedge \varphi_2) \vee \psi_1$ si applica la proprietà distributiva di $\vee$ su $\wedge$ in modo da spingere i connettivi $\vee$ il più possibile in profondità.

Si ottiene così una formula del tipo $(\varphi_1 \vee \psi_1) \wedge (\varphi_2 \vee \psi_1)$ si procede poi ricorsivamente con il punto 2.

Il processo di generazione delle clausole prende il nome di *clausificazione*. Questa tecnica di clausificazione nella peggiore delle ipotesi porta a una generazione di un numero di clausole esponenziale rispetto alla dimensione della formula originale. Ad esempio la formula $(s_1 \wedge s_2) \vee (s_3 \wedge s_4) \vee ... \vee (s_{n-1} \wedge s_n)$ genera esattamente 2^n clausole diverse tutte da n letterali.

1.1.4 Naming

Come già accennato il processo di normalizzazione di una formula può portare a una crescita esponenziale del numero di clausole generate. Si assuma ad esempio di voler clausificare una formula del tipo:

$$\varphi \vee \psi$$

E che φ e ψ generino rispettivamente n e m clausole. Continuando a clausificare con l'algoritmo descritto nel paragrafo 1.1.3, si ottengono $n \cdot m$ clausole. Questo perché la continua applicazione della proprietà distributiva porta a una duplicazione considerevole delle sottoformule. Una possibile tecnica di ottimizzazione è quella del *Naming* [8] anche detto *Renamig*. In generale per *Naming* si intende una qualunque tecnica di rinomina di letterali o sottoformule. In questo caso, per *Naming* si intende la rinomina di sottoformule tramite l'aggiunta di un nuovo simbolo di costante. Per l'esempio precedente si assuma che s_n sia un simbolo di costante non presente nella formula originale. Applicando il *Naming* si ottiene la seguente formula:

$$(\varphi \vee s_n) \wedge (\neg s_n \vee \psi)$$

In questo modo, la clausificazione della formula genera solo $n + m$ clausole al costo dell'aggiunta di un nuovo simbolo di costante. Un discorso simile vale per formule con la doppia implicazione e lo xor, solo che in questo caso anche solo la normalizzazione in NNF può portare a una crescita esponenziale del numero di sottoformule. La trasformazione in NNF e poi in CNF della formula:

$$((...((s_1 \Leftrightarrow s_2) \Leftrightarrow s_3) \Leftrightarrow ...) \Leftrightarrow s_n)$$

Porta alla generazione di 2^{n-1} clausole. Nell'esempio particolare in cui $n = 6$ si ottengono 32 clausole ma con l'introduzione di due nuovi nomie s_7 e s_8 è possibile ottenere la seguente formula:

$$
\begin{gathered}
(s_7 \Leftrightarrow (((s_1 \Leftrightarrow s_2) \Leftrightarrow s_3) \Leftrightarrow s_4)) \\
\wedge \\
(s_8 \Leftrightarrow (s_7 \Leftrightarrow s_5)) \\
\wedge \\
(s_8 \Leftrightarrow s_6)
\end{gathered}
$$

Che genera solo 22 clausole. Il processo per stabilire quale sottoformula rinominare è un problema complesso. Solitamente stabilisce un numero detto *Threshold* (Soglia) che rappresenta il numero massimo di clausole che si è disposti a generare. Se una sottoformula genera un numero di clausole maggiore del Threshold, allora viene rinominata. Ma anche questo è un discorso non banale e non verrà approfondito oltremodo in questo documento. In generale la nuova formula generata dal namig è equisoddisfacibile all'originale.

1.2 Logica del primo ordine

La logica dei predicati rappresenta un'estensione della logica proposizionale, ampliando il suo ambito per trattare in maniera più ricca e dettagliata le relazioni tra gli oggetti e le proprietà delle entità coinvolte. Mentre la logica proposizionale si occupa di proposizioni atomiche, la logica dei predicati introduce i predicati, che sono relazioni che sono vere o false a seconda dell'interpretazione e dalle variabili che contengono. Vengono generalizzati anche i connettivi logici $\wedge$ e $\vee$ tramite i quantificatori universali ed esistenziali $\forall$ e $\exists$ in modo da poter parlare di anche di insiemi di oggetti non finiti.

1.2.1 Termini e Formule

Si introducono tre nuovi insiemi di simboli:

- $\Sigma_f = \{f_1, f_2, ...\}$ insieme di simboli di funzione
- $\Sigma_p = \{p_1, p_2, ...\}$ insieme di simboli di predicato (o relazione)
- $\Sigma_x = \{x_1, x_2, ...\}$ insieme di simboli di variabile

Si definisce la funzione *arity* : $\Sigma_f \cup \Sigma_p \to \mathbb{N}$ che associa ad ogni simbolo di funzione o predicato la sua arità. Funzioni e predicati di arità 0 sono detti rispettivamente *Funzioni costanti* e *Predicati costanti*. I simboli contenuti in $\Sigma_f \cup \Sigma_p$ sono detti *simboli non logici* e ogni suo sottoinsieme è detto *tipo*. Un *termine* è una stringa generata dalla seguente grammatica:

$$\tau := X \mid f(\tau_1, ..., \tau_n)$$

Dove X è un simbolo di variabile e f è un simbolo di funzione tale che $arity(f) = n$. In altre parole:

- Ogni variabile è un termine
- Ogni funzione costante è un termine
- Se $\tau_1, ..., \tau_n$ sono termini e f è un simbolo di funzione di arità n allora $f(\tau_1, ..., \tau_n)$ è un termine

Si indica con T l'insieme di tutti i termini generati dalla grammatica precedente. Verranno chiamati *Atomo* tutte le stringhe del tipo $p(\tau_1, ..., \tau_n)$ dove p è un simbolo di relazione di arità n e $\tau_1, ..., \tau_n$ sono termini. Vengono chiamati *Letterali* tutti gli atomi e la loro negazione. Termini e Letterali sono detti *ground* se non contengono variabili. Come già visto per le formule proposizionali, è possibile rappresentare un termine o un letterale attraverso il proprio albero di derivazione. Ad esempio, il letterale $p_1(f_2(x_1, f_3), f_1)$ può essere rappresentato dal seguente albero sintattico:

$$\begin{tikzcd} & & p \arrow[dl, dash] \arrow[dr, dash] & \\ & f_2 \arrow[dl, dash] \arrow[dr, dash] & & f_1 \\ x_1 & & f_3 & \end{tikzcd}$$

Come intuibile i sottoalberi di un termine sono detti *sottotermini*. Si assuma di avere due letterali $p_1(f_2, f_1(x_1, f_3))$ e $\neg p_1(f_2(f_1(x_1, f_3), f_3), f_3)$ di volerli rappresentare in un unico grafo. Al posto di creare una foresta con due alberi indipendenti, è possibile creare un unica struttura condividendo i sottotermini comuni:

$$\begin{tikzcd} & p_1 \arrow[dl] \arrow[dr] & & & \neg p_1 \arrow[dl] \arrow[ddl] \\ f_2 & & f_1 \arrow[dl] \arrow[dr] & f_2 \arrow[l] \arrow[d] & \\ & x_1 & & f_3 & \end{tikzcd}$$

Una struttura del genere è detta *Prfectly Shared* (Perfettamente condivisa). Nella pratica questa tecnica di condivisione di sottotermini è indispensabile dato che, anche se a un costo per la creazione e la gestione non indifferente, permette un risparmio di memoria e di tempo considerevole. Per effettuare ad esempio un controllo di uguaglianza tra due sottotermini è sufficiente controllare che le due frecce che partono dai termini padre puntino allo stesso sottotermine, senza dover visitare l'intera sotto struttura, rendendo così tale operazione a tempo costante.

A questo punto si definisce finalmente le formule della logica del primo ordine. Prendendo come punto di partenza le formule proposizionali, si definisce alfabeto delle formule del primo ordine l'insieme: $\Sigma' = (\Sigma \setminus \Sigma_s) \cup \Sigma_f \cup \Sigma_p \cup \Sigma_x \cup \{\forall, \exists\}$ e Si definisce come formule del primo ordine il linguaggio F' generato dalla seguente grammatica:

$$\phi := \top \mid \bot \mid A \mid \neg\phi \mid (\phi \wedge \phi) \mid (\phi \vee \phi) \mid \forall x(\phi) \mid \exists x(\phi)$$

Dove A è un atomo e x è un simbolo di variabile. I simboli $\forall$ e $\exists$ sono detti quantificatori universali ed esistenziali. Una variabile x è detta *vincolata* se è contenuta in una formula del tipo $\forall x(\varphi')$ o $\exists x(\varphi')$ altrimenti è detta *libera*. Una formula è detta *enunciato* se non contiene variabili libere. Una formula è detta *ground* se tutti i suoi letterali sono ground.

Per comodità di scrittura è possibile raggruppare catene di quantificatori dello stesso tipo. Ad esempio, la formula $\forall x_1 \forall x_2 \forall x_3 \exists x_4 \exists x_5 \forall x_6 \forall x_7(\phi)$ può essere scritta come $\forall x_1 x_2 x_3 \exists x_4 x_5 \forall x_6 x_7(\phi)$. In modo simile a come visto per $\vee$ e $\wedge$ è possibile vedere $\forall$ e $\exists$ come operatori n-ari che prendono in input n-1 variabili e una formula.

1.2.2 Unificazione

Dato un termine τ (o un letterale), con la scrittura $\tau[x_k/t]$ si indica il termine (o il letterale) ottenuto sostituendo tutte le occorrenze della variabile x_k con il termine t in τ. Ad esempio se $\tau = f_1(x_1, x_2)$ allora $\tau[x_1/f_2] = f_1(f_2, x_2)$. Si può estendere questa notazione in modo da poter sostituire più variabili contemporaneamente. Si definisce come *sostituzione* una qualunque funzione da un insieme di variabili a termini. Dato un termine τ e una sostituzione $\sigma = \{(x_1, t_1), ..., (x_n, t_n)\}$, con la scrittura $\tau[x_1/t_1, ..., x_n/t_n]$ oppure τ^σ si indica il termine ottenuto sostituendo *contemporaneamente* tutte le occorrenze delle variabili $x_1, ..., x_n$ con i termini $t_1, ..., t_n$ in τ. Con contemporaneamente si intende che ogni singola sostituzione viene effettuata sul termine originale, senza essere influenzata dalle sostituzioni precedenti o successive. Ad esempio, se $\tau = f(x_1, x_2)$ allora $\tau[x_1/x_2, x_2/x_1] = f(x_2, x_1)$ e non $f(x_1, x_1)$ che è invece il risultato dell'applicazione sequenziale delle due regole $\tau[x_1/x_2][x_2/x_1]$.

Dati due termini τ_1 e τ_2 si dice che τ_1 è *più generico* di τ_2 o che τ_2 è *più specifico* di τ_1 se e solo se esiste una sostituzione σ tale che $\tau_1^\sigma = \tau_2$. Se esiste una sostituzione σ tale che $\tau_1^\sigma = \tau_2^\sigma$ allora i due termini sono detti *unificabili* e la sostituzione σ è detta *unificatore* dei due termini.

Date due sostituzioni σ_1 e σ_2 si dice che σ_1 è *più generica* di σ_2 o che σ_2 è *più specifica* di σ_1 se e solo se per ogni termine θ, σ_2 è sussunta da σ_1, ossia sse esiste una sostituzione θ tale che $\tau^{\sigma_2} = (\tau^{\sigma_1})^\theta$. Dati due termini unificabili τ_1 e τ_2 si dice che σ_1 è un *MGU* (Most General Unifier) di τ_1 e τ_2 se è la sostituzione più generica tra tutti gli unificatori dei due termini.

È possibile generalizzare il concetto di unificazione per insiemi di termini, letterali e insiemi di letterali. Dato un insieme di termini T, si dice che T è unificabile se esiste una sostituzione σ tale che $\tau_1^\sigma = \tau_2^\sigma$ per ogni coppia di termini $\tau_1, \tau_2 \in T$. In questo caso σ è detto unificatore di T. Due letterali della stessa arità $L_1 = p_1(\tau_1, ..., \tau_n)$ e $L_2 = p_2(\tau'_1, ..., \tau'_n)$ sono unificabili se e solo se esiste una sostituzione che li eguaglia ignorando il simbolo di predicato. In altre parole sia f una funzione di arità n allora L_1 e L_2 sono unificabili se e solo se sono unificabili i termini $f(\tau_1, ..., \tau_n)$ e $f(\tau'_1, ..., \tau'_n)$. Letterali di diversa arità non sono mai unificabili. Un insieme di letterali è unificabile se e solo se esiste una sostituzione che unifica a due a due tutti i letterali dell'insieme.

Un importante risultato è questo:

Proposizione 1. *Se due termini non sono unificabili allora vale una delle seguenti affermazioni:*

1. *I due termini hanno arità diverse*
2. *I due termini presentano una function obstruction*
3. *I due termini presentano una variable obstruction*

Una function obstruction è una situazione in cui visitando allo stesso modo l'albero sintattico dei termini si incontrano due simboli di funzione diversi. Ad esempio, i termini $f_1(x_1, f_2(x_2))$ e $f_1(x_1, f_3(x_2))$ non sono unificabili in quanto presentano una function obstruction, $f_2 \neq f_3$. Una variable obstruction invece è una situazione in cui visitando allo stesso modo l'albero sintattico dei termini si incontra una variabile x nel primo termine e si incontra un sottotermine t $(\neq x)$ che contiene x nel secondo termine. Ad esempio i termini $f_1(x_1, f_2(x_2))$ e $f_1(f_2(x_1), x_1)$ non sono unificabili in quanto presentano una variable obstruction, x_1 è contenuta in $f_2(x_1)$.

Un noto algoritmo per la ricerca di un MGU di due termini è l'algoritmo di unificazione di *Robinson*. Il collo di bottiglia di questo algoritmo è la occurrence-check, ovvero la ricerca di una variable obstruction. Se si assume che i termini in input non contengono variabili in comune è possibile ignorare questa situazione. Di seguito viene riportato l'algoritmo di Robinson senza occurrence-check:

Algorithm 1: Algoritmo di unificazione di Robinson senza occurrence-check

Firma: unify(τ_1, τ_2)
Input: τ_1, τ_2 due termini
Output: σ un MGU di τ_1 e τ_2 o $\perp$ se non esiste

```
S := Empty Stack of pair of terms;
σ := Empty Substitution;

S.push(τ1, τ2);
while S is not Empty do
    (τ1, τ2) := S.pop();
    while τ1 is a variable ∧ τ1 ≠ τ1^σ do
        τ1 := τ1^σ;
    end
    while τ2 is a variable ∧ τ2 ≠ τ2^σ do
        τ2 := τ2^σ;
    end

    if τ1 ≠ τ2 then
        switch τ1τ2 do
            case τ1 is a variable x and τ2 is a variabile y ⇒ σ := σ ∪ {x/y};
            case τ1 is a variable x ⇒ σ := σ ∪ {x/τ2};
            case τ2 is a variable y ⇒ σ := σ ∪ {y/τ1};
            case τ1 = f(s1, ..., sn) and τ2 = f(t1, ..., tn) ⇒
             S.push(s1, t1), ..., S.push(sn, tn);
            case τ1 = f(s1, ..., sn) and τ2 = g(t1, ..., tm) ⇒ return ⊥;
        end
    end
end
return σ;
```

1.2.3 Semantica

Si indica con *tipo* di φ, l'insieme di tutti i simboli non logici presenti nella formula (costanti, funzioni e predicati). Si definisce *Modello* di un tipo Γ una coppia $\mathcal{M} = (D, I)$ dove D è un insieme non vuoto detto *Dominio* e I è una funzione: $I : \Gamma \to D$ detta d'*Interpretazione* che associa ogni simbolo non logico del tipo a un elemento del dominio in modo tale che per ogni simbolo non logico s:

- Se s è un simbolo di funzione n-aria allora $I(s)$ è una funzione n-aria su D: $I(s) : D^n \to D$.

- Se s è un simbolo di predicato n-ario allora $I(s)$ è una relazione n-aria su D: $I(s) \subseteq D^n$.

Per l'interpretazioni delle costanti si assume che nel dominio siano presenti almeno due oggetti distinti $\{0, 1\} \in D$:

- Se s è un simbolo di funzione costante allora $I(s) \in D$ è un oggetto del dominio.
- Se s è un simbolo di predicato costante allora $I(s) \in \{0, 1\}$.
- Se s è $\top$ allora $I(s) = 1$
- Se s è $\bot$ allora $I(s) = 0$

Per semplicità di lettura, d'ora in poi l'applicazione della funzione I ad un simbolo s verrà indicata con s^I. Si definisce *Contesto*, una qualunque mappatura $\gamma : \Sigma_x \to D$ che associa variabili a un elemento del dominio. Con la scrittura $\gamma[x/t]$ si indica il contesto ottenuto sostituendo il valore della variabile x con l'elemento t. Ad esempio se $\gamma = \{x_1 \to a, x_2 \to b\}$ allora $\gamma[x_1/b] = \{x_1 \to b, x_2 \to b\}$. Per l'applicazione di un contesto a una variabile si utilizza la stessa notazione usata per le funzioni, ovvero $\gamma(x)$. Per l'esempio precedente $\gamma(x_1) = a$, $\gamma(x_2) = b$ e $\gamma[x_1/b](x_1) = b$. Se una variabile x non è presente nel contesto allora si assume che $\gamma(x) = x$. Si definisce l'interpretazione di un termine o predicato τ nel contesto γ secondo l'interpretazione I:

- Se τ è una variabile x allora $\tau^I_\gamma = \gamma(x)$
- Se τ è una funzione $f(\tau_1, ..., \tau_n)$ allora $\tau^I_\gamma = f^I({\tau_1}^I_\gamma, ..., {\tau_n}^I_\gamma)$
- se τ è un predicato $p(\tau_1, ..., \tau_n)$ allora $\tau^I_\gamma = p^I({\tau_1}^I_\gamma, ..., {\tau_n}^I_\gamma)$

Data una formula del primo ordine φ si dice che il modello $\mathcal{M}$ è appropriato per la formula φ se e solo se il tipo della formula è contenuto nel tipo del modello. Un modello appropriato $\mathcal{M}$ soddisfa una formula φ nel contesto γ se e solo se:

- Se φ è un predicato costante p (o $\top/\bot$) allora $\mathcal{M}, \gamma \models p$ se e solo se $p^I = 1$
- Se φ è della forma $\neg\psi$ (dove ψ è una formula) allora $\mathcal{M}, \gamma \models \varphi$ se e solo se $\mathcal{M}, \gamma \not\models \psi$
- Se φ è della forma $(\psi \wedge \chi)$ (con ψ e χ formule) allora $\mathcal{M}, \gamma \models \varphi$ se e solo se $\mathcal{M}, \gamma \models \psi$ e $\mathcal{M}, \gamma \models \chi$
- Se φ è della forma $(\psi \vee \chi)$ (con ψ e χ formule) allora $\mathcal{M}, \gamma \models \varphi$ se e solo se $\mathcal{M}, \gamma \models \psi$ o $\mathcal{M}, \gamma \models \chi$
- Se φ è della forma $\forall x(\psi)$ (dove ψ è una formula) allora $\mathcal{M}, \gamma \models \varphi$ se e solo se per ogni elemento $m \in D$ vale $\mathcal{M}, \gamma[x/m] \models \psi$

- Se φ è della forma $\exists x(\psi)$ (dove ψ è una formula) allora $\mathcal{M}, \gamma \models \varphi$ se e solo se esiste un elemento $m \in D$ tale che $\mathcal{M}, \gamma[x/m] \models \psi$
- Infine se φ è un letterale $p(\tau_1, ..., \tau_n)$ allora $\mathcal{M}, \gamma \models \varphi$ se e solo se $p^I({\tau_1}^I_\gamma, ..., {\tau_n}^I_\gamma)$

Data una formula φ si dice che un modello $\mathcal{M}$ soddisfa φ o anche che $\mathcal{M}$ è un modello di φ se e solo se $\mathcal{M}$ è appropriato per φ e $\mathcal{M}, \gamma \models \varphi$ per ogni contesto γ, in notazione $\mathcal{M} \models \varphi$. Una formula è detta *soddisfacibile* se esiste un modello che la soddisfa. Una formula è detta *valida* se ogni modello la soddisfa. La relazione di conseguenza logica, equivalenza, equisoddisfacibilità e soundness sono definite in modo analogo alla logica proposizionale cambiando la parola 'assegnamento' con 'modello'.

Così come è stata posta questa semantica la soddisfacibilità per formule con variabili libere non è ben definita. Per ovviare a questa cosa si potrebbe estende la definizione di soddisfacibilità per formule con variabili libere. Dato un modello $\mathcal{M}$ appropriato ad una formula φ si dice che $\mathcal{M}$ soddisfa φ se e solo se

- Se la formula non contiene variabili libere allora la definizione di soddisfacibilità rimane invariata.
- Se φ contiene variabili libere allora $\mathcal{M} \models \varphi$ se e solo se per ogni contesto γ che contiene almeno ogni variabile libera di φ vale $\mathcal{M}, \gamma \models \varphi$.

In questo modo fomule con variabili libere divengono uguali in termini di semantica alle stesse formule con le variabili libere quantificate universalmente. Da questo momento in poi ogni variabile libera presente in una formula verrà considerata come vincolata da un quantificatore universale posto all'inizio della formula.

Un'altra osservazione interessante è che se nella formula sono presenti esclusivamente predicati costanti e simboli logici allora il modello si comporta esattamente come un'assegnazione proposizionale. In questo caso è sufficiente rimuovere eventuali quantificatori e cambiare i simboli di predicato da p a s mantenendo l'indice e si ottiene una formula proposizionale che, ha una assegnazione se e solo se la formula originale ha un modello. È possibile estendere questa considerazione anche alle formule ground, ovvero formule senza variabili. Infatti se ogni predicato ground viene sostituito da un nuovo simbolo di costante proposizionale allora la formula proposizionale risultante si comporterà, in termini di soddisfacibilità, esattamente come la formula originale. Ad esempio la formula ground $(p_1(f_2) \vee p_2(f_3, f_1(f_2))) \wedge \neg p_1(f_2) \wedge p_3$ è esattamente equivalente alla formula proposizionale $(s_2 \vee s_3) \wedge \neg s_2 \wedge s_1$, nel senso che per ogni assegnamento per la seconda formula esiste un modello per la prima e viceversa.

1.2.4 Skolemizzazione e Forme Normali

In questo paragrafo verrà descritta una procedura fondamentale per la dimostrazione automatica di teoremi, la *Skolemizzazione*. Varrà inoltre introdotta una nuova forma normale chiamata *PNF* e verranno estese le forme normali descritte nel paragrafo della logica proposizionale per adattarle alla logica del primo ordine.

La definizione per le forme ENNF e NNF per la logica del primo ordine è pressoché identica a quella della logica proposizionale. Come per la logica proposizionale, la trasformazione di una formula in ENNF/NNF preserva la relazione di conseguenza logica ed è sempre possibile trasformare una formula in una equivalente in ENNF/NNF. Il calcolo per la normalizzazione viene effettuato allo stesso modo ma con l'aggiunta di due regole per la negazione dei quantificatori:

$$\neg\forall x(\varphi) \vdash \exists x(\neg\varphi)$$
$$\neg\exists x(\varphi) \vdash \forall x(\neg\varphi)$$

Chiameremo *prefisso di quantificatori* una lista di quantificatori (es. $\forall x_1 \forall x_2 \exists x_3$). Un prefisso viene detto *universale* se è composto esclusivamente da quantificatori universali e viene detto *esistenziali* se è composto esclusivamente da quantificatori esistenziali. Una formula è in formato PNF (Prenex Normal Form) se tutti e soli i quantificatori si trovano all'inizio della formula. La classe di formule PNF è generata dalla seguente grammatica:

$$P_0 := \rho(P)$$
$$P := \top \mid \bot \mid A \mid \neg P \mid (P \wedge P) \mid (P \vee P)$$

Dove ρ è un prefisso di quantificatori e A è un atomo. La parte della formula generata dalla seconda regola viene spesso chiamata *matrice*. È sempre possibile normalizzare una formula in PNF, è un processo sound, ma non è sempre possibile mantenere la relazione di conseguenza logica.

La skolemizzazione è una procedura che permette di eliminare i quantificatori esistenziali da una formula. Sia ρ un prefisso di quantificatori qualunque, la funzione $sk : F' \to F'$ può essere descritta in questo modo:

- Se la formula è del tipo $\exists x(\rho\phi)$ allora sk rimuove il primo quantificatore esistenziale e sostituisce la variabile x all'interno della formula con una nuova costante f_{n+1}, dove n è il massimo indice di costante presente nella formula.
- Se la formula è del tipo $\forall x_k ... x_{k+m-1} \exists x_{k+m}(\rho\phi)$ allora sk rimuove il primo quantificatore esistenziale in ordine lessicografico e si sostituisce la variabile

x_{k+m} all'interno della formula con una nuova funzione $f_{n+1}(x_k, ..., x_{k+m-1})$ $m-1$-aria, dove n è il massimo indice di funzione presente nella formula.

Applicando la funzione sk tante volte quanto il numero di quantificatori esistenziali presenti nella formula si ottiene una formula senza quantificatori esistenziali. Anche la skolemizzazione è un processo sound, ma non è detto che preservi la conseguenza logica.

Combinando le tecniche apprese finora è possibile definire una procedura di normalizzazione che permette di trasformare una formula del primo ordine in formato CNF. Le formule CNF per il primo ordine sono definite come segue:

$$\zeta_0 := \rho(\zeta_1)$$
$$\zeta_1 := \xi \mid (\xi \wedge \zeta_1)$$
$$\xi := \top \mid \bot \mid L \mid (\xi \vee \xi)$$

Dove ρ è un prefisso di quantificatori universale e L un letterale. Per ottenere una formula CNF è sufficiente:

1. Normalizzare la formula in NNF
2. Normalizzare la formula in PNF
3. Skolemizzare la formula
4. Applicare lo stesso algoritmo di clausificazione descritto per la logica proposizionale sulla matrice della formula

Visto le tecniche applicate il processo di clausificazione per la logica del primo ordine è sound ma non preserva la conseguenza logica. Dato che in una formula CNF tutte le variabili sono universalmente quantificate per brevità è possibile omettere il prefisso di quantificatori. Ad esempio, la formula CNF $\forall x_1 x_2 x_3 x_4 (\neg p_1(x_1) \vee p_2(x_2) \vee p_3(x_3)) \wedge (\neg p_4(x_4))$ può essere scritta come $(\neg p_1(x_1) \vee p_2(x_2) \vee p_3(x_3)) \wedge (\neg p_4(x_4))$ e quindi rappresentata in forma insiemistica come $\{\{\neg p_1(x_1), p_2(x_2), p_3(x_3)\}, \{\neg p_4(x_4)\}\}$. Un'osservazione interessante è che visto che tutte le variabili sono universalmente quantificate, è possibile rinominare le variabili di clausole diverse senza cambiare il significato della formula. È quindi possibile normalizzare le clausole in modo tale che ogni coppia di clausole contenga variabili diverse. Un caso d'uso tipico è quello di voler unificare due letterali di due clausole diverse. Con l'assunzione che le variabili siano tutte diverse è possibile applicare l'algoritmo di unificazione senza occurrence-check visto nel capitolo sull'Unificazione.

1.3 Soddisfacibilità e Validità

Nei precedenti capitoli si è data la definizione di soddisfacibilità e validità per formule proposizionali e del primo ordine. In questa sezione verrà approfondito il discorso e verranno introdotte alcune nozioni fondamentali per capire il funzionamento di un *ATP system* (Automatic Theorem Prover), nonché per capire le motivazioni che hanno spinto la creazione di sistemi ATP e i vincoli teorici con cui si devono confrontare. Questa non vuole essere una trattazione esaustiva dei teoremi di incompletezza di Gödel o della teoria della computazione, ma una panoramica generale discorsiva e informale. Il contesto storico è stato estrapolato dai famosi best-seller di *Douglas Hofstadter 'Gödel, Escher, Bach: un'Eterna Ghirlanda Brillante'* [5] e *Martin Davis 'Il calcolatore universale'* [3] mentre i dettagli tecnici sono stati presi dal libro di *Dirk van Dalen 'Logic and Structure'* [12].

Il problema della soddisfacibilità/validità è il problema di determinare se una formula è soddisfacibile/valida o meno (Una tautologia nel caso della logica proposizionale). Sorge spontanea la domanda: *È possibile creare una procedura di decisione che risolva questo problema?*

In primo luogo è bene specificare che il problema della validità è riducibile al problema della soddisfacibilità. Si immagini di voler dimostrare che una formula $\varphi = (A_1 \wedge ... \wedge A_n) \Rightarrow C$ è valida. Nel caso fosse valida allora la sua negazione $A_1 \wedge ... \wedge A_n \wedge \neg C$ sarebbe insoddisfacibile. Nel caso $\neg\varphi$ fosse soddisfacibile allora esisterebbe un modello $\mathcal{M}$ tale che $\mathcal{M} \models \neg\varphi$ che quindi per definizione $\mathcal{M} \not\models \varphi$. In tal caso φ non è valida. In altre parole se un'implicazione è vera allora non è possibile che le premesse siano vere e la conclusione falsa. Tecniche dimostrative del genere sono dette di *Refutazione* o prove per *Assurdo*. Esistono anche altre tecniche dimostrative ma la quasi totalità degli ATP si basano sulla refutazione.

Nel caso della logica proposizionale la risposta alla domanda precedente è affermativa, anche se il problema equivalente *Circuit-SAT* è stato dimostrato essere *NP-completo* dal noto teorema *Cook-Levin*. Data una formula proposizionale di n costanti esistono al massimo 2^n possibili assegnazioni quindi un approccio naive per risolvere il problema della soddisfacibilità potrebbe essere quello di creare un algoritmo brute-force che prova tutte le possibili assegnazioni di verità. Altri metodi verranno discussi nel capitolo 1.4. Nel contesto della logica del primo ordine, la risposta è intricata e la sua soluzione è considerata uno dei risultati più significativi del secolo scorso, rilevante nel campo della matematica, della logica e della filosofia.

Agli inizi del 900' il matematico David Hilbert pubblicò un articolo in cui elencava 23 problemi matematici, all'epoca aperti, che avrebbero dovuto guidare la ricerca matematica del secolo successivo. Il secondo problema di Hilbert era il seguente:

> *È possibile dimostrare la coerenza dell'insieme degli assiomi dell'aritmetica?*

La domanda di Hilbert si riferisce al sistema di assiomi dell'aritmetica basati sulla logica del primo ordine proposti nei tre volumi di *Principia Mathematica* (*PM*) di Bertrand Russell e Alfred North Whitehead. L'indagine del problema portò il suo risolutore, il logico Kurt Gödel, alla scoperta di due importanti risultati.

Il primo risultato, detto *Primo teorema di incompletezza* afferma che:

> *In ogni sistema formale consistente che contiene un'aritmetica elementare, esiste una formula che non è dimostrabile in quel sistema.*

Il secondo risultato, detto *Secondo teorema di incompletezza* afferma che:

> *Se un sistema formale consistente contiene un'aritmetica elementare, allora non può dimostrare la propria coerenza.*

Per capire il senso dei due teoremi è necessario introdurre, i concetti di *Teoria*, *Teorema* e *Sistema formale*. Un sistema formale è un insieme di questi quattro elementi:

- Un alfabeto di simboli.
- Delle regole per la generazione di stringhe dette formule.
- Un insieme di formule dette assiomi.
- Un insieme di regole sintattiche dette d'inferenza che associano formule ad altre formule.

Se si considerano ad esempio le regole sintattiche definite in 1.1.1, la regola d'inferenza del modus ponens della sezione 1.1.2 e come insieme di assiomi le formule proposizionali $\{s_1, s_1 \Rightarrow s_2\}$ si ottiene a tutti gli effetti un sistema formale basato sulla logica proposizionale. Un *Teorema* è un assioma o una qualunque formula che può essere ottenuta applicando un numero finito di volte le regole d'inferenza agli assiomi. Nell'esempio precedente s_2 è un teorema del sistema formale $(s_1, (s_1 \Rightarrow s_2) \vdash s_2)$. Una derivazione sintattica di questo tipo è detta *Dimostrazione*. Per coerenza si intende la proprietà che il sistema non possa dimostrare una contraddizione, come una formula del tipo $\psi \wedge \neg\psi$. Una *Teoria* è un insieme T di formule che rispetta le seguenti proprietà:

1. $\mathcal{A} \subseteq T$ contiene gli assiomi.
2. Per ogni $P \in T^n$ se $P \vdash C$ per qualche regola d'inferenza del sistema formale allora $C \in T$ (Chiusura rispetto la derivazione).

Rispetto l'esempio precedente l'insieme $\{s_1, s_1 \Rightarrow s_2\, s_2\}$ è una teoria. In notazione si scrive $\vdash_T \varphi$ per indicare che la formula φ è dimostrabile nella teoria T.

Tornando alla tesi di Gödel, essa si basa sulla costruzione di una codifica delle formule e delle dimostrazioni di PM nello stesso linguaggio formale PM. Ogni formula (e regola di inferenza) φ viene mappata in un numero naturale $\ulcorner\varphi\urcorner$. In particolare grazie a questa codifica riesce a formalizzare i seguenti predicati all'interno di PM stesso:

- $Proof(x, y)$ che è vero se e solo se x è la codifica di una dimostrazione di una formula codificata in y.
- $Thm(x)$ che è vero se e solo se esiste $y \in \mathbb{N} : Proof(y, x)$
- $Cons$ che è vero se e solo se il sistema è consistente. Si può anche dire che $Cons$ è vero sse $\neg Thm(\ulcorner 0 = 1\urcorner)$

Un lemma fondamentale per la dimostrazione dei teoremi di Gödel è il *Teorema del punto fisso*:

> Per ogni formula φ con un unica variabile libera x allora esiste una formula γ tale che: $\vdash_{PM} \gamma \Leftrightarrow \varphi[x/\ulcorner\gamma\urcorner]$

Se si applica il teorema del punto fisso alla formula $\neg Thm(x)$ si ottiene:

> Esiste una formula γ tale che $\vdash_{PM} \gamma \Leftrightarrow \neg Thm(\ulcorner\gamma\urcorner)$

γ è chiamato *enunciato gödeliano* e in pratica afferma "sono vero se e solo se non sono dimostrabile". Spesso il predicato gödeliano viene associato al *paradosso del mentitore* che è oggetto di studi e dibattiti nell'ambito della filosofia da oltre 2000 anni. Il primo teorema di incompletezza si può quindi riformulare in questo modo:

1. Se PM (o un sistema simile) è coerente allora $\nvdash_{PM} \gamma$ e $\nvdash_{PM} \neg\gamma$

Mentre secondo si può riformulare in questo modo:

2. Se PM (o un sistema simile) è coerente allora $\nvdash_{PM} Cons$

Si potrebbe fare un discorso molto lungo su cosa significhi *'un sistema simile a PM'* ma ciò richiederebbe una dettagliata esplorazione delle dimostrazioni dei teoremi di Gödel e un approfondimento nella Teoria della *Computabilità*. Tuttavia, eviteremo di approfondire ulteriormente su questi argomenti.

In sintesi i teoremi di Gödel evidenziano i limiti di metodi sintattici per la dimostrazione di teoremi all'interno di un sistema formale. Erroneamente si potrebbe pensare che i teoremi provino l'esistenza di formule indimostrabili con ogni metodo, ma in realtà non mostrano nessuna limitazione sul fatto che una formula sia dimostrabile ad esempio in un altro sistema formale o con metodi semantici.

A chiudere il cerchio tra sistemi formali, aritmetica e logica arriva in aiuto il teorema di Church sull'indecidibilità della logica del primo ordine:

Teorema di Church 1. *La logica del primo ordine è indecidibile.*

Il teorema di Church afferma che non esiste un algoritmo che, dato un qualunque enunciato φ di una qualunque teoria T, determini se φ è dimostrabile in T. La prova si basa sull'applicazione del primo teorema di incompletezza di Gödel al sistema formale Q creato dal logico Raphael Robinson. Q è un sistema formale che contiene un'aritmetica elementare descritta da un numero finito di assiomi (al contrario di PM che ne contiene infiniti), Si assuma che $\{q_1, ..., q_n\}$ siano gli assiomi di Q e φ una qualunque formula. Se esistesse una procedura di decisione per la logica allora varrebbe:

$$q_1, ..., q_n \vdash \varphi \text{ se e solo se } \vdash (q_1 \wedge ... \wedge q_n) \Rightarrow \varphi$$

Quindi esisterebbe una procedura di decisione anche per Q, ma in Q vale il primo teorema di Gödel e quindi si giunge a una contraddizione visto che si potrebbe provare il predicato godeliano, $\vdash_Q \gamma$ o $\vdash_Q \neg\gamma$. Ciò discende anche dal fatto che il predicato Thm è *parzialmente calcolabile* ma non *calcolabile*, di conseguenza l'insieme che ha come funzione caratteristica Thm è *ricorsivamente enumerabile* ma non *ricorsivo*. Quindi si può concludere dicendo che la logica del primo ordine è semidecidibile ma non decidibile.

Il teorema di Church ha delle importanti conseguenze sui sistemi di dimostrazione automatica. Un ATP infatti può essere visto come un algoritmo di manipolazione sintattica che cerca di dimostrare la validità di una formula tramite un sistema di inferenze, ma il teorema di Church afferma che, per quanto sofisticato, esisteranno sempre degli input per cui l'algoritmo non terminerà. Questa limitazione rappresenta una barriera fondamentale nell'ambito della dimostrazione automatica.

1.4 Resolution e Dimostrazione Automatica

Resolution o *Risoluzione* è una regola d'inferenza per logica proposizionale e del primo ordine. La regola per la logica proposizionale è la seguente:

$$\frac{\{L\} \cup C_1, \{\neg L\} \cup C_1}{C_1 \cup C_2}$$

Dove C_1 e C_2 sono clausole in notazione insiemistica e L è un letterale. La regola afferma che se si hanno due clausole dove una contiene un letterale e l'altra la sua negazione allora è possibile ottenere una nuova clausola che è l'unione delle due clausole senza il letterale e la sua negazione. Resolution preserva la conseguenza. Con il nome Resolution spesso ci si riferisce anche ad una classe di algoritmi che sfrutta questa regola come inferenza principale per risolvere il problema della soddisfacibilità. Un famoso esempio di algoritmo per la logica proposizionale basato su Resolution è l'algoritmo di Davis-Putnam anche chiamato *DPP* (Davis-Putnam-Procedure). La DPP in sintesi procede in questo modo:

1. Si trasforma la formula una CNF equivalente
2. Si eliminano le clausole tautologiche (Quelle che contengono sia un letterale che la sua negazione $l \vee \neg l$).
3. Si sceglie un qualunque letterale L da una qualunque clausola nella formula ϕ ottenuta nel punto precedente.
4. Si calcolano gli insiemi $S = \{C \in \phi \mid l \notin C \text{ e } \neg l \notin C\}$ e $T = \phi \setminus S$.
5. Si calcola l'insieme $R = \{(C_1 \setminus \{L\}) \cup (C_2 \setminus \{\neg L\}) \mid C_1, C_2 \in T \text{ e } L \in C_1 \text{ e } \neg L \in C_2\}$, detto dei risolventi, applicando la regola di Resolution.
6. Si riapplica la procedura dal punto 2. con la nuova formula $\phi' = S \cup R$ finchè o $\phi' = \{\}$ o ϕ' contiene una clausola vuota.

Se l'algoritmo termina con una clausola vuota vuol dire che nel passo precedente la formula conteneva due clausole del tipo $L \wedge \neg L$, quindi la formula originale è insoddisfacibile. Se l'algoritmo termina con $\phi' = \{\}$ allora la formula originale è soddisfacibile. L'algoritmo seppur corretto è molto inefficiente e non viene utilizzato nella pratica, ma è stato il primo basato su Resolution per il problema della soddisfacibilità. I SAT solver (programmi che risolvono il problema della soddisfacibilità) si basano su tecniche più raffinate e spesso non sono bassati su Resolution. Un esempio è l'algoritmo *DPLL* (Davis-Putnam-Logemann-Loveland) che si basa sulle tecniche di *unit propagation* e *pure literal elimination*.

Per la logica del primo ordine, come al solito, il discorso è più complesso. I primi tentativi di creare un algoritmo per determinare la soddisfacibilità di una formula del primo ordine si devono ai risultati teorici dei logici Skolem, J. Herbrand e R. Robinson. I risultati di Herbrand permettono di 'ridurre' il problema della soddisfacibilità di formule universalmente quantificate al problema della soddisfacibilità di una formula proposizionale. La strategia si basa sulla creazione di un modello il quale dominio, detto universo di Herbrand, è generato da tutti i termini ground ottenibili galla combinazione dei termini della formula originale. L'universo di Herbrand è un insieme finito (se la formula non contiene funzioni) o infinito numerabile. Il secondo passo è chiamato istanziazione ground e consiste nel sostituire tutte le variabili con un sottoinsieme finito di elementi dell'universo di Herbrand. Si ottiene così una formula ground che, come descritto in 1.2.3, può essere trasformata in una formula proposizionale e risolta da un Sat Solver. Se il Sat solver trova un'assegnazione che soddisfa la formula allora si sceglie un altro sottoinsieme finito diverso dal precedente dell'universo di Herbrand e si ripete il procedimento. Se il Sat solver non trova un'assegnazione allora la formula originale è insoddisfacibile. Se si verificano tutti i sottoinsiemi finiti dell'universo di Herbrand senza trovare formule insoddisfacibili allora la formula originale è soddisfacibile.

È chiaro che la strategia di Herbrand è più un risultato teorico che un metodo pratico. L'unico caso in cui è garantita la terminazione è quando la formula non contiene funzioni e quindi l'universo di Herbrand è finito, così come il numero di tutti i sui sottoinsiemi. I primi algoritmi utilizzabili nella pratica iniziarono a nascere dopo che Robinson introdusse la regola di risoluzione per la logica del primo ordine:

$$\frac{\{L(\tau_1, ..., \tau_n)\} \cup C_1, \{\neg L(\omega_1, ..., \omega_n)\} \cup C_1}{(C_1 \cup C_2)^\sigma}$$

Dove σ è un unificatore dei due letterali $L(\tau_1, ..., \tau_n)$ e $L(\omega_1, ..., \omega_n)$ e C_1, C_2 sono clausole. Anche per la logica del primo ordine la regola di risoluzione è corretta e preserva la conseguenza logica. Da qui vi è un punto di svolta nello sviluppo degli ATP. Il primo ATP ad alte prestazioni basato su Resolution per la logica del primo ordine fu *Otter* sviluppato da William McCune. Otter fa parte di una classe di theorem prover basati sulla *Saturazione*. La saturazione è una tecnica concettualmente molto semplice che consiste nel generare tutte le clausole possibili a partire da un insieme di clausole iniziali. Se la formula è insoddisfacibile allora prima o poi verrà generata una clausola vuota. Se la formula è soddisfacibile allora vi sono due casi possibili. Nel primo caso l'algoritmo genera tutte le clausole generabili (satura il sistema) e l'algoritmo termina. Nel secondo caso il numero di clausole generabili dal sistema iniziale non è un numero finito. In tal caso o l'algoritmo capisce che alcune aree di ricerca non porteranno mai alla generazione di una clausola vuota e quindi termina, oppure, nell'ipotesi peggiore, l'algoritmo non termina mai rimanendo in un loop infinito. Quest'ultima è una conseguenza inevitabile dei teoremi di Church e Gödel. Una descrizione più dettagliata di Otter verrà data nel capitolo 3 sull'implementazione di Vampire.

1.5 Il formato TPTP

In questa sezione verrà descritto il formato TPTP [11] (Thousands of Problems for Theorem Provers) per la rappresentazione di problemi di logica del primo ordine. TPTP è una nota libreria di problemi utilizzata per testare e valutare diversi ATP systems (Automatic Theorem Prover). TPTP fornisce diversi formati per la rappresentazione dei problemi, in questa sezione ci si soffermerà sui formati *CNF* (Clausal Normal Form) e *FOF* (First Order Formula).

La traduzione dei predicati, termini e variabili è la stessa sia per il formato CNF che per il formato FOF. Ogni variabile è rappresentata da una stringa alfanumerica Maiuscola. Simboli di funzione, predicato e costanti sono tutti rappresentati senza distinzione da stringhe alfanumeriche minuscole. Le regole della grammatica della generazione dei termini è esattamente la stessa vista nel paragrafo 1.2.1. Per esempio il predicato $p_1(f_1(x_1), x_2, p_2)$ può essere rappresentato come `p1(f1(X1), X2, p2)` o anche `pred(fun(VAR_A), VAR_B, costante)`.

Per la traduzione dei simboli logici si utilizza la seguente mappatura:

Simbolo	**Traduzione**
$\top$	`$true`
$\bot$	`$false`
$\neg$	`~`
$\wedge$	`&`
$\vee$	`|`
$\Rightarrow$	`=>`
$\Leftrightarrow$	`<=>`
$\oplus$	`<~>`
$\forall$	`!`
$\exists$	`?`

Tabella 1.1: Traduzione dei simboli logici

Anche le regole per la generazione delle formule sono le stesse viste nella sezione 1.2.1, con l'unica differenza che i simboli logici vengono tradotti secondo la tabella 1.1. Le parentesi '(' e ')' possono essere omesse e in tal caso si segue il seguente ordine di valutazione dei simboli: `!`, `?`, `~`, `&`, `|`, `<=>`, `=>`, `<~>`. Dopo un quantificatore (`!/?`) è necessaria la lista delle variabili quantificate racchiuse tra parentesi quadre '[' e ']' seguite da ':' e la formula quantificata. Per esempio la formula $\forall x_1 x_2 \exists x_3 (p_1(x_1) \vee p_2(x_2) \vee p_3(x_3))$ viene rappresentata come `![X1, X2] : (?[X3] : (p1(X1) | p2(X2) | p3(X3)))`. Se non presenti quantificatori nella formula le variabili libere vengono considerate quantificate universalmente.

Il formato FOF prevede una lista di assiomi seguiti da una congettura. Il formato cambia a seconda della domanda che si vuole porre all'ATP. Data una lista di assiomi $A_1, ..., A_n$ e una congettura C:

- Se si da in input la lista di assiomi $A_1, ..., A_n$ l'ATP cerca di determinare la Soddisfacibilità della formula $A_1 \wedge ... \wedge A_n$.
- Se vengono dati sia gli assiomi che la congettura l'ATP cerca di determinare se $A_1 \wedge ... \wedge A_n \Rightarrow C$ è valida.
- Se invece viene data solo la congettura l'ATP cerca di determinare se C è valida.

Il formato per inserire un assioma o la congettura è il seguente:

```
fof(<nome>, <tipo>, <formula>).
```

’Nome’ è una stringa alfanumerica che identifica la formula, ’tipo’ può essere `axiom` per gli assiomi e `conjecture` per la congettura. ’Formula’ è una formula del primo ordine in formato FOF. Ad esempio con il file di input:

```
fof(ax1, axiom, p(X)).
fof(ax2, axiom, p(X) => q(X)).
fof(conj, conjecture, q(X)).
```

L’ATP cercherà di determinare se la formula $(p(X) \wedge (p(X) \Rightarrow q(X))) \Rightarrow q(X)$ è valida. Per le formula CNF invece il formato è il seguente:

```
cnf(<nome>, axiom, <clausola>).
```

Dove ’nome’ è definito come per il formato FOF e ’clausola’ è una clausola del primo ordine. L’unico tipo consentito è `axiom` e non è possibile inserire una congettura. In questo formato ogni clausola deve essere scritta in un’annotazione separata. Ad esempio lo stesso problema dell’esempio precedente può essere posto all’ATP in questo modo:

```
cnf(ax1, axiom, p(X)).
cnf(ax2, axiom, ~p(X) | q(X)).
cnf(conj, axiom, ~q(X)).
```

Capitolo 2

Algoritmo di decisione di Frammenti Binding

Nella sezione 1.3, sono stati esaminati i teoremi di Gödel e Church, mentre nella sezione 1.4 sono state viste alcune delle loro conseguenze. La logica del primo ordine è intrinsecamente indecidibile; tuttavia, è possibile identificare alcune sue componenti che risultano decidibili. Queste componenti sono dette *Frammenti* della logica del primo ordine. Si pensi ad esempio ai risultati di Herbrand citati nella sezione 1.4. Se una formula non contiene funzioni ed è universalmente quantificata allora l'universo di Herbrand è finito e vi sono un numero finito di possibili instanziazioni ground. In questo caso determinare la soddisfacibilità di una formula di questo tipo è riducibie al problema della soddisfacibilità proposizionale che è notoriamente decidibile. In letteratura questo frammento è noto come *Bernays–Schönfinkel Fragment*. Altre esempi di frammenti decidibili sono il *Monadic Fragment*, il *Two-variable Fragment*, *Unary negation fragment* e il *Guarded Fragment*. In questo capitolo verrà descritta una famiglia di frammenti relativamente recente chiamata *Binding Fragments* [7] [2].

2.1 Tassonomia dei Frammenti Binding

Si dice che una formula del primo ordine appartiene alla classe *Boolean Binding* (BB) se generata dalla seguente grammatica:

$$\varphi := \top \mid \bot \mid (\varphi \vee \varphi) \mid (\varphi \wedge \varphi) \mid \mathcal{P}(\psi)$$
$$\psi := \rho \mid (\psi \vee \psi) \mid (\psi \wedge \psi)$$

Dove $\mathcal{P}$ è un prefisso di quantificatori e ρ è una combinazione booleana di letterali che hanno come argomento tutti la stessa lista di termini. Una formula di questo tipo verrà chiamata con il nome τ-Binding, dove τ indica la lista di termini comune. Ad esempio sono $(f_1(x_1), f_2)$-Binding le formule: $p_1(f_1(x_1), f_2)$, $p_1(f_1(x_1), f_2) \vee \neg p_3(f_1(x_1), f_2)$. Per semplicità di scrittura è possibile omettere la lista di termini comune e posizionarla in notazione postfissa:

$$p_1(f_1(x_1), f_2) \vee \neg p_3(f_1(x_1), f_2) \text{ diventa } (p_1 \vee \neg p_3)(f_1(x_1), f_2)$$

Con $\mathcal{B}^\tau$ verrà indicato l'insieme di tutte le formule τ-Binding. Si definisce la funzione $term : \mathcal{B}^\tau \to T^n$ che associa ogni τ-Binding alla sua lista di termini comune τ. Ad esempio $term((p_1 \vee \neg p_3)(f_1(x_1), f_2)) = (f_1(x_1), f_2)$. Verranno chiamati impropriamente τ-Binding anche formule universalmente quantificate la cui matrice è un τ-Binding. In questo caso ci si riferisce esclusivamente alla matrice della formula eliminando i quantificatori.

I frammenti Binding possono essere ottenuti restringendo le regole di ψ:

- Il frammento *One Binding* (1B) viene ottenuto restringendo la seconda formula a $\psi := \rho$
- Il frammento *Conjunctive Binding* (CB o $\wedge$B) viene ottenuto restringendo la seconda formula a $\psi := \rho \mid (\psi \wedge \psi)$
- Il frammento *Disjunctive Binding* (DB o $\vee$B) viene ottenuto restringendo la seconda formula a $\psi := \rho \mid (\psi \vee \psi)$

Un'istanza particolare del frammento 1B è quando la formula non contiene quantificatori esistenziali. Una formula 1B con soli prefissi universali viene detta del frammento *Universal One Binding* ($\forall 1B$).

2.2 Soddisfacibilità dei frammenti Binding

In questa sezione verrà analizzato il problema della soddisfacibilità dei frammenti binding. In particolare verrà descritto l'algoritmo di decisione per i frammenti 1B e CB che è il soggetto principale dello studio di questa tesi.

Data una formula del frammento 1B è facile osservare che il processo di skolemizzazione converte la formula in formato $\forall 1B$. Se si applica la stessa procedura ad una formula CB, le sottoformule generate dalla regola ψ saranno del tipo: $\mathcal{P}(\rho_1 \wedge ... \wedge \rho_n)$ con $\mathcal{P}$ un prefisso universale e $(\rho_1 \wedge ... \wedge \rho_n)$ τ-Binding. In questo caso è possibile distribuire il '$\forall$' sui vari τ-Binding e si ottiene così una formula equisoddisfacibile in formato $\forall$1B.

Teorema: Decidibilità dei frammenti 1B e CB 2.2.1. *I frammenti 1B e CB sono frammenti decidibili del primo ordine.*

Una dimostrazione dettagliata di questo teorema può essere trovata nell'articolo [2]. Si può osservare che il processo di clausificazione del primo ordine porta alla generazione di una formula equisoddisfacibile che rispetta il formato DB. Ne consegue immediatamente per il teorema di Church:

Teorema: Indecidibilità del frammento Disjunctive Binding 2.2.2. *Il frammento DB è un frammento indecidibile del primo ordine.*

Dimostrazione. Per assurdo Esiste un algoritmo di decisione totale S per formule del frammento DB. Data una qualunque formula φ è possibile trasformarla in una equisoddisfacibile in formato CNF. Se si distribuisce il quantificatore universale sulle clausole si ottiene una formula φ' che rispetta i requisiti sintattici del frammento DB. S è quindi una procedura di decisione totale per tutta la logica del primo ordine ma ciò è in contraddizione con il teorema di Church. □

Il processo di skolemizzazione consente di concentrarsi sullo studio del frammento ∀1B per la risoluzione del problema della soddisfacibilità. Prima di descrivere l'algoritmo bisogna introdurre tre nuovi concetti: L'Unificazione per τ-Binding, Implicante di una formula del primo ordine e la conversione booleana di un τ-Binding. Data una formula del primo ordine φ per Implicante di φ si intende la conversione del primo ordine di un implicante della 'struttura proposizionale esterna'. ad esempio la formula $\forall x_1(p_1(x_1) \vee p_2(x_1)) \wedge (p_1(f_1) \vee \exists x_2(p_3(x_2))) \wedge \neg p_1(f_1) \wedge \exists x_2(p_3(x_2))$ ha la seguente struttura booleana $s_1 \wedge (s_2 \vee s_3) \wedge \neg s_2 \wedge s_3$. Un implicante (e anche il solo) di questa formula è l'insieme $\{s_1, s_3\}$ che ri-convertito nel primo ordine diventa l'insieme $\{\forall x_1(p_1(x_1) \vee p_2(x_1)), \exists x_2(p_3(x_2))\}$. In questo caso è stata creata implicitamente un bi-mappa tra costanti proposizionali e formule del primo ordine:

- $s_1 \rightleftarrows \forall x_1(p_1(x_1) \vee p_2(x_1))$
- $s_2 \rightleftarrows p_1(f_1)$
- $s_3 \rightleftarrows \exists x_2(p_3(x_2))$

Un τ_1-Biding e un τ_2-Biding sono detti unificabili se e solo se l'insieme congiunto di tutti i loro letterali è unificabile. Si può anche dire che sono unificabili sse le due liste τ_1 e τ_2 hanno la stessa lunghezza n e dato un qualunque predicato p n-ario $p(\tau_1)$ e $p(\tau_2)$ sono unificabili. Una insieme di τ-Biding è unificabile sse esiste una sostituzione che unifica a due a due tutti gli elementi dell'insieme. Dato un τ-Binding ϕ la sua conversione booleana $bool(\phi)$ è una formula proposizionale che si ottiene da ϕ mantenendo la sua struttura proposizionale, eliminando gli argomenti dai letterali e convertendo i simboli di predicato in simboli di costante con lo stesso indice. Ad esempio il τ-Binding $((p_1 \wedge p_4) \vee p_2 \vee \neg p_4)(\tau)$ viene convertito nella seguente formula proposizionale $(s_1 \wedge s_4) \vee s_2 \vee \neg s_4$

A questo punto è possibile enunciare il teorema di caratterizzazione della soddisfacibilità del frammento ∀1B.

Teorema: Caratterizzazione della soddisfacibilità per il frammento ∀1B 2.2.3. *Data una formula φ del frammento ∀1B, φ è soddisfacibile se e solo se:*
Esiste un implicante I dove: per ogni sottoinsieme $U \subseteq I$ di τ-Binding, se $U = \{\gamma_1, ..., \gamma_n\}$ è unificabile allora la formula proposizionale $bool(\gamma_1) \wedge ... \wedge bool(\gamma_n)$ è soddisfacibile.

Dal teorema appena descritto si estrapola intuitivamente l'algoritmo per la soddisfacibilità delle formule del frammento:

Algorithm 2: Algoritmo per la soddisfacibilità del frammento ∀1B

```
Firma: oneBindingAlgorithm(φ)
Input: φ una formula ∀1B
Output: ⊤ o ⊥
foreach I Implicant of φ do
    res := ⊤;
    foreach (U := {γ1, ..., γn}) ⊆ I do
        if U is unifiable then
            if bool(γ1) ∧ ... ∧ bool(γn) is not satisfiable then
                res := ⊥;
                Break;
            end
        end
    end
    if res = ⊤ then
        return ⊤
    end
end
return ⊥
```

I prossimi capitoli si concentreranno sullo studio dei dettagli tecnici per l'implementazione di questo algoritmo, con annesse osservazioni sulle sfide implementative e una analisi dei risultati sperimentali ottenuti.

Capitolo 3

Il Theorem prover Vampire

Vampire [9] [6] [1] è un dimostratore di teoremi automatico per la logica del primo ordine basato su *Resolution*. Nasce nel 1998 come progetto di ricerca degli autori Andrei Voronkov e Alexandre Riazanov, adesso è correntemente mantenuto e sviluppato da un team più ampio presso il dipartimento di Computer Science dell'Università di Manchester. Il software è open-source, sviluppato in C++ e al momento della scrittura di questa tesi è giunto alla versione 4.8 con licenza BSD-3. Vampire incorpora un complesso sistema strutture dati, algoritmi per la manipolazione di formule e termini e un vasto sistema di inferenze. Uno dei suoi punti di forza è l'efficienza, Il team di sviluppo infatti partecipa annualmente al *CASC* (The CADE ATP System Competition), una competizione tra sistemi ATP, e fino ad ora ha sempre vinto almeno in una categoria ogni anno. Questa ambizione per l'efficienza ha influenzato molto la struttura di Vampire e la sua implementazione. Questo è sia un lato positivo che negativo, infatti se da un lato ci si ritrova con funzioni efficienti e ben ottimizzate, dall'altro lato ci si ritrova spesso con un codice complesso e difficile da comprendere che predilige la velocità alla pulizia. Ogni suo componente è riconducibile ad un articolo che ne spiega il funzionamento ad alto livello ma spesso, alcune scelte implementative sono poco o per nulla documentate. Spesso lo stesso nome di una funzione o di una classe fa intuire il suo scopo e funzionamento ma non sempre è così e altrettanto spesso si è costretti a fare 'Reverse Engineering' del codice sorgente per capire come è stato utilizzato in altri contesti. Questo è un problema di cui il team di sviluppo è consapevole e negli ultimi anni sta cercando di migliorare. In questo capitolo si cercherà di dare una panoramica generale di Vampire, spiegando le sue componenti principali e come queste interagiscono tra di loro, con un focus particolare su quelle che sono state utilizzate per la realizzazione della procedura di decisione per frammenti Binding. Nella figura 3.1 è mostrata la disposizione delle cartelle di Vampire. La struttura è molto piatta ma assolutamente organizzata. Nella cartella *Kernel* sono presenti le componenti principali del sistema come ad

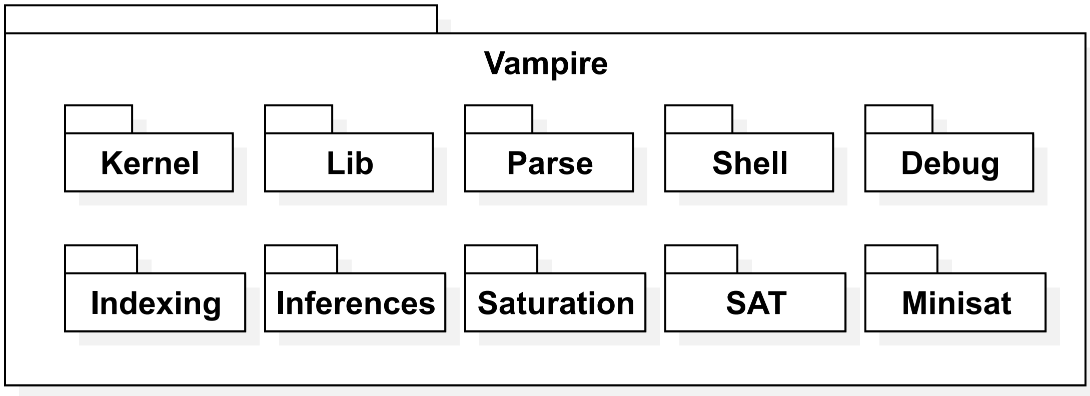


Figura 3.1: Struttura delle cartelle di Vampire.

esempio le strutture per le formule e i termini, e il 'Main Loop' del programma che si occupa di gestire il processso di dimostrazione. La struttura delle formule verrà trattata nella sezione 3.1. Nella cartella *Lib* sono presenti le strutture dati e le funzioni di utilità come Array, Mappe, Liste, Stack, ecc. Nella cartella *Parse* sono presenti le classi che decodificano i file in formato TPTP o SMT. Nella cartella *Shell* sono presenti le classi per la gestione dell'input/output da riga di comando e tutte le funzioni necessarie per il Preprocessing. Gli step del preprocessing verranno approfonditi nella sezione 3.3. Nella cartella *Indexing* sono presenti i componenti per l'indicizzazione dei termini. Le particolari strutture per l'unificazione verranno trattate nella sezione 3.5. Nelle cartelle *Inferences* e *Saturation* sono presenti le classi che contengono le regole di inferenza e gli algoritmi di saturazione. Questi verranno trattati nelle sezioni 3.4 e 3.6. Nelle cartelle *SAT* e *Minisat* sono presenti le interfacce per utilizzare i SAT-Solver e il codice di Minisat, un SAT-Solver open-source. Il funzionamento dei sat solver verrà discusso nella sezione 3.6. Nella cartella *Debug* sono presenti le classi e le macro per la misurazione dei tempi e le statistiche di esecuzione. Alcuni esempi verranno mostrati nella sezione 3.7.

3.1 I Termini

I termini, insieme a clausole e formule, sono la struttura dati più importante in un dimostratore di teoremi ed è quindi fondamentale che siano rappresentati nel modo più efficiente possibile. Nella figura 3.2 è mostrata una rappresentazione ad alto livello e molto semplificata della struttura dei termini implementata in Vampire. Un termine come inteso nella sezione 1.2.1 è rappresentata dalla classe *TermList*. TermList è composto da tre elementi principali: *term*, *content* e *info*. I tre componenti sono definiti all'interno di una **union** per risparmiare memoria.

- *term* è un puntatore ad un oggetto della classe *Term*

- content è un intero di 64 bit
- info è una struttura BitField di esattamente 64 bit

Essendo definiti all'interno di una union, ogni TermList dovrebbe occupare esattamente 64 bit di memoria. In vampire ogni variabile è rappresentata da un numero intero senza segno mentre i termini complessi composti da funzioni sono rappresentati dalla classe Term. Se TermList rappresenta una variabile allora content shiftato di 2 bit verso destra rappresenta l'indice di quella variabile ($content/4$), nel caso rappresenti una funzione allora term punta ad un oggetto di tipo Term che contiene l'effettiva struttura della funzione. Nella classe Term il nome della funzione è rappresentata da un intero senza segno globalmente univoco definito nella classe *Signature*. La classe Signature contiene le informazioni relative all'indice, arità e nome di funzioni e predicati. Term inoltre contiene un Array di TermList di lunghezza pari ad $arity + 1$ che rappresenta gli argomenti della funzione listati da destra verso sinistra. L'elemento in posizione 0 contiene un Termlist fittizio che contiene le info dello stesso termine.

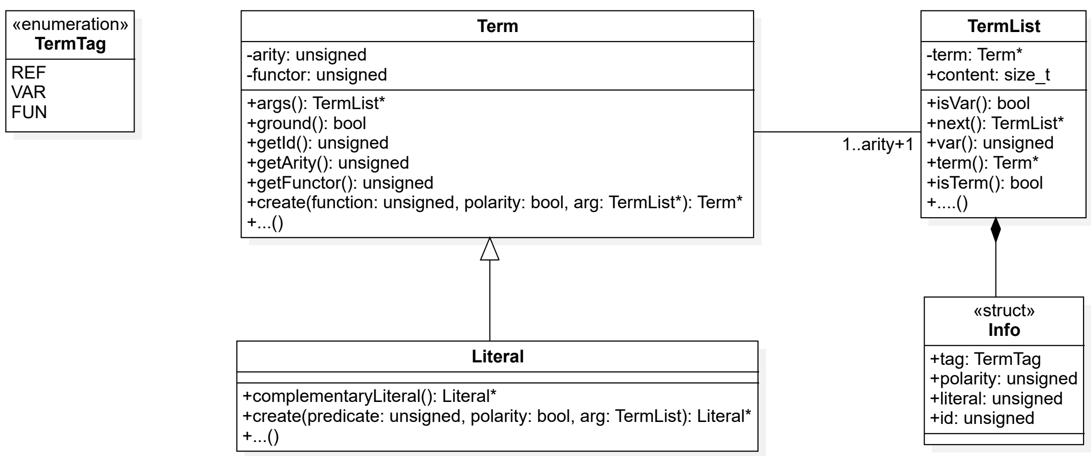


Figura 3.2: Struttura dei termini

Tutti i termini di default sono rappresentati da una struttura Perfectly Shared (come descritto in 1.2.1) per risparmiare memoria e velocizzare le operazioni di confronto. I letterali sono rappresentati dalla classe *Literal* che è una specializzazione della classe *Term*. Nell'implementazione Vampire non fa nessuna distinzione tra nomi di funzione funzioni o predicati essi sono infatti rappresentati entrambi nella Signature come funzioni. Termini e Letterali sono salvati nella Signature in strutture di indicizzazione (SubstitutionTree) per permettere un accesso veloce. Un accenno a queste strutture verrà fatto nella sezione 3.5. Literal contiene inoltre funzioni specifiche per la manipolazione dei letterali, come *complementaryLiteral* che restituisce lo stesso letterale negato (dalla struttura di indicizzazione se presente altrimenti ne crea uno nuovo). Le funzioni

Term::create e Literal::create sono le funzioni utilizzate per creare nuovi termini e letterali e inserirli nella struttura di indicizzazione.

Ad esempio, il predicato $\neg p_1(f_2, f_3(x_5), x_5, f_3(x_5))$ viene rappresentato in memoria con una struttura del genere:

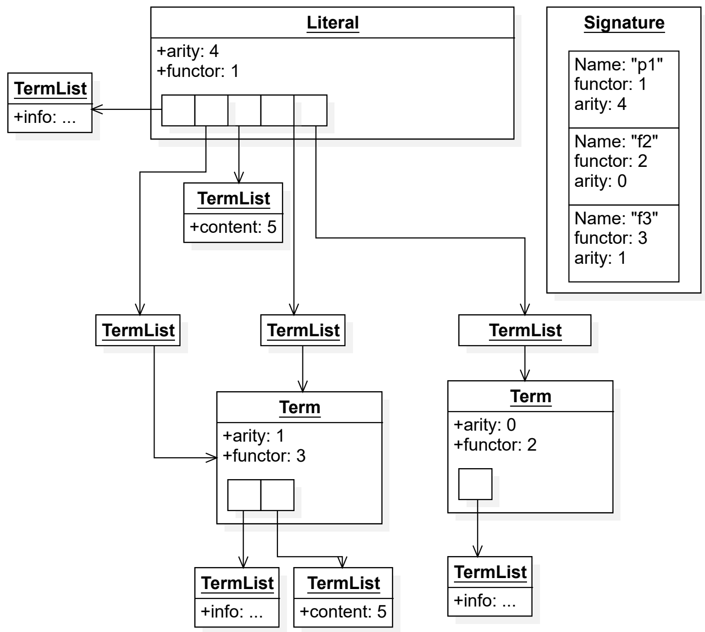


Figura 3.3: Esempio di rappresentazione di un termine

3.2 Unità, Formule e Clausole

Vampire prende in input formule del tipo $A_1 \wedge A_2 \wedge ... \wedge A_n \wedge \neg C$, dove $A_1, A_2, ..., A_n$ sono assiomi e C è la congettura, e cerca di dimostrarne l'insoddisfacibilità. Per fare ciò il problema principale viene scomposto in una lista di elementi chiamati *Unità*. Un unità è una formula o una clausola affiancata da una regola di inferenza che lo ha generata. In sostanza vi sono due tipi di inferenze, quelle che rappresentano unità date in input come *Axiom* per indicare che l'unità è un assioma in input o *Negated Conjecture* per indicare che l'unità è la negazione della congetture e quelle che rappresentano altre formule/clausole generate all'interno del processo dimostrativo. Le inferenze di questo tipo includono anche una reference alle formule che hanno generato la nuova unità, rendendo quindi possibile risalire alla dimostrazione al termine dell'esecuzione.

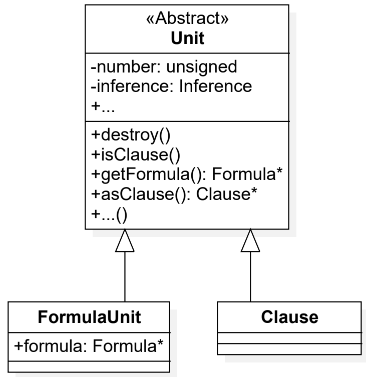


Figura 3.4: Struttura delle unità

Le unità come mostrato in figura 3.4 sono rappresentate dalla classe astratta *Unit*, che si specializza nelle classi *FormulaUnit* che contiene un puntatore ad un oggetto di tipo *Formula* e *Clause* che verrà trattata in seguito. Le formule sono rappresentate da una struttura ad albero esattamente come quelle vista in 1.1.1 e 1.2.1. La classe *Formula* 3.5 è una classe astratta che si specializza nelle classi:

- *AtomicFormula* che rappresenta una formula composta da un solo letterale.
- *BinaryFormula* rappresenta le formule binarie $A \Rightarrow B$, $A \Leftrightarrow B$ e $A \oplus B$.
- *NegatedFormula* rappresenta le formule negate del tipo $\neg A$.
- *QuantifiedFormula* rappresenta le formule quantificate del tipo $\forall/\exists x1, x2, ..., xn : A$.
- *JunctionFormula* rappresenta le formule composte dalla concatenazione di $\wedge$ e $\vee$.

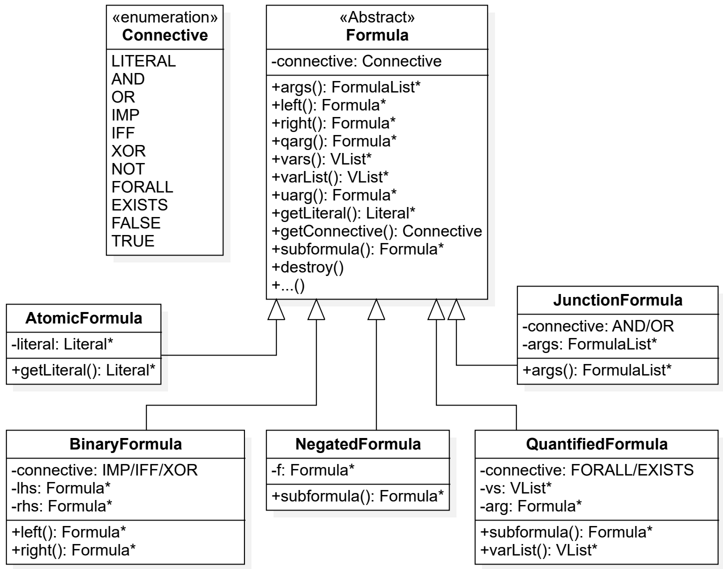


Figura 3.5: Struttura delle formule

Le clausole sono rappresentate dalla classe *Clause* 3.6 che è una specializza della classe *Unit*. Ogni clausola contiene un Array di letterali e sono quindi rappresentate in maniera molto simile alla notazione insiemistica vista in 1.1.3.

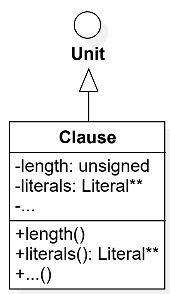


Figura 3.6: Struttura delle Clausole

3.3 Preprocessing

Vampire è in grado di elaborare formule del primo ordine in qualsiasi forma, tuttavia, poiché il suo algoritmo di saturazione opera esclusivamente su clausole, è necessario eseguire delle operazioni di trasformazioni sull'input. L'insieme di queste operazioni è chiamato *Preprocessing*. Il preprocessing non ha solo lo scopo di convertire la formula in input in una equisoddisfacibile in forma clausale ma ha anche l'obbiettivo di rendere il problema più semplice (se possibile). In generale tutte le operazioni non superano la complessità di $O(n \cdot log(n))$ e le operazioni non essenziali sono attivabili/disattivabili tramite opzioni da riga di comando.

Di seguito sono riportate le principali operazioni di preprocessing eseguite da Vampire:

1. ***Rectify***: Verifica se la formula contiene variabili libere e in caso affermativo le quantifica e verifica che la stessa variabile non sia quantificata più volte nello stesso ramo dell'albero sintattico.
2. ***Simplify***: Semplifica e Rimuove le occorrenze di $\perp$ e $\top$ quanto possibile.
3. ***Flatten***: Unisce sequenze di $\wedge/\vee$ in un'unica congiunzione/disgiunzione o sequenze di $\forall/\exists$ in un'unica quantificazione.
4. ***Unused definitions and pure predicate removal*** (opzionale): Elimina i predicati costanti che non possono creare contraddizioni.
5. ***ENNF***: Trasforma la formula in forma ENNF come visto in 1.2.4.
6. ***Naming*** (opzionale): Applica una tecnica di naming simile a come visto in 1.1.4 ma estesa alla logica del primo ordine.
7. ***NNF***: Trasforma la formula in forma NNF come visto in 1.2.4.
8. ***Skolemization***: Elimina i quantificatori esistenziali come visto in 1.2.4 con l'unica differenza che si evita di convertire la formula in PNF.
9. ***Clausification***: Clausifica la formula in modo simile a come accennato in 1.2.4.

3.4 Algoritmo di Saturazione

Vampire fa parte di una famiglia di ATP basati su saturazione che implementa la *Given Clause Architecture* (*GCA*). Data una formula in formato CNF la GCA prevede due insiemi di clausole dette *Attive* e *Passive*. Inizialmente l'insieme delle clausole attive è vuoto e l'insieme delle clausole passive contiene tutte le clausole della formula. Dopo questo setup iniziale comincia quello che viene chiamato *Main Loop*. Il Main Loop è un ciclo che termina quando l'insieme delle clausole passive è vuoto o quando viene trovata una clausola vuota. Ad ogni iterazione il Main Loop seleziona una clausola dall'insieme delle clausole passive. Questa clausola viene chiamata *Given Clause* (*GC*). Lo step successivo consiste nell'applicare tutte le inferenze possibili tra la GC e le clausole attive. Le nuove clausole generate vengono aggiunte all'insieme delle clausole passive mentre la GC viene spostata nell'insieme delle clausole attive. Se una delle nuove clausole generate è la clausola vuota all'ora il Main Loop termina e la formula è insoddisfacibile. Se invece l'insieme delle clausole passive viene totalmente svuotato allora il sistema è Saturo e la formula è soddisfacibile. Nel caso la formula non sia insoddisfacibile l'insieme delle clausole passive potrebbe non svuotarsi mai, in questo caso Il Main Loop termina quando sono terminate le risorse disponibili.

Algorithm 3: Architettura Given Clause

Firma: Saturation(φ)
Input: φ Una formula in formato CNF
Output: $\top$ se φ è soddisfacibile, $\bot$ altrimenti

```
active = ∅
passive = φ
while passive ≠ ∅ do
    current := select(passive);
    passive.remove(current);
    active.add(current);
    newClauses := infer(current, active);
    if □ ∈ newClauses then
        return ⊥;
    end
    passive.add(newClauses);
end
return ⊤;
```

Come già accennato in 1.4 Otter è stato uno dei primi ATP basati su saturazione e su GCA. In particolare Otter aggiunge a GCA due nuovi step chiamati di semplificazione. Il sistema di inferenze viene diviso in due classi, le inferenze generative e di semplificazione. Le inferenze generative prendono una o più clausole e generano nuove clausole. Le inferenze di semplificazione prendono una o più clausole e inferiscono una nuova clausola, generalmente più corta, che rende le premesse ridondanti in modo da poterle sostituire con la clausola generata. Nel primo step di semplificazone chiamato *Forward semplification* dopo aver selezionato una clausola dall'insieme delle clausole passive, si tenta di applicare le inferenze di semplificazione alla clausola selezionata. Il secondo step di semplificazione chiamato *Backward semplification* consiste nell'applicare le inferenza di semplificazione alle clausole attive e passive. Gli algoritmi che implementano questa struttura vengono detti *Otter* o che implementano la *Otter's Architecture*. Un esempio di Otter's Architecture è mostrato in 4.

Algorithm 4: Architettura Otter

Firma: Saturation(φ)
Input: φ Una formula in formato CNF
Output: $\top$ se φ è soddisfacibile, $\bot$ altrimenti

```
active = ∅
passive = φ
while passive ≠ ∅ do
    current := select(passive);
    passive.remove(current);
    if retained(current) then
        current := forwardSimplify(current, active, passive);
        if current = □ then
            return ⊥;
        end
        if retained(current) then
            (active, passive) := backwardSimplify(current, active, passive);
            active.add(current);
            newClauses := infer(current, active);
            if □ ∈ newClauses then
                return ⊥;
            end
            passive.add(newClauses);
        end
    end
end
return ⊤;
```

Con la funzione *retained* si intende una funzione che restituisce *true* se la clausola è utile per la dimostrazione e *false* altrimenti. Ad esempio se la clausola è una tautologia viene scartata. Questa fase è detta *Retention Test*. Vampire implementa tre algoritmi di saturazione, *Otter*, *LRS Discount*. Otter è una versione leggermente modificata della Otter's Architecture. Al posto di avere solo due insiemi di clausole attive e passive, ha un terzo insieme chiamato *unprocessed*. L'algoritmo LRS (Low resource strategy) è una versione modificata di Otter che guida l'algoritmo in base ai limiti di tempo e memoria dati in input e il tempo/memoria restanti. In particolare quando si effettua il retention test LSR valuta se la clausola è troppo grande da processare in base al tempo e alla memoria rimanente. Se la clausola è troppo grande viene scartata anche a costo della perdita di completezza. Nel caso specifico in cui la formula è soddisfacibile e LSR scarta una clausola, l'algoritmo termina con un errore. Nel caso in cui la formula è insoddisfacibile LSR in alcuni casi performa meglio quando si restringono i limiti di tempo e memoria. L'algoritmo Discount è simile ad Otter solo che gli step di semplificazione vengono eseguiti solo sull'insieme delle clausole attive. Questo è dovuto al fatto che l'insieme delle clausole passive è significativamente più ampio rispetto a quello delle clausole attive. Le clausole attive sono spesso solo l'1% di quelle passive, il che comporta tempi prolungati durante gli step di semplificazione poiché è necessario dedicare molto tempo alla semplificazione delle clausole passive.

Vampire implementa un vasto sistema di inferenze ma il sottoinsieme minimo necessario per la completezza (per la soddisfacibilità di formule senza uguaglianza) è composto dalle inferenze di *Resolution* e *Factoring*.

Resolution:

$$\frac{\{L(\tau_1, ..., \tau_n)\} \cup C_1, \{\neg L(\omega_1, ..., \omega_n)\} \cup C_1}{(C_1 \cup C_2)^\sigma}$$

Factoring:

$$\frac{\{L(\tau_1, ..., \tau_n)\} \cup C \cup \{L(\omega_1, ..., \omega_n)\}}{(\{L(\tau_1, ..., \tau_n)\} \cup C)^\sigma}$$

Con L un letterale, C_1, C_2 clausole, $\tau_1, ..., \tau_n, \omega_1, ..., \omega_n$ termini e σ un unificatore.

3.5 Unificazione e Substitution Trees

Come visto nella sezione precedente l'unificazione è un passo fondamentale per l'applicazione delle regole di inferenza di *Resolution* e *Factoring*. La GCA inoltre prevede che dopo aver selezionato la GC si applichino tutte le possibili inferenze tra la GC e le clausole attive. Senza un'apposita struttura di indicizzazione la ricerca delle inferenze applicabili sarebbe spaventosamente lento. Si pensi di voler verificare se è applicabile la regola di resolution tra due clausole C_1 e C_2. Un algoritmo naive potrebbe essere ad esempio quello di scorrere tutti i letterali di C_1 e C_2 e verificare se la polarità di un letterale è opposta a quella di un altro letterale e se sono unificabili. Questo algoritmo avrebbe un costo nel caso peggiore di $\mid C_1 \mid \cdot \mid C_2 \mid$ per il costo di $unifiable$. È chiaro che una strategia del genere non è sostenibile soprattutto se l'obbiettivo è quello di essere il più rapidi possibile. Vampire utilizza una struttura di indicizzazione chiamata *Substitution Tree* (ST) per velocizzare le operazioni di unificazione. Un SubstitutionTree è una struttura ad albero in cui ogni nodo rappresenta una sostituzione. Si aggiunge oltre all'insieme standard di variabili Σ_x un altro insieme di variabili $\Sigma_x^* = \{*_1, *_2, ...\}$, dette speciali, che servono per la costruzione delle sostituzioni. I termini speciali sono termini che possono contenere anche variabili speciali oltre a quelle standard.

Per sostituzione in un ST si intende una mappa da variabili speciali a termini speciali e vengono scritte in notazione: $\{*_i = \tau, *_j = \tau', ...\}$. Per $im(\sigma)$ si indica l'insieme dei termini speciali che compaiono a destra del simbolo dell'uguaglianza in σ. Per $dom(\sigma)$ si indica l'insieme delle variabili speciali che compaiono a sinistra del simbolo dell'uguaglianza in σ. Per composizione di sostituzioni $\sigma_1 \circ \sigma_2$ si intende la sostituzione che si ottiene applicando σ_2 ai termini mappati da σ_1. Ad esempio se $\sigma_1 = \{*_0 = f(*_1, *_2)\}$ e $\sigma_2 = \{*_1 = a, *_2 = g(*_3)\}$ allora $(\sigma_1 \circ \sigma_2) = \{*_0 = f(a, g(*_3))\}$.

Un nodo di un ST è rappresentato da una coppia (σ, T) dove σ è una sostituzione e T è un insieme di ST. Vi sono tre tipi di nodi:

- **Nodo Radice**: Se il nodo è la radice dell'albero allora σ è la sostituzione vuota e T è un insieme non vuoto
- **Nodo Foglia**: Se il nodo è una foglia allora T è l'insieme vuoto. Ad ogni nodo foglia viene associato una struttura chiamata *LeafData* che può contenere dati arbitrari.
- **Nodo Interno**: Se il nodo è interno ne σ ne T sono vuoti.

Ogni ST rispetta i seguenti vincoli:

1. Per ogni percorso $(\sigma_1, T_1), ..., (\sigma_n, T_n)$ dalla radice ad una foglia, l'immagine della composizione delle sostituzioni non contiene variabili speciali (è un termine standard): cioè vi sono solo termini standard in $im(\sigma_1 \circ ... \circ \sigma_n)$.
2. Per ogni percorso $(\sigma_1, T_1), ..., (\sigma_n, T_n)$ dalla radice ad una foglia, ogni variabile speciale è mappata al massimo una volta per percorso: cioè per ogni $i \neq j$, $dom(\sigma_i) \cap dom(\sigma_j) = \emptyset$.

In Vampire è implementata una variante di ST chiamati *Downward Linear Substitution Tree* che facilitano le operazioni di inserimento e ricerca. Una trattazione più approfondita può essere trovata in [4]. Un ST accetta termini e letterali. Il processo di inserimento costruisce un percorso di sostituzioni all'interno dell'albero in modo tale che applicando tutte le sostituzioni del percorso alla variabile $*_0$ si ottiene il termine inserito inizialmente. Prima dell'inserimento le variabili dei termini vengono normalizzate in modo da avere gli indici più bassi possibili. I termini originali vengono poi salvati nelle LeafData per essere recuperati. Per la normalizzazione delle variabili vengono utilizzati i *Variable Banks*, che sono essenzialmente un secondo indice delle variabili, che servono a tenere sempre disgiunte le variabili dei termini "query" (i termini passati come argomento in una funzione di ricerca) da quelle dei termini nel ST. Così facendo è possibile utilizzare, quando necessario, algoritmi di unificazione senza occurrence check tra letterali query e letterali nel ST. Ad esempio se si vuole inserire il termine $f(x, y)$ viene salvato nel ST come $f(x_{1/1}, x_{2/1})$ dove /1 indica il Variable Bank.

Vampire mette a disposizione numerose classi per gestire varie tipologie di indicizzazione tramite ST. Ad esempio la classe *LiteralSubstitutionTree* viene utilizzata per salvare i letterali in un ST. Nei LeafData vengono salvati il letterale e la clausola a cui appartiene. La funzione $getUnifications(Literal * query, bool\ complementary, bool\ retrieveSubstitutions)$ restituisce un iteratore di letterali che unificano con il letterale query. Se complementary è true allora cerca solo i letterali con polarità opposta. Se retrieveSubstitutions è true allora restituisce anche la sostituzione. Tornando all'esempio dell'inizio della sezione, per trovare le clausole su cui si può applicare la regola di resolution con la GC un algoritmo più semplice e molto più efficiente rispetto a quello proposto inizialmente consiste nell'inserire tutte le clausole attive nel ST e chiamare la funzione $getUnifications$ su ogni letterale della GC:

$newClauses := \emptyset$
foreach $l \in GC$ **do**
  $iter := getUnifications(l, true, true)$
  while $iter.hasNext()$ **do**
    $res := iter.next()$
    $(l' : Literal*, C : Clause*) := res.leafData()$
    $\sigma := res.substitution()$
    $newClauses.add(resolution(l, l', GC, C, \sigma))$
  end
end

LiteralSubstitutionTree crea una mappa da simboli di predicato a ST in modo tale che ogni letterale con lo stesso predicato venga inserito nello stesso ST. Per inserire letterali con simboli di predicato diversi si può utilizzare la classe *SubstitutionTree* a patto che i predicarti abbiano stessa arità. Con la classe *SubstitutionTree::UnificationsIterator* e la funzione *SubstitutionTree::iterator* è possibile ottenere gli stessi risultati dell'esempio precedente:

3.6 Il SAT-Solver

Vampire non implementa un SAT-Solver ma ha un vasto sistema di interfacce per utilizzare al meglio SAT-Solver esterni. Al momento gli unici SAT-Solver supportati sono MiniSat e Z3, anche se l'inclusione di Z3 è ancora in fase sperimentale. Per utilizzare un SAT-Solver è necessario creare Clausole e letterali appositi per la rappresentazione delle costanti proposizionali. Nella figura 3.7 è mostrata la struttura delle classi e delle interfacce per il SAT-Solver. La classe *SATLiteral* rappresenta una costante proposizionale ed è costituita da una coppia intero-booleano che rappresenta l'indice della costante e la sua polarità. La classe *SATClause* come la classe *Clause* è costituita da un Array, in questo caso di *SATLiteral*. La classe *Sat2FO* è uno dei componenti Built-in di Vampire che si occupa di convertire letterali e clausole del primo ordine (FO) in oggetti di tipo *SATLiteral* e *SATClause* (SAT) e viceversa. La chiamata della funzione *Sat2FO::toSat(Literal*)* aggiunge ad una bi-mappa il puntatore al letterale e lo associa ad un nuovo numero unsigned ≥ 1 se non è già presente altrimenti restituisce il numero già associato. La funzione *Sat2FO::toFO(SATLiteral*)* restituisce il puntatore al letterale associato al numero passato come argomento se presente altrimenti *nullptr*.

La classe astratta *SATSolver* rappresenta un generico SAT-Solver e contiene le funzioni virtuali comuni a tutti i SAT-Solver come *addClause(SATClause*)*, *solve* e *trueInAssignment(SATLiteral)* per aggiungere clausole, risolvere il problema e ottenere l'assegnamento delle variabili proposizionali (se soddisfacibile). Ogni variabile prima di essere utilizzata ha bisogno di essere 'registrata' tramite la funzione *newVar* che restituisce l'indice incrementale della nuova variabile registrata. Un altro metodo è quello di utilizzare la funzione *ensureVarCount(count)* che assicura che il numero di variabili registrate sia almeno pari a count. Se si è utilizzata la classe *Sat2FO* è possibile utilizzare la combinazione *SATSolver::ensureVarCount(Sat2FO::maxSATVar())* per assicurarsi che il numero di variabili registrate sia almeno pari al numero di costanti proposizionali utilizzate.

SATSolverWithAssumption estende *SATSolver* e permettere di aggiungere delle assunzioni (Dei letterali che devono essere veri nella soluzione). *PrimitiveProofRecordingSATSolver* estende *SATSolverWithAssumption* e definisce le funzioni per risalire alla dimostrazione di insoddisfacibilità. *MinisatInterfacing* è una classe che si occupa di interfacciare Vampire con la classe *Minisat* che contiene l'effettivo codice di Minisat.

Ad esempio si pensi di voler determinare la soddisfacibilità della formula CNF FO ground $\varphi := (p_1(f_1) \vee p_2 \vee \neg p_3) \wedge (\neg p_2 \vee \neg p_3) \wedge (\neg p_1(f_1))$. In primo luogo le clausole vengono divise in unità e rappresentate come Array di letterali:

$$unitList := [[p_1(f_1), p_2, \neg p_3], [\neg p_2, \neg p_3], [\neg p_1(f_1)]]$$

Applicando la funzione *Sat2FO::toSat(Clause*)* ad ogni clausola si ottiene una lista di SATClausole:

$$satUnitList := [[1, 2, -3], [-2, -3], [-1]]$$

A questo punto vanno registrate le variabili nel SAT-solver e aggiunte le clausole:

$satSolver = newSatSolver()$
$satSolver.ensureVarCount(sat2Fo.maxSATVar())$
for $c \in satUnitList$ **do**
| $satSolver.addClause(c)$

end

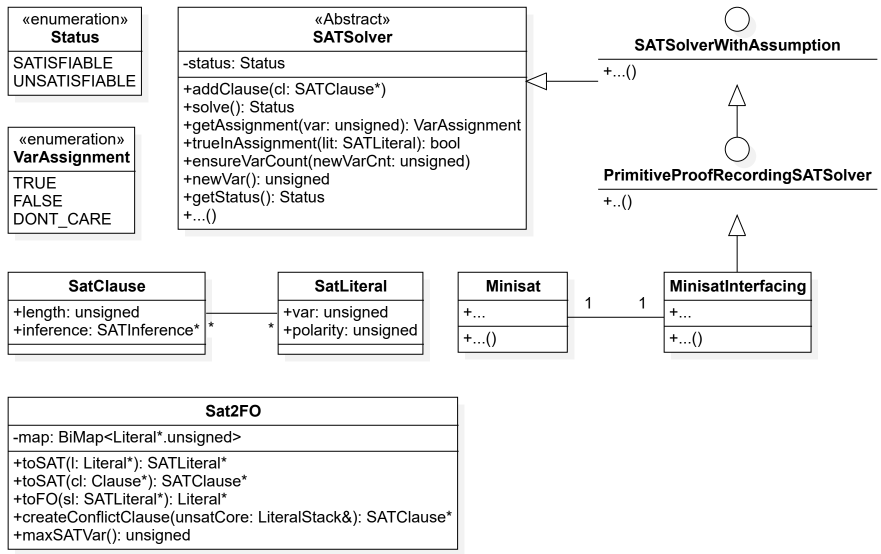


Figura 3.7: Classi e interfacce per il SAT-Solver

È possibile quindi chiamare la funzione *solve* del SAT-Solver per ottenere la soddisfacibilità della formula. In questo caso la formula è soddisfacibile un possibile assegnamento è $[1 \rightarrow false, 2 \rightarrow false, 3 \rightarrow false]$. SatSolver non ha una funzione per ottenere l'assegnamento direttamente ma è possibile ottenere l'assegnamento di ogni singola variabile tramite la funzione *trueInAssignment(SATLiteral)*.

$\alpha := EmptyMap < Literal*, bool > ()$
foreach $c \in unitList$ **do**
  foreach $l \in c$ **do**
    if $l.polarity()$ **then**
      $\alpha[l] := solver.trueInAssignment(sat2Fo.toSat(l))$
    end
  end
end

Per chiedere al SAT-Solver di cercare un altro assegnamento è possibile aggiungere una nuova clausola che rende l'assegnamento trovato incompatibile. Una clausola del genere è detta clausola bloccante (Blocking Clause) o clausola di conflitto (Conflict Clause). In questo caso una possibile clausola bloccante è $[1, 2, 3]$. Con l'aggiunta di questa clausola la formula diventa insoddisfacibile e il SAT-Solver restituirà *UNSATISFIABLE* alla chiamata di *solve*. Un modo per creare una clausola bloccante è quello di utilizzare la funzione Built-in di *Sat2FO::createConflictClause(LiteralStack)* che prende in input una lista di letterali e restituisce una SatClausola con i letterali negati.

3.7 Misurazione dei Tempi

Quando le performance sono un fattore critico è necessario avere un insieme di strumenti per misurare i tempi di esecuzione. Vampire mette a disposizione vari modi per misurare i tempi ed eseguire statistiche. In questa sezione ne verranno trattati essenzialmente tre. Il primo metodo più classico consiste semplicemente nel rilevare due tempi e calcolare la differenza. Questo può essere fatto utilizzando la funzione *elapsedMilliseconds* della classe *Timer* che restituisce il tempo in millisecondi trascorso dall'inizio dell'esecuzione del programma. Un timer globale è disponibile nell'oggetto globale *env* della classe *Lib/Environment*.

$t_{start} := env.timer \rightarrow elapsedMilliseconds()$
...
$t_{end} := env.timer \rightarrow elapsedMilliseconds()$
$\Delta t := t_2 - t_1$

Il secondo metodo consiste nell'utilizzare la macro *TIME_TRACE(name)* che misura il tempo trascorso tra l'invocazione e la fine del blocco di codice. 'name' è una stringa che di norma dovrebbe essere definita nella classe *Debug/TimeProfiling* con tipo *static constexpr const char* const*. È possibile chiamare più volte *TIME_TRACE* (con 'name' diversi) in più blocchi annidati e alla fine dell'esecuzione, con l'opzione *-tstat* attiva, Vampire stamperà un report con un albero delle chiamate e i tempi trascorsi, il numero di chiamate e il tempo medio per chiamata. Un esempio di report è mostrato in figura 3.8.

Il terzo metodo non è serve a misurare il tempo di esecuzione ma conta il numero di invocazioni. La macro *RSTAT_CTR_INC(name)* definita in *Debug/RuntimeStatistics* definisce un contatore associato ad ogni 'name' e lo incrementa di 1 ad ogni invocazione. Anche in questo caso Vampire stamperà un report alla fine dell'esecuzione con il formato *'name': 'count'*. Vampire utilizza questa macro ad esempio per contare il numero di clausole create e il numero di clausole eliminate. Un esempio di report è mostrato in figura 3.9.

```
===== start of time trace =====
[root] (total: 6772 µs, avg: 6772 µs, cnt: 1)
  ├──[61%] main loop (total: 4169 µs, avg: 4169 µs, cnt: 1)
  │    ├──[99%] run (total: 4149 µs, avg: 4149 µs, cnt: 1)
  │    │    ├──[58%] forward simplification (total: 2431 µs, avg: 29 µs, cnt: 83)
  │    │    │    ├──[94%] forward subsumption (total: 2295 µs, avg: 27 µs, cnt: 83)
  │    │    │    │    └──[45%] forward subsumption resolution (total: 1053 µs, avg: 15 µs, cnt: 70)
  │    │    │    ├──[ 0%] splitting component index usage (total: 6569 ns, avg: 96 ns, cnt: 68)
  │    │    │    ├──[ 0%] term sharing (total: 5989 ns, avg: 2994 ns, cnt: 2)
  │    │    │    └──[ 0%] splitting component index maintenance (total: 1265 ns, avg: 316 ns, cnt: 4)
  │    │    ├──[19%] activation (total: 823 µs, avg: 24 µs, cnt: 33)
  │    │    │    ├──[76%] clause generation (total: 628 µs, avg: 3344 ns, cnt: 188)
  │    │    │    │    ├──[66%] resolution (total: 419 µs, avg: 1559 ns, cnt: 269)
  │    │    │    │    │    └──[21%] term sharing (total: 91 µs, avg: 569 ns, cnt: 161)
  │    │    │    │    └──[ 0%] term sharing (total: 3489 ns, avg: 872 ns, cnt: 4)
  │    │    │    ├──[ 8%] add clause (total: 67 µs, avg: 2054 ns, cnt: 33)
  │    │    │    │    └──[85%] binary resolution index maintenance (total: 57 µs, avg: 1749 ns, cnt: 33)
  │    │    │    │         └──[ 8%] term sharing (total: 5064 ns, avg: 389 ns, cnt: 13)
  │    │    │    ├──[ 6%] clause selection (total: 51 µs, avg: 1563 ns, cnt: 33)
  │    │    │    │    └──[88%] literal selection (total: 45 µs, avg: 1381 ns, cnt: 33)
  │    │    │    ├──[ 0%] splitting (total: 1689 ns, avg: 51 ns, cnt: 33)
  │    │    │    └──[ 0%] redundancy check (total: 1669 ns, avg: 50 ns, cnt: 33)
  │    │    ├──[ 5%] passive container maintenance (total: 222 µs, avg: 2249 ns, cnt: 99)
  │    │    │    ├──[63%] forward subsumption index maintenance (total: 141 µs, avg: 2521 ns, cnt: 56)
  │    │    │    │    └──[ 9%] term sharing (total: 12 µs, avg: 359 ns, cnt: 36)
  │    │    │    └──[ 8%] unit clause index maintenance (total: 19 µs, avg: 1903 ns, cnt: 10)
  │    │    ├──[ 0%] immediate simplification (total: 32 µs, avg: 373 ns, cnt: 87)
  │    │    ├──[ 0%] SAT solver (total: 11 µs, avg: 5903 ns, cnt: 2)
  │    │    ├──[ 0%] backward simplification (total: 3697 ns, avg: 56 ns, cnt: 66)
  │    │    ├──[ 0%] minimizing solver time (total: 2094 ns, avg: 2094 ns, cnt: 1)
  │    │    └──[ 0%] splitting model update (total: 721 ns, avg: 721 ns, cnt: 1)
  │    └──[ 0%] init (total: 17 µs, avg: 17 µs, cnt: 1)
  ├──[23%] parsing (total: 1615 µs, avg: 1615 µs, cnt: 1)
  │    └──[ 1%] term sharing (total: 25 µs, avg: 387 ns, cnt: 67)
  ├──[ 7%] preprocessing (total: 527 µs, avg: 527 µs, cnt: 1)
  │    ├──[45%] property evaluation (total: 241 µs, avg: 120 µs, cnt: 2)
  │    ├──[ 5%] term sharing (total: 26 µs, avg: 714 ns, cnt: 37)
  │    └──[ 1%] naming (total: 9235 ns, avg: 577 ns, cnt: 16)
  └──[ 0%] sat proof minimization (total: 20 µs, avg: 20 µs, cnt: 1)
===== end of time trace =====
```

Figura 3.8: Esempio di report di Time Trace

```
----  Runtime statistics ----
clauses created: 93
clauses deleted: 19
ssat_new_components: 2
ssat_non_splittable_sat_clauses: 1
ssat_sat_clauses: 3
total_frozen: 1
-----------------------------
```

Figura 3.9: Esempio di report di Runtime Statistics

Capitolo 4

Implementazione di procedure di decisione per frammenti Binding in Vampire

In questo capitolo verrà descritto in che modo è stato implementato l'algoritmo di decisione per frammenti Binding introdotto nel capitolo 2 utilizzando gli strumenti e le funzionalità descritte nel capitolo 3 offerte da Vampire. Per ragioni di integrazione e manutenibilità del codice, si è deciso di limitare le modifiche alle funzioni e al Kernel di Vampire al minimo indispensabile, privilegiando l'impiego di componenti e funzionalità preesistenti. Questa decisione, tuttavia, ha comportato alcune complessità nell'implementazione, e questo è evidente nella sezione 4.1. Non tutti gli algoritmi standard di Vampire sono direttamente applicabili alle formule dei frammenti binding. Di conseguenza, anziché apportare modifiche dirette alle funzioni del kernel, si è optato per la creazione di strutture ausiliarie al fine di garantire la coerenza con la formula originale, sebbene ciò possa incidere sull'efficienza del sistema. Ad esempio si può notare che la procedura di preprocessing genera un elevato numero di nuovi letterali; tuttavia, mediante la modifica delle funzioni del kernel, sarebbe possibile ridurlo anche del 50%. Nonostante ciò, l'obiettivo primario di questo studio rimane confrontare l'approccio adottato con un approccio general-purpose basato su Resolution e GivenClause Architecture. È importante sottolineare che la fase di preprocessing, che è il componente meno ottimizzato, è esclusa dalla misurazione, pertanto non rappresenta un ostacolo significativo nel confronto tra i due approcci.

4.1 Preprocessing

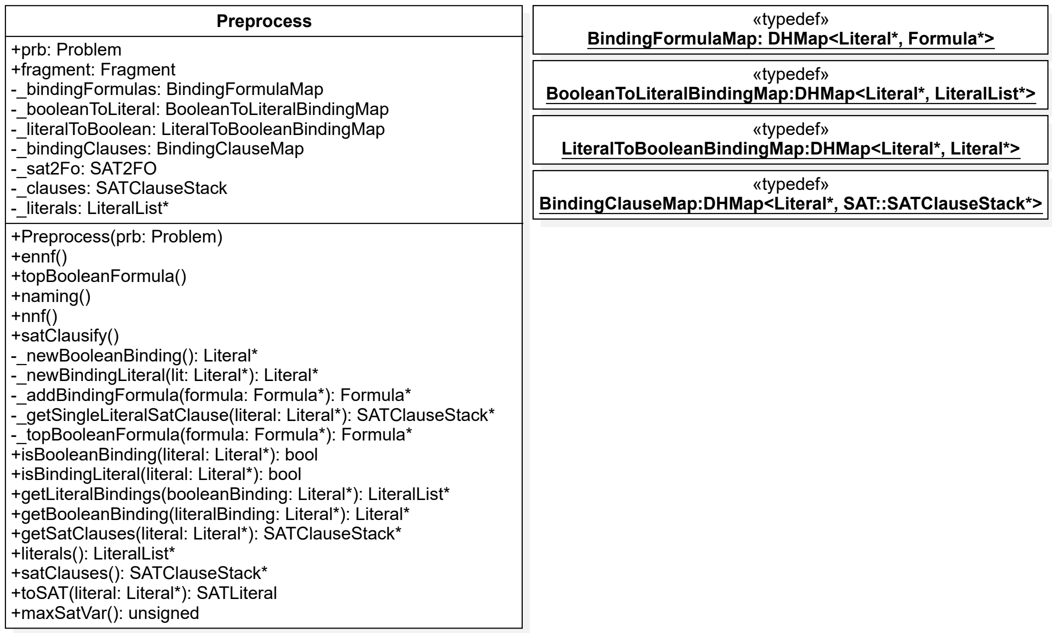


Figura 4.1: Struttura del Preprocessing

In questa sezione verrà descritto l'algoritmo di preprocessing utilizzato per trasformare una formula in input del frammento *1B* o *CB* in una struttura trattabile dall'algoritmo di decisione. Per utilizzare il SatSolver di Vampire per la ricerca degli implicanti è necessario clausificare la formula. Inoltre per evitare un esplosione esponenziale di formule causate dalle forme NNF e CNF è necessario utilizzare tecniche di naming. Qui sorgono i primi problemi visto che ne la clausificazione ne il naming sono processi conservativi rispetto ai frammenti. Ad esempio la semplice formula del frammento *1B* $\forall x_1(p_1(x_1)) \vee p_2$ clausificata diventa $\{\{p_1(x_1), p_2\}\}$ che fa parte del frammento *DB* . L'approccio utilizzato è stato quello di creare una nuova formula ground che rappresenta la struttura booleana esterna della formula originale, applicare le funzioni standard di preprocessing e mantenere una serie di strutture per risalire ai componenti originali. Per questo scopo viene introdotto un nuovo insieme di simboli di predicato $\Sigma_b = \{b_1, b_2, ...\}$. I predicati di Σ_b con arità 0 saranno chiamati *booleanBinding* e saranno associati ad una formula del frammento *1B* o *CB* . I predicati di Σ_b con arità $n > 0$ saranno chiamati *literalBindign* e fungeranno da rappresentati dei τ-Binding delle formule *1B* . Il preprocessing seguirà pressochè questa struttura:

1. Rettificazione
2. Trasformazione in ENNF

3. Creazione della formula booleana esterna (FBE) e associazione dei booleanBinding
4. Naming della FBE
5. Trasformazione in NNF della FBE
6. Creazione dei literalBinding e Sat-Clausificazione delle formule mappate dai booleanBinding
7. Creazione delle Sat-Clausole della FBE

La rettificazione e la trasformazione in ENNF sono processi conservativi rispetto ai frammenti e quindi verranno applicate direttamente le funzioni standard di Vampire. La creazione della FBE e l'associazione dei booleanBinding avviene tramite l'algoritmo 5.

Algorithm 5: Top Boolean Formula

Firma: topBooleanFormula(φ)
Input: φ una formula rettificata
Output: Una formula ground
GlobalData: bindingFormulas una mappa da booleanBinding a formula

```
switch φ do
    case Literal l do
        return new AtomicFormula(l);
    end
    case A[∧, ∨]B do
        return new JunctionFormula(topBooleanFormula(A), connective of φ,
         topBooleanFormula(B));
    end
    case ¬A do
        return new NegatedFormula(topBooleanFormula(A));
    end
    case [∀, ∃]A do
        b = newBooleanBinding();
        bindingFormulas[b] := φ;
        return new AtomicFormula(b);
    end
    case A[⇔, ⇒, ⊕]B do
        return new BinaryFormula(A, connective of φ, B);
    end
end
```

L'algoritmo prende in input una formula rettificata e restituisce una formula ground sostituendo le sottoformule quantificate con un nuovo booleanBinding aggiungendo la sottoformula originale alla mappa bindingFormulas. Da adesso in poi qualunque modifica fatta alla FBE preserverà l'appartenenza al frammento originale. Gli step successivi sono quindi applicare le funzioni standard di Vampire per il naming e la trasformazione in NNF. La trasformazione in NNF potrebbe portare alla negazione di qualche booleanBinding e va quindi aggiunta alla mappa bindingFormulas la formula negata associata.

foreach $l \in literals(\varphi)$ **do**
  if $\neg$*l.polarity()* **then**
    continue
  end
  $positiveFormula := bindingFormulas[positiveLiteral(l)]$
  $bindingFormulas[l] := newNegatedFormula(positiveFormula)$
end

A questo punto inizia il processo di SatClausificazione delle formule interne (quelle associate ai booleanBinding). Ogni letterale ground che non è un booleanBinding viene trasformato in una SatClausola di lunghezza 1 composta dal solo satLetterale associato al letterale.

foreach $l \in literals(\varphi)$ **do**
  if l *is not a booleanBinding* **then**
    $bindingClauses[l] := newSatClause\{toSat(l)\}$
  end
end

Per essere clausificate le formule della mappa bindingFormulas vanno trasformate in NNF, Skolemizzate. Anche in questo caso vengono utilizzate le funzioni standard di Vampire. Ogni booleanBinding è associato ad una formula del frammento ConjunctiveBinding, per questo dopo la skolemizzazione il quantificatore universale viene distribuito sull'and per ottenere le sottoformule del frammento OneBinding. Per ogni sottoformula OneBinding viene creato un nuovo LiteralBinding in rappresentaza della sottoformula. Il nuovo letterale avrà gli stessi termini del letterale più a sinistra della sottoformula (che sono gli stessi di tutti i letterali della sottoformula). Successivamente la formula viene SatClausificata. Si aggiunge alla mappa satClauses la coppia composta dal nuovo LiteralBinding e le satClausole della sottoformula. Alla mappa literalToBooleanBindings viene aggiunta la coppia composta dal nuovo LiteralBinding e il booleanBinding associato mentre alla mappa booleanBindingToLiteral viene aggiunta la coppia composta dal booleanBinding e la lista dei LiteralBinding che rappresentano le sottoformule della formula originale.

while $bindingFormulas \neq \emptyset$ **do**
$(booleanBinding, formula) := bindingFormulas.pop()$
$formula := nnf(formula)$
$formula := skolemize(formula)$
$toDo := \emptyset$

if *formula is ConjunctiveBinding* **then**
$formula := distributeForAll(formula)$
"Add each subformula to the todo list"
end
else
toDo.add(formula)
end
$literalBindings := \emptyset$
while $todo \neq \emptyset$ **do**
$subformula := todo.pop()$
$literalBinding := newLiteralBinding(subformula.mostLeftLiteral())$
$clauses := SatClausifyBindingFormula(subFormula)$

$satClauses[literalBinding] := clauses$
$literalToBooleanBindings[literalBinding] := booleanBinding$
$literalBindings.add(literalBinding)$

end
$booleanBindingToLiteral[booleanBinding] := literalBindings$
end

La funzione SatClausifyBindingFormula è una funzione che prende in input una formula la clausifica e converte tutte le clausole in SatClausole in modo che ogni satLetterale ha lo stesso indice del funtore del predicato associato. Questo è differente da quello che viene fatto dalla classe Sat2Fo che associa ogni puntatore a letterale ad un nuovo SatLetterale con un nuovo indice arbitrario. L'ultimo step è la SatClausificazione della FBE che avviene tramite le funzioni standard di Vampire della classe Sat2Fo. È importante ricordare che i satLetterali delle formule interne sono diversi dai satLetterali della FBE nonostante possano avere lo stesso indice.

Si prenda ad esempio la formula del frammento CB :

$$(\forall x_1, x_2((p_1(x_1)\vee p_2(x_1))\wedge p_2(f_1(x_2)))\wedge\forall x_1(p_3(x_1) \Rightarrow p1(x_1)))\vee(\forall x_1(p_2(x_1)) \Rightarrow p_4)$$

Il primo passo di preprocessing prevede la rettificazione e la trasformazione in ENNF. La formula è già rettificata mentre la trasformazione in ENNF porta all'eliminazione del $\Rightarrow$:

$$(\forall x_1, x_2((p_1(x_1)\vee p_2(x_1))\wedge p_2(f_1(x_2)))\wedge\forall x_1(\neg p_3(x_1)\vee p1(x_1)))\vee(\forall x_1(p_2(x_1)) \Leftrightarrow p_4)$$

La creazione della FBE porta alla generazione di un booleanBinding per ogni sottoformula quantificata:

$$(b_1 \wedge b_2) \vee (b_3 \Leftrightarrow p_4)$$

La mappa bindingFormulas contiene le seguenti coppie:

$b_1 \rightarrow \forall x_1, x_2((p_1(x_1) \vee p_2(x_1)) \wedge p_2(f_1(x_2)))$

$b_2 \rightarrow \forall x_1(\neg p_3(x_1) \vee p1(x_1))$

$b_3 \rightarrow \forall x_1(p_2(x_1))$

La formula ottenuta è troppo piccola per poter applicare il namig quindi si procede direttamente con la trasformazione in NNF:

$$(b_1 \wedge b_2) \vee ((\neg b_3 \vee p_4) \wedge (b_3 \vee \neg p_4))$$

Durante il processo di NNF il booleanBinding b_3 è stato negato e quindi va aggiunto alla mappa bindingFormulas:

$\neg b_3 \rightarrow \exists x_1(\neg p_2(x_1))$

A questo punto vengono trasformate in NNF e Skolemizzate le formule associate ai booleanBinding, vengono poi creati i literalBindings e le SatClausole delle formule interne. Il booleanBinding b_1 è associato ad una formula CB quindi viene distribuito il quantificatore universale sull'and e creati due literalBindings. La skolemizzazione della formula associata a $\neg b_3$ porta alla formula::

$\neg b_3 \rightarrow \neg p_2(sk_1)$

Vengono create così le mappe booleanBindingToLiteral e la sua inversa literal-ToBooleanBindings:

booleanBindingToLiteral	**literalToBooleanBindings**
$b_1 \to \{b_4(x_1), b_5(f_1(x_1))\}$	$b_4(x_1) \to b_1$
$b_2 \to \{b_6(x_1)\}$	$b_5(f_1(x_1)) \to b_1$
$b_3 \to \{b_7(x_1)\}$	$b_6(x_1) \to b_2$
$\neg b_3 \to \{b_8(sk_1)\}$	$b_7(x_1) \to b_3$
	$b_8(sk_1) \to \neg b_3$

Le formule associate ai literalBindings vengono clausificate:

- $\forall x_1, x_2((p_1(x_1) \vee p_2(x_1))) \to \{\{(p_1(x_1), p_2(x_1))\}\}$
- $\forall x_1, x_2(p_2(f_1(x_2))) \to \{\{p_2(f_1(x_2))\}\}$
- $\forall x_1(\neg p_3(x_1) \vee p1(x_1)) \to \{\{\neg p_3(x_1), p1(x_1)\}\}$
- $\forall x_1(p_2(x_1)) \to \{\{p_2(x_1)\}\}$
- $\neg p_2(sk_1) \to \{\{\neg p_2(sk_1)\}\}$

E successivamente SatClausificate e associate ai literalBindings:

- $b_4(x_1) \to \{\{s_1, s_2\}\}$
- $b_5(f_1(x_1)) \to \{\{s_2\}\}$
- $b_6(x_1) \to \{\{\neg s_3, s_1\}\}$
- $b_7(x_1) \to \{\{s_2\}\}$
- $b_8(sk_1) \to \{\{\neg s_2\}\}$

Gli ultimi due step sono la clausificazione della FBE:

$$\{\{b_1, \neg b_3, p_4\}, \{b_2, \neg b_3, p_4\}, \{b_1, b_3, \neg p_4\}, \{b_2, b_3, \neg p_4\}\}$$

E la SatClausificazione tramite sat2Fo:

$$\{\{s_1, \neg s_2, s_3\}, \{s_4, \neg s_2, s_3\}, \{s_1, s_2, \neg s_3\}, \{s_4, s_2, \neg s_3\}\}$$

Che crea internamente una bi-mappa che associa ogni satletterale ad un letterale:

- $s_1 \leftrightarrow b_1$
- $s_2 \leftrightarrow \neg b_3$
- $s_3 \leftrightarrow p_4$
- $s_4 \leftrightarrow b_2$

4.2 Procedura di Decisione

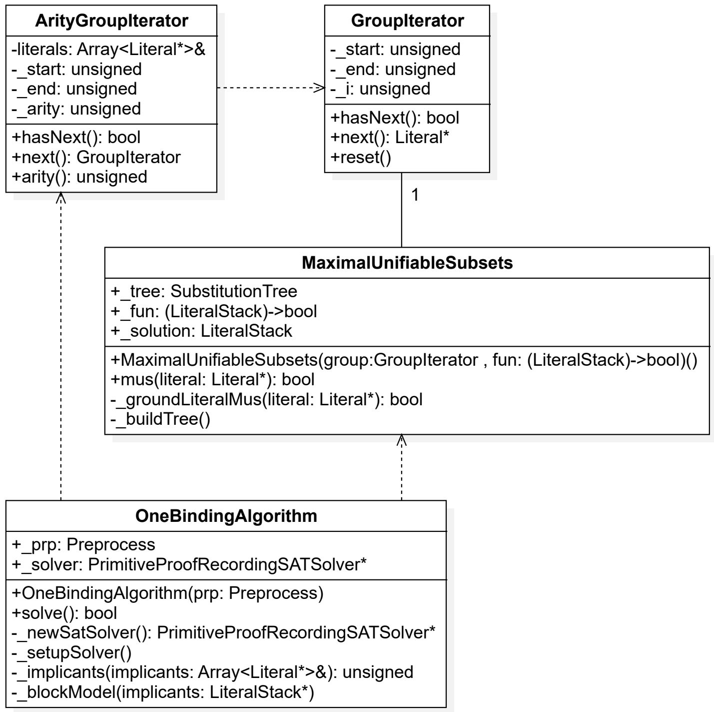


Figura 4.2: Struttura dell'algoritmo di decisione

In questa sezione verrà descritta l'implementazione dell'algoritmo di decisione 2 per frammenti Binding descritto nel capitolo 2. L'algoritmo è composto da tre parti principali: la ricerca degli implicanti, la ricerca di tutti i sottoinsiemi unificabili e la ricerca di un assegnamento proposizionale che soddisfi la congiunzione della conversione booleana dei sottoinsiemi unificabili. Da questo momento si assuma di avere una formula preprocessata con tutte le strutture ausiliarie come descritto nella sezione precedente.

La ricerca degli implicanti è la parte più facile da implementare (sat esterna). Data la FBE SatClausificata è sufficiente utilizzare il satSolver integrato in Vampire ed estrapolarne una assegnazione. Dopo aver ottenuto l'insieme degli implicanti proposizionali se la sua relativa formula del primo ordine è insoddisfacibile allora è sufficiente creare una clausola bloccante e cercare un nuovo assegnamento. Se non sono disponibili nuovi assegnamenti allora la formula originale è insoddisfacibile.

La ricerca di tutti i sottoinsiemi unificabili è senza dubbio la parte più complessa dell'algoritmo. L'approccio utilizzato nell'algoritmo 2 è troppo astratto e non utilizzabile nella pratica. Anche il solo problema di iterare su tutti i sottoinsiemi di un insieme è un problema non trivale. Vanno quindi necessariamente fatti dei tagli nello spazio di ricerca. La prima osservazione che si può fare è che se un insieme di letterali è unificabile allora i letterali hanno tutti la stessa arità. È quindi possibile ordinare l'insieme di implicanti in base all'arità e ricercare per ogni 'Gruppo di Arità' tutti i sottoinsiemi unificabili. Già in questo modo si riduce notevolmente lo spazio di ricerca eliminando tutti quei sottoinsiemi composti da letterali di arità diversa. La seconda osservazione è che dati due sottoinsiemi $U' \subseteq U$ se la congiunzione della conversione booleana dei letterali di U è soddisfacibile allora lo sarà anche quella di U'. Questo riduce ulteriormente lo spazio di ricerca ai soli sottoinsiemi massimali unificabili. Sfortunatamente la ricerca di tutti i sottoinsiemi massimali unificabili (Maximal Unifiable Subsets / MUS) è un problema NP-Completo così come il suo problema complementare cioè il problema di ricercare tutti i sottoinsiemi minimali non unificabili (minimal non unifiable subsets / mnus). Per questo motivo è stato creato un algoritmo euristico meno restrittivo che itera almeno su tutti i sottoinsiemi massimali non escludendo però la possibilità di trovare anche qualche sottoinsieme non massimale.

Dopo aver ottenuto un insieme di τ-Binding unificabili l'algoritmo procede con la ricerca di un assegnamento proposizionale che soddisfi la congiunzione della conversione booleana dei sottoinsiemi unificabili (sat interna). Anche in questo caso il problema è molto semplice. Ogni τ-Binding è rappresentato da un bindingLiteral creato nella fase di preprocessing e ogni bindingLiteral è associato ad un insieme di satClausole che rappresentano la conversione booleana citata sopra. Grazie a questa indicizzazione è possibile utilizzare il satSolver integrato per verificare la soddisfacibilità.

Maximal Unifiable Subsets

Per la ricerca dei mus è stato implementato un algoritmo ricorsivo che in modo incrementale costruisce un sottoinsieme di letterali unificabile. L'algoritmo sfrutta un SubstitutionTree per la ricerca degli unificatori e una mappa S che rappresenta la funzione caratteristica dell'insieme soluzione. In particolare per ogni letterale x se $S[x] = 1$ allora fa parte della soluzione, se $S[x] = 0$ allora non fa parte della soluzione e infine se $S[x] = -1$ allora vuol dire che non fa parte della soluzione e deve essere escluso dalle ricerche future. Prima di iniziare la ricerca va impostato l'ambiente in modo tale che il SubstitutionTree contenga tutti i letterali del gruppo di arità corrente e S mappi tutti i letterali a 0. Viene fornita anche una funzione *fun* che prende in input l'insieme soluzione e restituisce un booleano. La funzione 6 è la funzione che inizia la catena di chiamate ricorsive.

Algorithm 6: Maximal Unifiable Subsets

Firma: mus($literal$)
Input: $literal$ un puntatore ad un letterale
Output: $\top$ o $\bot$
GlobalData: S una mappa da letterali a interi
if $S[literal] \neq 0$ **then**
  | **return** $\top$;
end
if *literal is ground* **then**
  | **return** $groundLiteralMus(literal)$;
end
$S[literal] = 1$;
$tmpToFree := \emptyset$;
$res := mus(literal, tmpToFree)$;
foreach $i \in tmpToFree$ **do**
  | $S[i] = -1$;
end
$S[literal] = -1$;
return res;

Inizialmente verifica se il letterale è già stato esplorato e in quel caso restituisce $\top$. Se il letterale è ground allora chiama la funzione 8 che è un'ottimizzazione pensata per semplificare la ricerca con letterali ground. Se il letterale non è ground allora si inizia la vera e propria ricerca dei mus. Viene impostato il valore del letterale nella mappa S ad 1 in modo tale che faccia parte della soluzione e chiama la funzione 7 che prende in input il letterale e un insieme di letterali.

Algorithm 7: Maximal Unifiable Subsets

Firma: mus(*literal*, *FtoFree*)

Input: *literal* un puntatore ad un letterale, *FtoFree* un puntatore ad una lista di letterali

Output: ⊤ o ⊥

GlobalData: **S** una mappa da letterali a interi, **fun** una funzione da lista di letterali a bool, **tree** un SubstitutionTree

```
isMax := ⊤;
uIt = tree.getUnifications(query : literal, retrieveSubstitutions : true);
toFree := ∅;
while uIt.hasNext() do
    (u, σ) := uIt.next();
    if S[u] = 0 then
        S[u] = 1;
        l := literal^σ;
1       if l = literal then
            u' := u^σ;
            if u' = u then
                FtoFree := FtoFree ∪ {u};
            end
            else
                toFree := toFree ∪ {u};
            end
        end
        else
            isMax = ⊥;
            tmpToFree := ∅;
            if ¬mus(l, tmpToFree) then
                return ⊥;
            end
            S[u] = −1;
            foreach i ∈ tmpToFree do
                S[i] = −1;
            end
            toFree := toFree ∪ {u} ∪ tmpToFree;
        end
    end
end
if isMax then
    if ¬fun({x | S[x] = 1}) then
        return ⊥;
    end
end
while toFree ≠ ∅ do
    S[toFree.pop()] = 0;
end
return ⊤;
```

La funzione comincia inizializzando la variabile $isMax$ a $\top$ che rappresenta il fatto che il sottoinsieme è massimale. Se non vengono effettuate chiamate ricorsive allora $isMax$ non viene modificato e viene chiamata la funzione *fun* sull'insieme soluzione. Successivamente viene chiesto al SubstitutionTree di restituire un iteratore su tutti i letterali unificabili con il letterale in input. Viene inizializzata una lista $toFree$ che conterrà tutti gli elementi che verranno bloccati su questo livello dell'albero delle chiamate ricorsive.

Per capire meglio questo aspetto dell'algoritmo si consideri un insieme di letterali $\{l_1, ..., l_n\}$. Un modo di ottenere tutti i mus di questo insieme, che è anche il modo che è stato implementato, è quello di cercare tutti i mus che contengono l_1, tutti i mus che contengono l_2 e così via. Si supponga di aver già trovato tutti i mus che contengono l_1 e di voler cercare tutti i mus che contengono l_2. Se l'algoritmo rileva che l_2 è unificabile con l_1 tramite la sostituzione σ, dovrebbe inserire l_1 nella soluzione e cercare tutti i mus che contengono l_2^σ e così via. Ma si può notare che mus di questo tipo sono già stati esplorati quando si cercavano i mus che contenevano l_1. Per questo motivo in modo da evitare di ripetere del lavoro già svolto, alla fine della ricerca dei mus che contengono l_1, il letterale viene bloccato ($S[l_1] = -1$) e viene aggiunto ad una lista $toFree$. In generale per ogni l_x vengono cercati tutti i mus che contengono l_x escludendo dalla ricerca i letterali l_y con $y < x$. Una volta arrivati ad l_n si liberano ($S[l_{(...)}] = 0$) tutti i letterali bloccati in $toFree$.

Tornando alla descrizione dell'algoritmo, dopo aver inizializzato la lista $toFree$ si itera su tutti i letterali che unificano con il letterale in input $literal$. Per ogni letterale u se è già contenuto nella soluzione, o è stato bloccato, si ignora, altrimenti viene aggiunto alla soluzione. Si calcola il letterale $l = literal^\sigma$ ottenuto applicando la sostituzione σ al letterale $literal$. Se il letterale l è uguale a $literal$, cioè la sostituzione non ha apportato nessun cambiamento, allora si evita di effettuare una chiamata ricorsiva su l in quanto è possibile utilizzare lo stesso iteratore di $literal$. Se anche u^σ è uguale a u allora viene aggiunto alla lista $FtoFree$ passata in input. Questo perchè u ha esattamente gli stessi termini di $literal$ quindi tutti i mus che contengono u contengono anche $literal$. u va quindi rimosso/bloccato/sbloccato dalla soluzione esattamente quando viene rimosso/bloccato/sbloccato il letterale con cui è stata fatta l'unificazione al livello superiore che ha poi generato $literal$. In caso contrario viene aggiunto alla lista $toFree$ per essere rimosso alla fine dell'esecuzione del livello corrente.

Se il letterale l è diverso da $literal$ non è detto che la soluzione corrente sia massimale, quindi si imposta $isMax$ a $\bot$ e viene effettuata una chiamata ricorsiva dando come parametri l e una lista temporanea $tmpToFree$. Nel caso la chiamata ricorsiva restituisca $\bot$ allora la funzione propaga il risultato negativo restituendo $\bot$. Dopo la chiamata ricorsiva il letterale u viene rimosso dalla soluzione e bloccato per questo livello della ricorsione. Viene poi aggiunto alla lista $toFree$ per essere sbloccato alla fine dell'esecuzione del livello corrente. Vengono anche bloccati tutti i letterali restituiti dalla chiamata ricorsiva tramite la lista $tmpToFree$ e aggiunti a $toFree$ per essere sbloccati alla fine dell'esecuzione del livello corrente.

Alla fine dell'iterazione sui letterali unificabili se $isMax$ è $\top$ allora si compone la soluzione e viene chiamata la funzione *fun*. Se *fun* restituisce $\bot$ allora si restituisce $\bot$. Altrimenti si liberano i letterali della lista *toFree* e si restituisce $\top$.

Algorithm 8: Maximal Unifiable Subsets Ground

Firma: groundMus($literal$)
Input: $literal$ un puntatore ad un letterale ground
Output: $\top$ o $\bot$
GlobalData: **S** una mappa da letterali a interi, **fun** una funzione da lista di letterali a bool, **tree** un SubstitutionTree

```
if S[literal] ≠ 0 then
    return ⊤;
end
uIt = tree.getUnifications(query : literal, retrieveSubstitutions : true);
solution := ∅;
while uIt.hasNext() do
    (u, σ) := uIt.next();
    if S[u] = 0 then
        if u is ground then
            S[u] = −1;
        end
        solution := solution ∪ {u};
    end
end
return fun(solution);
```

La funzione 8 è un'ottimizzazione della funzione 7 per letterali ground. Si consideri un letterale ground g. Per qualunque sostituzione di variabili σ il letterale g^σ sarà sempre uguale a g. Quindi per qualunque letterale u se $u^\sigma = g^\sigma$ allora $u^\sigma = g$. Ciò significa che l'unico mus di g è proprio l'insieme di tutti i letterali che unificano con g. Il costo di questa funzione è pari al costo della visita nel SubstitutionTree che viene effettuata con la funzione *getUnifications* e l'iteratore *uit*, più il costo della chiamata della funzione *fun*. In linea di massima molto più conveniente rispetto alla funzione 7 che può effettuare potenzialmente un numero esponenziale di chiamate ricorsive.

Procedura di decisione

Algorithm 9: Algoritmo di decisione

```
Firma: solve(prp)
Input: prp il problema pre-processato
Output: ⊤ o ⊥
satSolver := newSatSolver();
satSolver.addClauses(prp.clauses);
while satSolver.solve() = SATISFIABLE do
    res := ⊤;
    implicants := getImplicants(satSolver, prp);
    implicants := sortImplicants(implicants);
1   if implicants contains only ground Literals then
        return ⊤;
    end
    agIt := ArityGroupIterator(implicants);
    while res And agIt.hasNext() do
        maximalUnifiableSubsets := SetupMus(group, internalSat);
        foreach lit ∈ group do
            if ¬maximalUnifiableSubsets.mus(lit) then
                res := ⊥;
2               blockModel(maximalUnifiableSubsets.getSolution());
                Break;
            end
        end
3       if res = ⊤ then
            return ⊤;
        end
    end
end
return ⊥;
```

Dato il problema preprocessato l'algoritmo comincia impostando il satSolver con le satClausole ottenute nella fase di preprocessing. Se la formula è soddisfacibile allora si recupera l'insieme degli implicanti tramite la funzione getImplicants 10.

Algorithm 10: getImplicants

Firma: getImplicants(solver, prp)
Input: *solver* un sat solver, *prp* il problema pre-processato
Output: Una lista letterali

```
implicants := ∅;
foreach l ∈ prp.literals() do
    satL := prp.toSat(l);
    if solver.trueInAssignment(satL) then
        if prp.isBooleanBinding(l) then
            implicants := implicants ∪ prp.getLiteralBindings(l);
        end
        else
            implicants := implicants ∪ {l};
        end
    end
end
return implicants;
```

Per ogni letterale del problema viene recuperato il corrispondente satLetterale. Se il satLetterale è soddisfatto dall'assegnamento allora se il letterale originale non è un booleanBinding viene aggiunto all'insieme di implicanti, altrimenti vengono aggiunti tutti i literalBinding associati al booleanBinding.

Dopo aver ottenuto l'insieme di implicanti, l'insieme viene ordinato in base all'arità dei letterali. La funzione sortImplicants può essere estesa aggiungendo varie euristiche. La prima euristica che è stata implementata è quella di spingere i letterali ground all'inizio di ogni gruppo di arità in modo da utilizzare l'algoritmo 8 per ridurre il numero di chiamate ricorsive che verrebbero fatte dall'algoritmo 7. La seconda euristica è quella di ordinare i letterali in base ai sottotermini in modo da avere vicino sequenze di letterali che hanno stessi termini per sfruttare l'ottimizzazione vista nell'algoritmo 7.

Se l'insieme di implicanti contiene solo letterali ground (che non sono literalBindings) allora la formula è soddisfacibile perchè l'assegnamento per la formula esterna è valido anche per le formule interne e l'algoritmo termina restituendo $\top$.

Altrimenti per ogni gruppo di arità si imposta l'ambiente per la ricerca dei mus e viene chiamata la funzione mus 6 per ogni letterale del gruppo. Se una di queste chiamate restituisce $\bot$ allora la formula FO corrispondente all'assegnamento booleano trovato è insoddisfacibile e si procede con la ricerca di un nuovo assegnamento. Se tutte le chiamate restituiscono $\top$ allora la formula è soddisfacibile e l'algoritmo termina restituendo $\top$. Se non sono disponibili nuovi assegnamenti allora la formula è insoddisfacibile e l'algoritmo termina restituendo $\perp$.

Algorithm 11: Sat interna

Firma: internalSat(literals)
Input: *literals* una lista di letterali
Output: $\top$ o $\perp$
if *literals.length = 1 And getSatClauses(literals.top()).length = 1* **then**
  | **return** $\top$;
end
$satSolver := newSatSolver()$;
foreach $l \in literals$ **do**
  | $satSolver.addClause(getSatClauses(l))$;
end
return $satSolver.solve() = SATISFIABLE$;

La funzione internalSat viene chiamata ogni volta che la funzione 7 trova un nuovo mus. Se il mus è composto da un solo letterale e la lista di satClausole associata è composta da una sola clausola allora la formula non può entrare in contraddizione, di conseguenza è soddisfacibile e la funzione restituisce $\top$, evitando di impostare il satSolver. In caso contrario viene impostato il satSolver con le clausole associate ai literalBindings dalla mappa satClauses e viene chiamato il metodo solve. La funzione restituisce $\top$ se il satSolver restituisce SATISFIABLE altrimenti $\perp$.

Algoritmo ottimizzato e algoritmo Naive

Nel corso della progettazione de dello sviluppo sono state effettuate diverse modifiche e ottimizzazioni rispetto all'algoritmo pensato inizialmente. Nel prossimo capitolo sull'analisi dei tempi, quando si farà riferimento all'algoritmo ottimizzato ci si riferirà all'algoritmo descritto in questo capitolo, mentre quando si farà riferimento all'algoritmo Naive ci si riferirà all'algoritmo che non sfrutta le euristiche descritte nelle sezioni precedenti. In particolare l'algoritmo naive non include i blocchi di codice numerati (1) e (2) nell'algoritmo 6 e Il blocco numerato con (1) nell'algoritmo 7. Inoltre nell'algoritmo 9 il blocco numerato con (1) era posto fuori dal ciclo while e veniva controllata se tutta la formula fosse ground. In tal caso veniva restituito direttamente il valore della funzione solve del SatSolver. Il sorting degli implicanti nell'algoritmo naif è effettuato solo in base all'arità dei letterali, mentre nell'algoritmo ottimizzato vengono spinti i letterali ground all'inizio di ogni gruppo di arità e vengono ordinati in base ai sottotermini. Come ultima modifica, sempre sull'algoritmo 9, la riga numerata con (2) era precedentemente posta dopo il blocco numerato con (3) e veniva usato tutto l'insieme di implicanti per generare la clausola bloccante al posto dell'insieme risultato dalla funzione mus. Maggiori informazioni sulle motivazioni e sull'effetto di queste modifiche verranno discusse nel prossimo capitolo.

4.3 Algoritmo di Classificazione

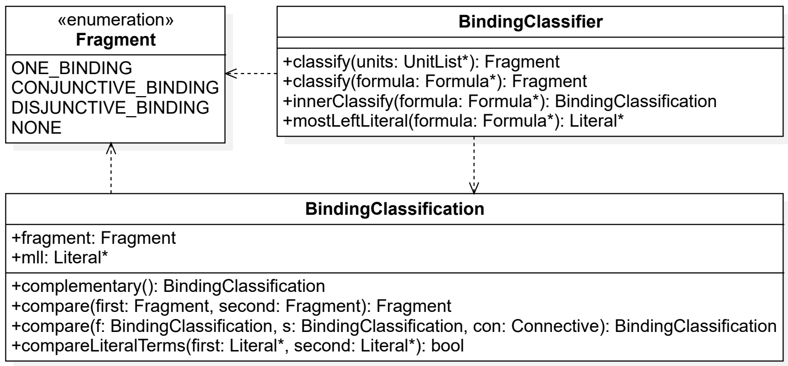


Figura 4.3: Classificatore

La precondizione più importante per la correttezza dell'algoritmo di decisione è che la formula non processata faccia parte del frammento CB . Per questo motivo è stato creato un classificatore in grado di stabilire se una formula è risolvibile o no dalla procedura. L'algoritmo non fa altro che verificare la forma sintattica della formula e capisce a quale grammatica della sezione 2.1 appartiene. Per questo scopo sono state create due funzioni chiamate Classificatore Esterno (Algoritmo 12) e Classificatore Interno (Algoritmo 14). La prima verifica la parte della formula senza quantificatori mentre la seconda verifica la parte interna ai quantificatori e confronta i termini dei letterali. Entrambi gli algoritmi hanno una struttura di visita dell'albero sintattico in postOrder e hanno una complessità lineare rispetto alla dimensione della formula.

Algorithm 12: Classificatore esterno

```
Firma: classify(φ) Input: φ Una formula rettificata
Output: Un elemento dell'enumerazione Fragment
switch φ do
    case Literal do
        return ONE_BINDING;
    end
    case A[∧, ∨]B do
        return compare(classify(A), classify(B));
    end
    case ¬A do
        return classify(A).complementary();
    end
    case [∀, ∃]A do
        sub := φ;
        connective := connective of φ;
        repeat
            sub := subformula of sub;
            connective := connective of sub;
        until connective ∉ {∀, ∃};
        (fragment, _) := innerClassify(sub);
        return fragment;
    end
    case A ⇔ B do
        return compare(classify(A ⇒ B), classify(B ⇒ A));
    end
    case A ⊕ B do
        return classify(A ⇔ B).complementary();
    end
    case A ⇒ B do
        return compare(classify(¬A), classify(B));
    end
end
```

Algorithm 13: Compare esterno

Firma: compare(A, B) **Input:** A, B due elementi dell'enumerazione Fragment

Output: Un elemento dell'enumerazione Fragment

if $A = B$ **then**
  return A;
end
if $One\_Binding \notin \{A, B\}$ **then**
  return $None$;
end
return $max(A, B)$;

Il classificatore esterno si appoggia ad una funzione ausiliaria chiamata *compare* che prende in input due elementi dell'enumerazione Fragment e restituisce il frammento risultante dalla combinazione booleana ($\wedge, \vee$) dei due frammenti. La combinazione di due *1B* è sempre un *1B* mentre la combinazione di un *1B* con un *CB* o *DB* è sempre un *CB* o *DB* . Infine la combinazione di un *CB* con un *DB* fa parte del frammento Boolean Binding che però in questa sezione verrà chiamato *None*. Per la comparazione è stato creato un ordinamento dei frammenti che segue una struttura a rombo:

$$\begin{array}{c} None \\ / \quad \backslash \\ CB \quad DB \\ \backslash \quad / \\ 1B \end{array}$$

Dove *1B* è il minimo e *None* è il massimo. Il risultato è un reticolo e la funzione compare restituisce l'estremo superiore dei due frammenti. La funzione complementary restituisce il frammento della negazione di una formula di un determinato frammento. In particolare il complementare di un *1B* è *1B* mentre il complementare di un *CB* è *DB* e viceversa.

Algorithm 14: Classificatore interno

Firma: innerClassify(φ) **Input:** φ Una formula rettificata
Output: Una coppia (Fragment, Literal)

```
switch φ do
    case Literal l do
        return (ONE_BINDING, l);
    end
    case A[∧, ∨]B do
        return
         innerCompare(innerClassify(A), innerClassify(B), connective of φ);
    end
    case ¬A do
        return innerClassify(A).complementary();
    end
    case A[⇒, ⇔, ⊕]B do
        return
         innerCompare(innerClassify(A), innerClassify(B), connective of φ);
    end
    else
        return (None, null);
    end
end
```

La struttura del classificatore interno è molto simile a quella del classificatore esterno, mentre il comparatore interno è leggermente più complesso. Il caso base è quando la formula è un singolo letterale che è sempre un *1B* . La visita in postOrder restituisce una coppia (Fragment, Literal) che rappresenta il frammento a cui appartiene la formula e un letterale di rappresentanza della formula in questo caso il letterale più a sinistra. Il letterale serve a mantenere una reference alla lista di termini delle formule del frammento *1B* .

Algorithm 15: Compare interno

```
Firma: innerCompare(A, B, con) Input: A, B due coppie (Fragment,
        Literal), con un connettivo
Output: Una coppia (Fragment, Literal)
switch A.first, B.first, con do
    case One_Binding, One_Binding, _ do
        if A.second has same terms of B.second then
            return A;
        end
        else if conn = ∧ then
            return (Conjunctive_Binding, null);
        end
        else if conn = ∨ then
            return (Disjunctive_Binding, null);
        end
    end
    case [One_Binding, Conjunctive_Binding | Conjunctive_Binding,
     One_Binding], ∧ do
        return (Conjunctive_Binding, null);
    end
    case [One_Binding, Disjunctive_Binding | Disjunctive_Binding,
     One_Binding], ∨ do
        return (Disjunctive_Binding, null);
    end
    case Conjunctive_Binding, Conjunctive_Binding, ∧ do
        return (Conjunctive_Binding, null);
    end
    case Disjunctive_Binding, Disjunctive_Binding, ∨ do
        return (Disjunctive_Binding, null);
    end
end
return (None, null);
```

La combinazione booleana di due frammenti *1B* (all'interno di un quantificatore) può portare a tre diversi risultati. Se i termini dei letterali di rappresentanza sono uguali allora la combinazione è ancora un *1B* altrimenti la combinazione è un *CB* se il connettivo è $\wedge$ e un *DB* se il connettivo è $\vee$. Il termine *null* viene usato come sostituto del letterale di rappresentanza in formule del frammento *CB* e *DB* in quanto sono una combinazione di più *1B* e non hanno un letterale di rappresentanza. Due frammenti *CB* rimangono *CB* solo se il loro connettivo è $\wedge$. La combinazione di un *1B* con un *CB* è un *CB* se il connettivo è $\wedge$. Stesso discorso per i DB e il connettivo $\vee$. In tutti gli altri casi la combinazione è *None*. Nell'algoritmo 15 sono stati omessi i casi con connettivi $\Rightarrow, \Leftrightarrow, \oplus$ in quanto non sono riconducibili a formule composte da $\wedge$ e $\vee$ come è stato fatto ad esempio nell'algoritmo 12. Con la funzione complementary applicata ad una coppia: (Fragment, Literal).complementary() si intende la coppia (Fragment.complementary(), Literal).

Capitolo 5

Analisi Sperimentale

5.1 La libreria TPTP

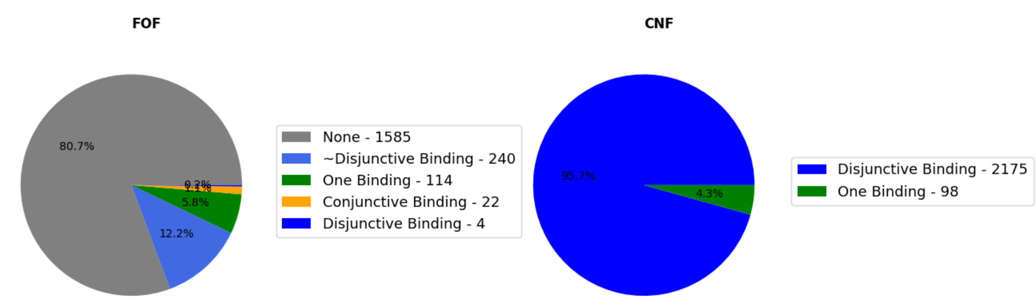


Figura 5.1: Classificazione Libreria TPTP fof e cnf senza uguaglianza

Per verificare la correttezza e l'efficienza dell'algoritmo implementato è stata scelta la libreria di problemi TPTP [10] come dataset di problemi da risolvere. La libreria TPTP è una collezione di problemi per sistemi ATP dove ogni problema è scritto in formato TPTP come descritto nella sezione 1.5. I problemi TPTP sono suddivisi in base al dominio di appartenenza e il formato (fof, cnf, tff, ecc). Il nome dei file segue il seguente schema:

```
<domain><number>[+,-]{.<variant>}.p
```

dove '<domain>' è il dominio di appartenenza del problema composto da tre lettere maiuscole e '<number>' è un numero, generalmente di tre cifre che identifica il problema all'interno del dominio. Il simbolo + indica che il problema è scritto con la sintassi *fof* mentre il simbolo - indica che il problema è scritto con la sintassi *cnf*. '.<variant>' è un suffisso opzionale che identifica una variante del problema ed in generale è un numero a tre cifre. Ad esempio un nome valido è *SYN001+1.003.p* che indica il problema 1 del dominio *Syntactic*, in formato *fof*, variante 3.

Non tutti i problemi sono adatti per essere risolti con l'algoritmo implementato ed è stato quindi necessario filtrare i problemi in base a determinate caratteristiche. In primo luogo sono stati scartati tutti i problemi non in formato *fof* o *cnf* e i problemi con uguaglianza. Questo è stato possibile tramite il comando *TPTP2T*:

- Per i problemi fof: `tptp2T -q2 -pps Form FOF -Equality`
- Per i problemi cnf: `tptp2T -q2 -pps Form CNF -Equality`

Il risultato delle due query ha restituito una lista di **1969** problemi in formato *fof* e **2274** problemi in formato *cnf*. Da entrambe le liste è stato scartato un problema puramente proposizionale, quindi poco significativo per la sperimentazione, ma estremamente grande da rallentare l'intero set di benchmark, riducendo il numero di potenziali problemi utili a **1965** per i problemi *fof* e **2273** per i problemi *cnf*.

Successivamente tutti i problemi sono stati classificati tramite il classificatore descritto nella sezione 4.3. Si ricorda che i problemi sono dati in formato $(A_1 \wedge A_2 \wedge ... \wedge A_n) \Rightarrow C$ dove $A_1, A_2, ..., A_n$ sono assiomi e C è la congettura, mentre sia l'algoritmo di decisione che Vampire lavorano sul problema negato ovvero $A_1 \wedge A_2 \wedge ... \wedge A_n \wedge \neg C$. Per classificazione di un problema P con congettura si intende quindi il frammento di appartenenza del problema negato $\neg P$. Nelle formule in cui non è presente la congettura, come quelle del formato *cnf*, per classificazione della formula P si intende il frammento di appartenenza del problema non negato $P = A_1 \wedge ... \wedge A_n$. I risultati della classificazione sono mostrati in figura 5.1.

Riguardo i problemi *fof*, dei 1965 problemi analizzati:

- **1585** sono stati classificati come *None* e quindi totalmente inutilizzabili.
- **114** sono *One Binding* e **22** sono *Conjunctive Binding*, quindi adatti per l'algoritmo.
- **244** sono *Disjunctive Binding* e quindi non adatti per l'algoritmo.

Dei 244 problemi *Disjunctive Binding*: **240** contengono la congettura mentre gli altri **4** non la contengono. I 240 problemi con la congettura sono stati recuperati negandoli in modo tale da portarli nel formato $\neg A_1 \vee ... \vee \neg A_n \vee C$ che fa parte del frammento *Conjunctive Binding* e quindi adatto per l'algoritmo (questi problemi nella figura 5.1 sono chiamati ∼*Disjunctive Binding*). Questo porta ad un numero totale di **376** problemi utili in formato *fof*.

Riguardo i problemi *cnf*, la suddivisione è più netta. Tutte le formule CNF sono infatti o del frammento *One Binding*, se per ogni clausola tutti i letterali della clausola hanno la stessa lista di termini, o altrimenti del frammento *Disjunctive Binding*. Dei 2273 problemi analizzati:

- **98** sono del frammento *One Binding*
- **2175** sono del frammento *Disjunctive Binding*

In questo caso la negazione delle formule *Disjunctive Binding* porterebbe a problemi puramente proposizionali e quindi poco utili per la sperimentazione. Il totale dei problemi utili in formato *cnf* è quindi di **98**. Per un totale di **474** problemi utili. La lista dei 474 problemi è riportata nell'appendice A. Ogni problema è stato numerato nella tabella A.1 per essere facilmente identificato.

5.2 Analisi dei risultati

In questa sezione verranno analizzati i risultati dell'esecuzione dell'algoritmo sui problemi selezionati nel paragrafo precedente e confrontati con i risultati ottenuti da Vampire. L'obbiettivo della sperimentazione è confrontare l'efficienza di un algoritmo gneral purpose basato su Resolution come quello implementato da Vampire con un algoritmo specializzato SMT come quello implementato. Vampire implementa numerose strategie, euristiche e inferenze di semplificazione per essere efficiente a livello competitivo quindi per indurlo a comportarsi il più possibile come il modello della *Given Clause* descritta nella sezione 3.4 è stato necessario disabilitare/impostare alcune opzioni. In particolare sono state disattivate tutte le semplificazioni e le euristiche applicabili ad entrambi gli approcci ma non presenti in modo comune alle attuali implementazioni. L'algoritmo di saturazione adottato è stato *Otter*, poiché l'algoritmo predefinito *LRS* non offre garanzie di completezza e si basa sull'uso di limiti di tempo e memoria come criteri di selezione/semplificazione. È preferibile evitare questa metodologia poiché si desidera che gli algoritmi confrontati abbiano quantomeno lo stesso input. Anche per il preprocessing sono state disabilitate tutte le semplificazioni non comuni a entrambi gli algoritmi. In particolare l'opzione *-updr* (Unused Predicate Definition Removal) è stata disabilitata in quanto non utilizzata dall'algoritmo Binding. Come regola di semplificazione è stata disattivata l'opzione *-fs* (Forward Subsumption) che elimina clausole che sono sussunte da altre clausole durante la fase di *Forward simplification* (Una clausola D è sussunta da una clausola C se esiste una sostituzione σ tale che $C^\sigma \subseteq D$, clausole del genere vengono cercate dalla fs e rimosse perchè ridondanti). È stata disattivata anche l'opzione *-av* (AVATAR-Advanced Vampire Architecture for Theories and Resolution) che è un metodo SMT implementato in Vampire per lo splitting delle clausole. L'opzione *-av* è stata disattivata dato che non si vuole che Vampire utilizzi metodi SMT per la risoluzione dei problemi. Il comando utilizzato per l'esecuzione di Vampire è quindi il seguente:

```
[fontsize=\small]
    vampire --mode vampire -sa otter -t 10m -m 12000 -av off -updr off -fs off
```

Dove <*problem*> è il problema da risolvere. Come limiti di tempo e memoria sono stati impostati rispettivamente 10 minuti e 12GB di ram. L'algoritmo Binding è stato eseguito con i seguenti parametri:

```
vampire --mode 1b -t 10m -m 12000 <problem>
```

I seguenti risultati sono estrapolati dall'esecuzione del programma su un Macbook Pro 2018, 2.9 GHz 6-Core Intel Core i9, 16 GB 2400 MHz DDR4 sistema operativo macOS Sonoma 14.0. Gli esperimenti sono stati poi ripetuti su un computer Windows 11 con processore Intel Core i9-13900K e ram 32GB DDR5 sul sottosistema Windows for Linux (WSL) e si sono ottenuti tempi di esecuzione, come da aspettativa, più bassi ma assolutamente coerenti, mentre per memoria i valori sono rimasti esattamente gli stessi.

5.3 Ottimizzazioni

5.4 Conclusioni e Possibili Sviluppi futuri

Appendice A

Numerazione Dei problemi TPTP

Problema	Numero	Problema	Numero	Problema	Numero	Problema	Numero
KRS021+1	1	KRS022+1	2	KRS023+1	3	KRS024+1	4
KRS025+1	5	KRS027+1	6	KRS041+1	7	KRS053+1	8
KRS054+1	9	KRS055+1	10	KRS056+1	11	KRS058+1	12
KRS061+1	13	KRS062+1	14	KRS064+1	15	KRS066+1	16
KRS067+1	17	KRS091+1	18	KRS093+1	19	KRS094+1	20
KRS103+1	21	KRS136+1	22	KRS137+1	23	KRS164+1	24
KRS165+1	25	KRS166+1	26	KRS168+1	27	KRS169+1	28
KRS170+1	29	KRS171+1	30	LCL181+1	31	LCL230+1	32
LCL662+1.001	33	LCL663+1.001	34	LCL679+1.001	35	LCL680+1.001	36
LCL681+1.001	37	MED011+1	38	NLP263+1	39	PUZ068+2	40
PUZ069+2	41	PUZ079+2	42	PUZ080+2	43	PUZ138+2	44
SWB001+2	45	SWB003+2	46	SYN001+1	47	SYN040+1	48
SYN041+1	49	SYN044+1	50	SYN045+1	51	SYN046+1	52
SYN047+1	53	SYN054+1	54	SYN055+1	55	SYN057+1	56
SYN060+1	57	SYN061+1	58	SYN062+1	59	SYN355+1	60
SYN373+1	61	SYN378+1	62	SYN387+1	63	SYN388+1	64
SYN389+1	65	SYN390+1	66	SYN391+1	67	SYN392+1	68
SYN393+1.003	69	SYN394+1	70	SYN395+1	71	SYN396+1	72
SYN397+1	73	SYN400+1	74	SYN403+1	75	SYN404+1	76
SYN405+1	77	SYN406+1	78	SYN407+1	79	SYN408+1	80
SYN410+1	81	SYN411+1	82	SYN416+1	83	SYN724+1	84
SYN725+1	85	SYN915+1	86	SYN916+1	87	SYN920+1	88
SYN922+1	89	SYN923+1	90	SYN924+1	91	SYN926+1	92
SYN928+1	93	SYN932+1	94	SYN933+1	95	SYN942+1	96
SYN951+1	97	SYN952+1	98	SYN953+1	99	SYN955+1	100
SYN956+1	101	SYN958+1	102	SYN959+1	103	SYN960+1	104
SYN962+1	105	SYN963+1	106	SYN964+1	107	SYN972+1	108
SYN973+1	109	SYN974+1	110	SYN975+1	111	SYN977+1	112
SYN978+1	113	SYN981+1	114	KRS065+1	115	SYN048+1	116
SYN329+1	117	SYN336+1	118	SYN337+1	119	SYN338+1	120
SYN339+1	121	SYN340+1	122	SYN341+1	123	SYN342+1	124
SYN357+1	125	SYN368+1	126	SYN371+1	127	SYN383+1	128
SYN384+1	129	SYN385+1	130	SYN946+1	131	SYN947+1	132
SYN948+1	133	SYN968+1	134	SYO579+1	135	SYO580+1	136

Problema	Numero	Problema	Numero	Problema	Numero	Problema	Numero
BOO109+1	137	CSR025+1	138	CSR025+2	139	CSR025+3	140
CSR026+1	141	CSR026+2	142	CSR026+3	143	CSR028+1	144
CSR028+2	145	CSR028+3	146	CSR029+1	147	CSR029+2	148
CSR029+3	149	CSR031+1	150	CSR031+2	151	CSR031+3	152
CSR033+1	153	CSR033+2	154	CSR033+3	155	CSR036+1	156
CSR036+2	157	CSR036+3	158	CSR037+1	159	CSR037+2	160
CSR037+3	161	CSR039+1	162	CSR039+2	163	CSR039+3	164
CSR040+1	165	CSR040+2	166	CSR040+3	167	CSR041+1	168
CSR041+2	169	CSR041+3	170	CSR043+1	171	CSR043+2	172
CSR043+3	173	CSR045+1	174	CSR045+2	175	CSR045+3	176
CSR046+1	177	CSR046+2	178	CSR046+3	179	CSR047+1	180
CSR047+2	181	CSR047+3	182	CSR049+1	183	CSR049+2	184
CSR049+3	185	CSR050+1	186	CSR050+2	187	CSR050+3	188
CSR052+1	189	CSR052+2	190	CSR052+3	191	CSR053+1	192
CSR053+2	193	CSR053+3	194	CSR055+1	195	CSR055+2	196
CSR055+3	197	CSR059+1	198	CSR059+2	199	CSR059+3	200
CSR061+1	201	CSR061+2	202	CSR061+3	203	CSR063+1	204
CSR063+2	205	CSR063+3	206	CSR064+1	207	CSR064+2	208
CSR064+3	209	CSR065+1	210	CSR065+2	211	CSR065+3	212
CSR068+1	213	CSR068+2	214	CSR068+3	215	CSR069+1	216
CSR069+2	217	CSR069+3	218	CSR070+1	219	CSR070+2	220
CSR070+3	221	CSR071+1	222	CSR071+2	223	CSR071+3	224
CSR072+1	225	CSR072+2	226	CSR072+3	227	CSR074+1	228
CSR074+2	229	CSR074+3	230	GEO169+1	231	GEO171+1	232
GEO171+3	233	GEO207+1	234	GEO207+3	235	GEO211+1	236
GEO211+3	237	GEO216+1	238	GEO216+3	239	GEO222+1	240
GEO222+3	241	GEO223+1	242	GEO223+3	243	GRA013+1	244
GRA014+1	245	GRA015+1	246	GRA016+1	247	GRA017+1	248
GRA018+1	249	GRA019+1	250	GRA020+1	251	GRA021+1	252
GRA022+1	253	GRA023+1	254	GRA024+1	255	GRA025+1	256
GRA026+1	257	KRS139+1	258	LCL414+1	259	LCL876+1	260
MSC018+1	261	MSC019+1	262	PRD001+1	263	PRD002+1	264
PRD003+1	265	PUZ047+1	266	PUZ128+1	267	SWV011+1	268
SWV437+1	269	SWV438+1	270	SWV439+1	271	SWV440+1	272
SYN079+1	273	SYN362+1	274	SYN369+1	275	SYN430+1	276
SYN431+1	277	SYN432+1	278	SYN433+1	279	SYN434+1	280
SYN435+1	281	SYN436+1	282	SYN437+1	283	SYN438+1	284
SYN439+1	285	SYN440+1	286	SYN441+1	287	SYN442+1	288
SYN443+1	289	SYN444+1	290	SYN445+1	291	SYN446+1	292
SYN447+1	293	SYN448+1	294	SYN449+1	295	SYN450+1	296
SYN451+1	297	SYN452+1	298	SYN453+1	299	SYN454+1	300
SYN455+1	301	SYN456+1	302	SYN457+1	303	SYN458+1	304
SYN459+1	305	SYN460+1	306	SYN461+1	307	SYN462+1	308
SYN463+1	309	SYN464+1	310	SYN465+1	311	SYN466+1	312
SYN467+1	313	SYN468+1	314	SYN469+1	315	SYN470+1	316
SYN471+1	317	SYN472+1	318	SYN473+1	319	SYN474+1	320

Problema	Numero	Problema	Numero	Problema	Numero	Problema	Numero
SYN475+1	321	SYN476+1	322	SYN477+1	323	SYN478+1	324
SYN479+1	325	SYN480+1	326	SYN481+1	327	SYN482+1	328
SYN483+1	329	SYN484+1	330	SYN485+1	331	SYN486+1	332
SYN487+1	333	SYN488+1	334	SYN489+1	335	SYN490+1	336
SYN491+1	337	SYN492+1	338	SYN493+1	339	SYN494+1	340
SYN495+1	341	SYN496+1	342	SYN497+1	343	SYN498+1	344
SYN499+1	345	SYN500+1	346	SYN501+1	347	SYN502+1	348
SYN503+1	349	SYN504+1	350	SYN505+1	351	SYN506+1	352
SYN507+1	353	SYN508+1	354	SYN509+1	355	SYN510+1	356
SYN511+1	357	SYN512+1	358	SYN542+1	359	SYN543+1	360
SYN722+1	361	SYN726+1	362	SYN945+1	363	SYN986+1.000	364
SYN986+1.001	365	SYN986+1.002	366	SYN986+1.003	367	SYN986+1.004	368
SYN986+1.005	369	SYN986+1.006	370	SYO525+1.015	371	SYO525+1.018	372
SYO525+1.021	373	SYO525+1.024	374	SYO525+1.027	375	SYO525+1.030	376
ANA041-2	377	ANA042-2	378	COL101-2	379	COL103-2	380
COL113-2	381	COL116-2	382	GRA001-1	383	HWV003-3	384
KRS004-1	385	LAT260-2	386	LAT261-2	387	LAT264-2	388
LAT265-2	389	LAT267-2	390	LCL181-2	391	LCL230-2	392
LCL440-2	393	MSC007-1.008	394	NUM285-1	395	PUZ002-1	396
PUZ004-1	397	PUZ008-2	398	PUZ009-1	399	PUZ013-1	400
PUZ014-1	401	PUZ015-2.006	402	PUZ016-2.004	403	PUZ016-2.005	404
PUZ028-3	405	PUZ028-4	406	PUZ029-1	407	PUZ030-1	408
PUZ030-2	409	PUZ033-1	410	SET818-2	411	SET819-2	412
SET856-2	413	SWV309-2	414	SWV310-2	415	SWV312-2	416
SWV330-2	417	SWV333-2	418	SWV334-2	419	SWV336-2	420
SWV349-2	421	SWV350-2	422	SWV351-2	423	SYN001-1.005	424
SYN003-1.006	425	SYN004-1.007	426	SYN008-1	427	SYN010-1.005.005	428
SYN011-1	429	SYN028-1	430	SYN029-1	431	SYN030-1	432
SYN032-1	433	SYN040-1	434	SYN041-1	435	SYN044-1	436
SYN045-1	437	SYN046-1	438	SYN047-1	439	SYN048-1	440
SYN054-1	441	SYN060-1	442	SYN061-1	443	SYN062-1	444
SYN063-2	445	SYN064-1	446	SYN085-1.010	447	SYN086-1.003	448
SYN087-1.003	449	SYN089-1.002	450	SYN090-1.008	451	SYN091-1.003	452
SYN092-1.003	453	SYN093-1.002	454	SYN094-1.005	455	SYN095-1.002	456
SYN096-1.008	457	SYN097-1.002	458	SYN098-1.002	459	SYN099-1.003	460
SYN100-1.005	461	SYN302-1.003	462	SYN317-1	463	SYN329-1	464
SYN336-1	465	SYN337-1	466	SYN338-1	467	SYN339-1	468
SYN340-1	469	SYN341-1	470	SYN342-1	471	SYN724-1	472
SYN731-1	473	SYN915-1	474				

Tabella A.1: Numerazione dei problemi della libreria TPTP

Appendice B

Tabelle delle misurazioni di tempo e memoria

One Binding FOF

$N°$	**Vampire**	**1b naif**	**1b**
1	**0.734ms**	0.841ms	1.022ms
2	0.567ms	0.814ms	**0.521ms**
3	0.571ms	0.79ms	**0.5ms**
4	0.603ms	0.786ms	**0.591ms**
5	**0.531ms**	0.57ms	0.881ms
6	0.625ms	0.402ms	**0.354ms**
7	**TimeLimit**	**TimeLimit**	**1.786ms**
8	0.788ms	0.647ms	**0.358ms**
9	0.653ms	1.023ms	**0.384ms**
10	0.691ms	1.039ms	**0.366ms**
11	1.044ms	1.032ms	**0.383ms**
12	0.592ms	0.489ms	**0.323ms**
13	0.53ms	0.591ms	**0.409ms**
14	0.602ms	**0.395ms**	0.451ms
15	0.464ms	0.36ms	**0.332ms**
16	0.734ms	0.364ms	**0.335ms**
17	11.0ms	0.484ms	**0.384ms**
18	0.974ms	0.585ms	**0.359ms**
19	1.368ms	0.591ms	**0.514ms**
20	1.292ms	0.571ms	**0.396ms**
21	**TimeLimit**	5.353ms	**1.044ms**
22	1.221ms	0.488ms	**0.418ms**
23	342.0ms	6.848ms	**1.1ms**
24	337.0ms	6.975ms	**1.07ms**
25	**TimeLimit**	0.892ms	**0.647ms**
26	242.0ms	0.806ms	**0.584ms**
27	2.937ms	0.653ms	**0.535ms**
28	39.0ms	1.93ms	**1.071ms**
29	8.136ms	1.153ms	**0.845ms**
30	3.402ms	1.036ms	**0.684ms**
31	0.68ms	**0.002852ms** (g)	0.002953ms (g)
32	0.645ms	**0.002833ms** (g)	0.003199ms (g)
33	0.402ms	0.279ms	**0.251ms**
34	0.229ms	0.248ms	**0.224ms**
35	0.126ms	0ms	**0ms**
36	0.398ms	0.281ms	**0.249ms**
37	0.237ms	0.234ms	**0.217ms**
38	18000.0ms	74.0ms (g)	**74.0ms** (g)

$N°$	**Vampire**	**1b naif**	**1b**
39	34000.0ms	**136.0ms** (g)	137.0ms (g)
40	**TimeLimit**	1.887ms (g)	**1.878ms** (g)
41	**TimeLimit**	1.903ms (g)	**1.896ms** (g)
42	**TimeLimit**	1.501ms (g)	**1.491ms** (g)
43	**TimeLimit**	**1.54ms** (g)	1.554ms (g)
44	**TimeLimit**	6.307ms (g)	**6.29ms** (g)
45	0.614ms	0.003116ms (g)	**0.002933ms** (g)
46	0.443ms	0.347ms	**0.289ms**
47	0.44ms	0.002824ms (g)	**0.002807ms** (g)
48	0.604ms	**0.002817ms** (g)	0.002902ms (g)
49	0.465ms	0.002864ms (g)	**0.002794ms** (g)
50	1.455ms	0.026ms (g)	**0.026ms** (g)
51	3.638ms	**0.032ms** (g)	0.033ms (g)
52	0.608ms	**0.002861ms** (g)	0.00295ms (g)
53	**TimeLimit**	**0.042ms** (g)	0.044ms (g)
54	2.662ms	0.595ms	**0.441ms**
55	1.09ms	0.594ms	**0.313ms**
56	1.188ms	0.936ms	**0.48ms**
57	1.128ms	0.35ms	**0.31ms**
58	0.953ms	0.385ms	**0.311ms**
59	0.957ms	**0.385ms**	0.393ms
60	0.691ms	0.474ms	**0.294ms**
61	2.078ms	0.792ms	**0.575ms**
62	0.443ms	0.358ms	**0.298ms**
63	0.421ms	0.003014ms (g)	**0.002854ms** (g)
64	0.429ms	**0.002817ms** (g)	0.00294ms (g)
65	0.447ms	**0.002823ms** (g)	0.003353ms (g)
66	0.458ms	**0.002869ms** (g)	0.0031ms (g)
67	0.852ms	**0.023ms** (g)	0.024ms (g)
68	**TimeLimit**	0.038ms (g)	**0.038ms** (g)
69	**TimeLimit**	**0.07ms** (g)	0.071ms (g)
70	0.53ms	0.339ms	**0.282ms**
71	0.541ms	0.352ms	**0.307ms**
72	0.854ms	**0.42ms**	0.486ms
73	0.927ms	0.436ms	**0.396ms**
74	0.437ms	**0.002794ms** (g)	0.002931ms (g)
75	0.584ms	0.286ms	**0.268ms**
76	0.42ms	**0.325ms**	0.334ms

$N°$	**Vampire**	**1b naif**	**1b**
77	0.625ms	0.355ms	**0.293ms**
78	0.651ms	0.354ms	**0.294ms**
79	0.767ms	0.35ms	**0.291ms**
80	0.424ms	0.322ms	**0.284ms**
81	0.437ms	0.338ms	**0.321ms**
82	0.892ms	0.489ms	**0.407ms**
83	0.469ms	**0.002802ms** (g)	0.002873ms (g)
84	1.155ms	**0.432ms**	0.507ms
85	0.327ms	0.356ms	**0.312ms**
86	0.119ms	0ms	**0ms**
87	0.086ms	0ms	**0ms**
88	3433.0ms	0.862ms	**0.633ms**
89	1.802ms	0.828ms	**0.515ms**
90	0.55ms	0.413ms	**0.358ms**
91	2.666ms	0.714ms	**0.597ms**
92	0.508ms	0.487ms	**0.369ms**
93	0.437ms	**0.302ms**	0.33ms
94	0.421ms	**0.002708ms** (g)	0.003116ms (g)
95	0.43ms	**0.002837ms** (g)	0.002914ms (g)
96	0.757ms	0.347ms	**0.308ms**
97	1.196ms	1.754ms	**0.393ms**
98	0.42ms	0.348ms	**0.322ms**
99	0.567ms	0.352ms	**0.309ms**
100	0.563ms	0.338ms	**0.281ms**
101	0.535ms	0.362ms	**0.288ms**
102	0.556ms	0.354ms	**0.291ms**
103	0.43ms	0.321ms	**0.266ms**
104	0.866ms	0.481ms	**0.393ms**
105	0.495ms	**0.41ms**	0.43ms
106	0.807ms	0.459ms	**0.365ms**
107	0.377ms	0.354ms	**0.274ms**
108	0.423ms	0.399ms	**0.397ms**
109	0.412ms	0.003173ms (g)	**0.003125ms** (g)
110	0.395ms	0.317ms	**0.276ms**
111	0.429ms	0.333ms	**0.292ms**
112	0.711ms	**0.002826ms** (g)	0.002872ms (g)
113	0.624ms	0.002856ms (g)	**0.002846ms** (g)
114	0.788ms	0.348ms	**0.297ms**

Tabella B.1: Tempi di esecuzione in millisecondi dei problemi One Binding (FOF) di Vampire, 1b naif e 1b

$N°$	**Vampire**	**1b naif**	**1b**
1	381Kb	379Kb	**379Kb**
2	381Kb	379Kb	**379Kb**
3	381Kb	379Kb	**379Kb**
4	381Kb	379Kb	**379Kb**
5	380Kb	379Kb	**379Kb**
6	380Kb	379Kb	**378Kb**
7	9925815Kb	874859Kb	**432Kb**
8	382Kb	380Kb	**379Kb**
9	381Kb	381Kb	**380Kb**
10	381Kb	381Kb	**380Kb**
11	381Kb	380Kb	**379Kb**
12	380Kb	379Kb	**378Kb**
13	380Kb	379Kb	**379Kb**
14	380Kb	379Kb	**378Kb**
15	379Kb	379Kb	**378Kb**
16	381Kb	379Kb	**379Kb**
17	428Kb	381Kb	**380Kb**
18	384Kb	385Kb	**383Kb**
19	384Kb	385Kb	**383Kb**
20	384Kb	385Kb	**382Kb**
21	9210243Kb	495Kb	**432Kb**
22	384Kb	381Kb	**381Kb**
23	3573Kb	399Kb	**382Kb**
24	3573Kb	399Kb	**382Kb**
25	9486812Kb	383Kb	**382Kb**
26	4038Kb	383Kb	**382Kb**
27	397Kb	382Kb	**381Kb**
28	603Kb	383Kb	**382Kb**
29	424Kb	383Kb	**383Kb**
30	399Kb	383Kb	**383Kb**
31	380Kb	374Kb (g)	**374Kb** (g)
32	380Kb	375Kb (g)	**375Kb** (g)
33	379Kb	379Kb	**379Kb**
34	**376Kb**	379Kb	379Kb
35	373Kb	373Kb	**373Kb**
36	379Kb	379Kb	**379Kb**
37	**376Kb**	379Kb	379Kb
38	**366755Kb**	373599Kb (g)	373599Kb (g)
39	**585890Kb**	600171Kb (g)	600171Kb (g)
40	11997208Kb	9143Kb (g)	**9143Kb** (g)
41	12275854Kb	9143Kb (g)	**9143Kb** (g)
42	11436239Kb	9144Kb (g)	**9144Kb** (g)
43	11627330Kb	9144Kb (g)	**9144Kb** (g)
44	11350340Kb	9143Kb (g)	**9143Kb** (g)
45	380Kb	375Kb (g)	**375Kb** (g)
46	379Kb	379Kb	**378Kb**
47	378Kb	374Kb (g)	**374Kb** (g)
48	380Kb	374Kb (g)	**374Kb** (g)
49	378Kb	374Kb (g)	**374Kb** (g)
50	383Kb	375Kb (g)	**375Kb** (g)
51	394Kb	375Kb (g)	**375Kb** (g)
52	380Kb	374Kb (g)	**374Kb** (g)
53	7228190Kb	376Kb (g)	**376Kb** (g)
54	387Kb	382Kb	**381Kb**
55	383Kb	383Kb	**381Kb**
56	384Kb	382Kb	**381Kb**
57	383Kb	380Kb	**380Kb**
58	382Kb	381Kb	**380Kb**
59	383Kb	381Kb	**380Kb**
60	381Kb	380Kb	**380Kb**
61	390Kb	381Kb	**380Kb**
62	380Kb	380Kb	**379Kb**
63	378Kb	374Kb (g)	**374Kb** (g)
64	378Kb	374Kb (g)	**374Kb** (g)
65	378Kb	374Kb (g)	**374Kb** (g)
66	378Kb	374Kb (g)	**374Kb** (g)
67	380Kb	374Kb (g)	**374Kb** (g)
68	6776295Kb	376Kb (g)	**376Kb** (g)
69	10184024Kb	375Kb (g)	**375Kb** (g)
70	381Kb	380Kb	**379Kb**
71	381Kb	380Kb	**379Kb**
72	382Kb	380Kb	**380Kb**
73	382Kb	380Kb	**380Kb**
74	378Kb	374Kb (g)	**374Kb** (g)
75	381Kb	379Kb	**379Kb**
76	378Kb	378Kb	**378Kb**
77	381Kb	380Kb	**379Kb**
78	381Kb	380Kb	**380Kb**
79	382Kb	380Kb	**380Kb**
80	379Kb	380Kb	**379Kb**
81	378Kb	379Kb	**378Kb**
82	384Kb	381Kb	**380Kb**
83	378Kb	374Kb (g)	**374Kb** (g)
84	385Kb	380Kb	**380Kb**
85	379Kb	380Kb	**379Kb**
86	374Kb	373Kb	**373Kb**
87	373Kb	373Kb	**373Kb**
88	38045Kb	382Kb	**381Kb**
89	388Kb	381Kb	**380Kb**
90	381Kb	380Kb	**379Kb**
91	393Kb	381Kb	**380Kb**
92	381Kb	380Kb	**379Kb**
93	378Kb	378Kb	**378Kb**
94	378Kb	374Kb (g)	**374Kb** (g)
95	378Kb	374Kb (g)	**374Kb** (g)
96	382Kb	380Kb	**380Kb**
97	383Kb	380Kb	**379Kb**
98	378Kb	378Kb	**378Kb**
99	381Kb	380Kb	**379Kb**
100	381Kb	380Kb	**379Kb**
101	381Kb	380Kb	**379Kb**
102	381Kb	380Kb	**379Kb**
103	379Kb	380Kb	**379Kb**
104	382Kb	381Kb	**380Kb**
105	381Kb	380Kb	**380Kb**
106	382Kb	380Kb	**380Kb**
107	379Kb	379Kb	**379Kb**
108	380Kb	380Kb	**380Kb**
109	378Kb	374Kb (g)	**374Kb** (g)
110	379Kb	379Kb	**379Kb**
111	379Kb	380Kb	**379Kb**
112	380Kb	374Kb (g)	**374Kb** (g)
113	380Kb	374Kb (g)	**374Kb** (g)
114	381Kb	380Kb	**380Kb**

Tabella B.2: Memoria di esecuzione in kilobyte dei problemi One Binding (FOF) di Vampire, 1b naif e 1b

One Binding CNF

$N°$	**Vampire**	**1b naif**	**1b**
377	0.445ms	0.385ms	**0.32ms**
378	**0.469ms**	0.572ms	0.581ms
379	0.492ms	0.512ms	**0.399ms**
380	**0.439ms**	0.506ms	0.61ms
381	0.449ms	0.435ms	**0.419ms**
382	0.476ms	**0.42ms**	0.446ms
383	**TimeLimit**	**0.044ms** (g)	0.045ms (g)
384	**TimeLimit**	**0.213ms** (g)	0.223ms (g)
385	0.679ms	0.393ms	**0.331ms**
386	0.435ms	0.003143ms (g)	**0.003072ms** (g)
387	0.416ms	**0.00294ms** (g)	0.003088ms (g)
388	0.421ms	**0.002989ms** (g)	0.003266ms (g)
389	0.415ms	**0.0028ms** (g)	0.003191ms (g)
390	0.448ms	**0.002779ms** (g)	0.003118ms (g)
391	0.644ms	**0.002972ms** (g)	0.003437ms (g)
392	0.675ms	**0.002836ms** (g)	0.003109ms (g)
393	0.478ms	0.777ms	**0.407ms**
394	531000.0ms	354.0ms (g)	**345.0ms** (g)
395	**TimeLimit**	0.138ms (g)	**0.136ms** (g)
396	1.688ms	0.708ms	**0.388ms**
397	1.622ms	**0.002926ms** (g)	0.00309ms (g)
398	555.0ms	**0.003052ms** (g)	0.00325ms (g)
399	2.95ms	**0.002962ms** (g)	0.00419ms (g)
400	3.747ms	**0.002961ms** (g)	0.003091ms (g)
401	203000.0ms	0.04ms (g)	**0.04ms** (g)
402	570000.0ms	**1.853ms** (g)	1.863ms (g)
403	**TimeLimit**	0.077ms (g)	**0.074ms** (g)
404	**TimeLimit**	0.485ms (g)	**0.327ms** (g)
405	319.0ms	**0.33ms** (g)	0.355ms (g)
406	233.0ms	0.202ms (g)	**0.202ms** (g)
407	3.047ms	0.96ms	**0.399ms**
408	**TimeLimit**	**TimeLimit**	**60.0ms**
409	**TimeLimit**	0.161ms (g)	**0.159ms** (g)
410	1.981ms	0.00318ms (g)	**0.003162ms** (g)
411	0.483ms	**0.459ms**	0.565ms
412	0.527ms	0.448ms	**0.417ms**
413	0.399ms	0.003159ms (g)	**0.003009ms** (g)
414	0.619ms	0.469ms	**0.4ms**
415	0.666ms	0.466ms	**0.407ms**
416	0.706ms	0.491ms	**0.418ms**
417	0.409ms	**0.388ms**	0.461ms
418	0.436ms	0.371ms	**0.301ms**
419	0.729ms	0.363ms	**0.3ms**
420	0.424ms	0.512ms	**0.416ms**
421	0.45ms	0.358ms	**0.311ms**
422	0.421ms	0.38ms	**0.321ms**
423	0.425ms	0.382ms	**0.303ms**
424	**TimeLimit**	**0.161ms** (g)	0.177ms (g)
425	5.123ms	**0.002856ms** (g)	0.00348ms (g)
426	2.272ms	**0.002999ms** (g)	0.003285ms (g)
427	0.841ms	**0.00292ms** (g)	0.003395ms (g)
428	7.925ms	**0.00318ms** (g)	0.003195ms (g)
429	1.924ms	0.03ms (g)	**0.027ms** (g)
430	0.947ms	**0.003179ms** (g)	0.003479ms (g)
431	0.87ms	0.038ms (g)	**0.022ms** (g)
432	1.972ms	0.032ms (g)	**0.031ms** (g)
433	1.137ms	**0.041ms** (g)	0.042ms (g)
434	0.574ms	**0.002785ms** (g)	0.00324ms (g)
435	0.546ms	**0.002799ms** (g)	0.003133ms (g)
436	1.438ms	**0.027ms** (g)	0.028ms (g)
437	0.674ms	0.003327ms (g)	**0.003076ms** (g)
438	0.512ms	0.003254ms (g)	**0.003158ms** (g)
439	0.865ms	**0.003188ms** (g)	0.003424ms (g)
440	0.427ms	0.38ms	**0.368ms**
441	3.042ms	0.698ms	**0.457ms**
442	1.248ms	0.502ms	**0.339ms**
443	0.82ms	0.464ms	**0.329ms**
444	0.93ms	0.489ms	**0.351ms**
445	0.504ms	**0.003155ms** (g)	0.003182ms (g)
446	0.488ms	0.492ms	**0.393ms**
447	1.4ms	**0.002945ms** (g)	0.003119ms (g)
448	0.777ms	0.041ms (g)	**0.039ms** (g)
449	1.073ms	0.046ms (g)	**0.046ms** (g)
450	1.052ms	**0.002861ms** (g)	0.003092ms (g)
451	**TimeLimit**	**0.003086ms** (g)	0.003448ms (g)
452	**TimeLimit**	0.066ms (g)	**0.065ms** (g)
453	2.606ms	0.086ms (g)	**0.086ms** (g)
454	2.222ms	0.039ms (g)	**0.034ms** (g)
455	**TimeLimit**	0.224ms (g)	**0.212ms** (g)
456	0.932ms	0.816ms	**0.394ms**
457	**TimeLimit**	6.102ms	**0.847ms**
458	2.723ms	**0.05ms** (g)	0.054ms (g)
459	7.13ms	0.102ms (g)	**0.1ms** (g)
460	11.0ms	3.576ms	**0.718ms**
461	**TimeLimit**	8.69ms	**0.996ms**
462	2.427ms	0.074ms (g)	**0.073ms** (g)
463	0.684ms	0.67ms	**0.482ms**
464	0.455ms	0.498ms	**0.425ms**
465	**0.55ms**	0.849ms	0.835ms
466	**0.469ms**	0.606ms	0.624ms
467	**0.48ms**	0.524ms	0.555ms
468	0.461ms	**0.453ms**	0.543ms
469	**0.456ms**	0.519ms	0.52ms
470	0.569ms	**0.449ms**	0.487ms
471	0.392ms	**0.306ms**	0.344ms
472	1.552ms	1.346ms	**0.482ms**
473	0.415ms	0.471ms	**0.409ms**
474	0.113ms	0ms	**0ms**

Tabella B.3: Tempi di esecuzione in millisecondi dei problemi One Binding (CNF) di Vampire, 1b naif e 1b

$N°$	Vampire	1b naif	1b
377	378Kb	379Kb	**378Kb**
378	**378Kb**	381Kb	381Kb
379	**379Kb**	381Kb	380Kb
380	**379Kb**	381Kb	380Kb
381	**378Kb**	379Kb	379Kb
382	**379Kb**	380Kb	380Kb
383	6738678Kb	377Kb (g)	**377Kb** (g)
384	10251835Kb	452Kb (g)	**452Kb** (g)
385	379Kb	379Kb	**378Kb**
386	378Kb	375Kb (g)	**375Kb** (g)
387	378Kb	375Kb (g)	**375Kb** (g)
388	378Kb	375Kb (g)	**375Kb** (g)
389	378Kb	375Kb (g)	**375Kb** (g)
390	378Kb	375Kb (g)	**375Kb** (g)
391	379Kb	375Kb (g)	**375Kb** (g)
392	379Kb	375Kb (g)	**375Kb** (g)
393	**378Kb**	381Kb	379Kb
394	**MemoryLimit**	617Kb (g)	**617Kb** (g)
395	9341276Kb	451Kb (g)	**451Kb** (g)
396	**383Kb**	388Kb	385Kb
397	382Kb	377Kb (g)	**377Kb** (g)
398	2302Kb	382Kb (g)	**382Kb** (g)
399	388Kb	378Kb (g)	**378Kb** (g)
400	392Kb	379Kb (g)	**379Kb** (g)
401	1904861Kb	379Kb (g)	**379Kb** (g)
402	**MemoryLimit**	550Kb (g)	**550Kb** (g)
403	11126729Kb	453Kb (g)	**453Kb** (g)
404	11980325Kb	467Kb (g)	**467Kb** (g)
405	4305Kb	1080Kb (g)	**1080Kb** (g)
406	2257Kb	738Kb (g)	**738Kb** (g)
407	388Kb	390Kb	**386Kb**
408	10275650Kb	668220Kb	**480Kb**
409	3885891Kb	452Kb (g)	**452Kb** (g)
410	384Kb	377Kb (g)	**377Kb** (g)
411	**378Kb**	379Kb	379Kb
412	**378Kb**	379Kb	379Kb
413	378Kb	375Kb (g)	**375Kb** (g)
414	382Kb	381Kb	**380Kb**
415	382Kb	381Kb	**380Kb**
416	382Kb	381Kb	**380Kb**
417	379Kb	379Kb	**379Kb**
418	379Kb	379Kb	**379Kb**
419	379Kb	379Kb	**379Kb**
420	**380Kb**	382Kb	381Kb
421	379Kb	379Kb	**378Kb**
422	379Kb	379Kb	**378Kb**
423	379Kb	379Kb	**378Kb**
424	10496665Kb	445Kb (g)	**445Kb** (g)
425	394Kb	380Kb (g)	**380Kb** (g)
426	385Kb	378Kb (g)	**378Kb** (g)
427	381Kb	376Kb (g)	**376Kb** (g)
428	422Kb	400Kb (g)	**400Kb** (g)
429	383Kb	376Kb (g)	**376Kb** (g)
430	380Kb	375Kb (g)	**375Kb** (g)
431	380Kb	375Kb (g)	**375Kb** (g)
432	384Kb	376Kb (g)	**376Kb** (g)
433	381Kb	376Kb (g)	**376Kb** (g)
434	379Kb	375Kb (g)	**375Kb** (g)
435	378Kb	375Kb (g)	**375Kb** (g)
436	383Kb	375Kb (g)	**375Kb** (g)
437	379Kb	375Kb (g)	**375Kb** (g)
438	379Kb	375Kb (g)	**375Kb** (g)
439	380Kb	375Kb (g)	**375Kb** (g)
440	**378Kb**	379Kb	379Kb
441	387Kb	381Kb	**379Kb**
442	381Kb	381Kb	**380Kb**
443	380Kb	381Kb	**379Kb**
444	381Kb	381Kb	**379Kb**
445	379Kb	375Kb (g)	**375Kb** (g)
446	**378Kb**	379Kb	379Kb
447	387Kb	377Kb (g)	**377Kb** (g)
448	379Kb	379Kb (g)	**379Kb** (g)
449	380Kb	380Kb (g)	**380Kb** (g)
450	381Kb	377Kb (g)	**377Kb** (g)
451	5481777Kb	457Kb (g)	**457Kb** (g)
452	3595452Kb	400Kb (g)	**400Kb** (g)
453	**395Kb**	403Kb (g)	403Kb (g)
454	388Kb	381Kb (g)	**381Kb** (g)
455	4357063Kb	466Kb (g)	**466Kb** (g)
456	**381Kb**	385Kb	382Kb
457	5123079Kb	499Kb	**480Kb**
458	**395Kb**	454Kb (g)	454Kb (g)
459	**419Kb**	463Kb (g)	463Kb (g)
460	423Kb	480Kb	**418Kb**
461	5400145Kb	529Kb	**506Kb**
462	**387Kb**	455Kb (g)	455Kb (g)
463	380Kb	379Kb	**379Kb**
464	**378Kb**	380Kb	379Kb
465	**378Kb**	381Kb	381Kb
466	**376Kb**	380Kb	380Kb
467	**378Kb**	379Kb	379Kb
468	**378Kb**	379Kb	379Kb
469	**379Kb**	381Kb	381Kb
470	**378Kb**	379Kb	379Kb
471	378Kb	378Kb	**378Kb**
472	382Kb	379Kb	**378Kb**
473	**378Kb**	379Kb	379Kb
474	374Kb	373Kb	**373Kb**

Tabella B.4: Memoria di esecuzione in kilobyte dei problemi One Binding (CNF) di Vampire, 1b naif e 1b

Conjunctive Binding

$N°$	Vampire	1b naif	1b
115	0.565ms	0.357ms	**0.303ms**
116	0.398ms	0.348ms	**0.324ms**
117	**0.428ms**	0.494ms	0.457ms
118	0.616ms	**0.489ms**	0.516ms
119	**0.552ms**	0.64ms	0.655ms
120	0.478ms	0.435ms	**0.371ms**
121	**0.453ms**	0.46ms	0.47ms
122	0.507ms	**0.496ms**	0.554ms
123	**0.471ms**	0.493ms	0.482ms
124	0.37ms	**0.347ms**	0.358ms
125	0.429ms	0.319ms	**0.283ms**
126	0.385ms	0.352ms	**0.333ms**
127	**0.463ms**	0.592ms	0.583ms
128	0.436ms	0.415ms	**0.412ms**
129	**0.452ms**	0.459ms	0.471ms
130	0.433ms	0.391ms	**0.357ms**
131	0.534ms	0.548ms	**0.465ms**
132	0.496ms	0.399ms	**0.399ms**
133	0.502ms	0.39ms	**0.357ms**
134	0.395ms	0.35ms	**0.326ms**
135	**TimeLimit**	**TimeLimit**	**120.0ms**
136	**439.0ms**	**TimeLimit**	5810.0ms

Tabella B.5: Tempi di esecuzione in millisecondi dei problemi Conjunctive Binding di Vampire, 1b naif e 1b

$N°$	**Vampire**	**1b naif**	**1b**
115	381Kb	381Kb	**380Kb**
116	380Kb	380Kb	**380Kb**
117	380Kb	381Kb	**380Kb**
118	381Kb	381Kb	**381Kb**
119	**378Kb**	382Kb	382Kb
120	380Kb	380Kb	**380Kb**
121	**380Kb**	381Kb	381Kb
122	**381Kb**	383Kb	383Kb
123	**380Kb**	381Kb	381Kb
124	380Kb	380Kb	**380Kb**
125	379Kb	379Kb	**379Kb**
126	380Kb	380Kb	**380Kb**
127	**380Kb**	381Kb	381Kb
128	**380Kb**	381Kb	381Kb
129	**380Kb**	381Kb	381Kb
130	380Kb	381Kb	**380Kb**
131	381Kb	381Kb	**381Kb**
132	**380Kb**	381Kb	381Kb
133	380Kb	380Kb	**380Kb**
134	380Kb	380Kb	**380Kb**
135	10478401Kb	824716Kb	**984Kb**
136	**3618Kb**	997382Kb	18107Kb

Tabella B.6: Memoria di esecuzione in kilobyte dei problemi Conjunctive Binding di Vampire, 1b naif e 1b

Disjunctive Binding

$N°$	**Vampire**	**1b naif**	**1b**
137	0.417ms	0.6ms	**0.257ms**
138	5.311ms	1.522ms	**0.123ms** (g)
139	81.0ms	742.0ms	**21.0ms** (g)
140	519.0ms	36000.0ms	**277.0ms** (g)
141	5.224ms	2.811ms	**0.367ms** (g)
142	105.0ms	696.0ms	**19.0ms** (g)
143	813.0ms	36000.0ms	**314.0ms** (g)
144	3.583ms	1.894ms	**0.125ms** (g)
145	75.0ms	700.0ms	**18.0ms** (g)
146	501.0ms	36000.0ms	**285.0ms** (g)
147	5.94ms	3.474ms	**0.463ms** (g)
148	100.0ms	697.0ms	**21.0ms** (g)
149	792.0ms	35000.0ms	**285.0ms** (g)
150	5.562ms	2.161ms	**0.137ms** (g)
151	102.0ms	696.0ms	**19.0ms** (g)
152	803.0ms	36000.0ms	**283.0ms** (g)
153	6.032ms	4.588ms	**0.644ms** (g)
154	101.0ms	695.0ms	**21.0ms** (g)
155	805.0ms	36000.0ms	**283.0ms** (g)
156	16.0ms	16.0ms	**0.416ms** (g)
157	104.0ms	703.0ms	**19.0ms** (g)
158	806.0ms	35000.0ms	**281.0ms** (g)
159	6.729ms	3.084ms	**0.394ms** (g)
160	101.0ms	696.0ms	**19.0ms** (g)
161	799.0ms	36000.0ms	**286.0ms** (g)
162	15.0ms	14.0ms	**0.366ms** (g)
163	101.0ms	732.0ms	**19.0ms** (g)
164	806.0ms	38000.0ms	**287.0ms** (g)
165	12.0ms	12.0ms	**0.431ms** (g)
166	103.0ms	737.0ms	**20.0ms** (g)
167	803.0ms	38000.0ms	**289.0ms** (g)
168	6.53ms	4.305ms	**0.175ms** (g)
169	102.0ms	734.0ms	**19.0ms** (g)
170	806.0ms	38000.0ms	**281.0ms** (g)
171	4.902ms	2.36ms	**0.142ms** (g)
172	74.0ms	735.0ms	**18.0ms** (g)
173	495.0ms	37000.0ms	**280.0ms** (g)
174	8.484ms	3.983ms	**0.197ms** (g)

$N°$	**Vampire**	**1b naif**	**1b**
175	103.0ms	729.0ms	**19.0ms** (g)
176	798.0ms	38000.0ms	**277.0ms** (g)
177	4.918ms	2.289ms	**0.149ms** (g)
178	76.0ms	728.0ms	**18.0ms** (g)
179	497.0ms	37000.0ms	**281.0ms** (g)
180	3.657ms	0.688ms	**0.096ms** (g)
181	103.0ms	726.0ms	**19.0ms** (g)
182	807.0ms	37000.0ms	**286.0ms** (g)
183	16.0ms	19.0ms	**0.386ms** (g)
184	102.0ms	740.0ms	**19.0ms** (g)
185	801.0ms	37000.0ms	**286.0ms** (g)
186	7.059ms	4.382ms	**0.342ms** (g)
187	101.0ms	734.0ms	**19.0ms** (g)
188	802.0ms	38000.0ms	**287.0ms** (g)
189	9.249ms	4.908ms	**0.234ms** (g)
190	103.0ms	725.0ms	**19.0ms** (g)
191	800.0ms	37000.0ms	**287.0ms** (g)
192	3.874ms	0.682ms	**0.094ms** (g)
193	102.0ms	734.0ms	**19.0ms** (g)
194	803.0ms	37000.0ms	**285.0ms** (g)
195	5.762ms	2.162ms	**0.142ms** (g)
196	103.0ms	731.0ms	**20.0ms** (g)
197	800.0ms	37000.0ms	**281.0ms** (g)
198	4.209ms	1.585ms	**0.121ms** (g)
199	102.0ms	738.0ms	**19.0ms** (g)
200	799.0ms	37000.0ms	**287.0ms** (g)
201	11.0ms	7.433ms	**0.336ms** (g)
202	104.0ms	737.0ms	**19.0ms** (g)
203	801.0ms	37000.0ms	**286.0ms** (g)
204	11.0ms	5.461ms	**0.258ms** (g)
205	101.0ms	730.0ms	**19.0ms** (g)
206	798.0ms	38000.0ms	**280.0ms** (g)
207	8.537ms	2.358ms	**0.162ms** (g)
208	102.0ms	732.0ms	**19.0ms** (g)
209	803.0ms	37000.0ms	**280.0ms** (g)
210	5.208ms	2.82ms	**0.351ms** (g)
211	103.0ms	735.0ms	**20.0ms** (g)
212	802.0ms	37000.0ms	**286.0ms** (g)

$N°$	**Vampire**	**1b naif**	**1b**
213	5.018ms	2.466ms	**0.189ms** (g)
214	102.0ms	734.0ms	**19.0ms** (g)
215	803.0ms	37000.0ms	**281.0ms** (g)
216	3.505ms	0.708ms	**0.15ms** (g)
217	102.0ms	726.0ms	**19.0ms** (g)
218	804.0ms	37000.0ms	**286.0ms** (g)
219	7.006ms	4.541ms	**0.373ms** (g)
220	103.0ms	728.0ms	**19.0ms** (g)
221	801.0ms	37000.0ms	**289.0ms** (g)
222	6.82ms	3.756ms	**0.404ms** (g)
223	102.0ms	738.0ms	**19.0ms** (g)
224	796.0ms	37000.0ms	**286.0ms** (g)
225	5.02ms	2.306ms	**0.148ms** (g)
226	75.0ms	725.0ms	**18.0ms** (g)
227	496.0ms	38000.0ms	**281.0ms** (g)
228	5.732ms	3.541ms	**0.478ms** (g)
229	101.0ms	743.0ms	**20.0ms** (g)
230	800.0ms	37000.0ms	**285.0ms** (g)
231	5.143ms	4.775ms	**1.955ms** (g)
232	2.816ms	0.347ms	**0.256ms**
233	4.828ms	0.453ms	**0.285ms**
234	2.567ms	0.361ms	**0.285ms**
235	4.742ms	0.445ms	**0.275ms**
236	2.96ms	0.411ms	**0.257ms**
237	4.751ms	0.445ms	**0.27ms**
238	2.693ms	0.454ms	**0.267ms**
239	4.699ms	0.454ms	**0.277ms**
240	3.034ms	0.383ms	**0.258ms**
241	5.036ms	0.44ms	**0.267ms**
242	3.099ms	0.389ms	**0.261ms**
243	5.393ms	0.444ms	**0.276ms**
244	1.062ms	0.527ms	**0.066ms** (g)
245	0.914ms	0.551ms	**0.105ms** (g)
246	1.389ms	0.863ms	**0.1ms** (g)
247	1.452ms	1.124ms	**0.107ms** (g)
248	1.403ms	1.129ms	**0.115ms** (g)
249	1.385ms	1.207ms	**0.156ms** (g)
250	1.479ms	1.205ms	**0.165ms** (g)

N°	**Vampire**	**1b naif**	**1b**
251	1.663ms	1.343ms	**0.11ms** (g)
252	1.666ms	0.915ms	**0.108ms** (g)
253	1.85ms	0.969ms	**0.116ms** (g)
254	1.872ms	1.146ms	**0.147ms** (g)
255	2.052ms	0.855ms	**0.099ms** (g)
256	2.174ms	1.421ms	**0.141ms** (g)
257	2.077ms	1.404ms	**0.192ms** (g)
258	0.996ms	0.511ms	**0.081ms** (g)
259	0.331ms	**0.29ms**	0.405ms
260	0.36ms	0.305ms	**0.25ms**
261	16.0ms	29.0ms	**5.453ms** (g)
262	22.0ms	30.0ms	**5.385ms** (g)
263	158.0ms	1067.0ms	**7.036ms** (g)
264	154.0ms	995.0ms	**6.764ms** (g)
265	155.0ms	1001.0ms	**6.764ms** (g)
266	2.162ms	0.343ms	**0.061ms** (g)
267	0.396ms	0.461ms	**0.062ms** (g)
268	0.345ms	1.351ms	**0.141ms** (g)
269	6.83ms	11.0ms	**0.353ms** (g)
270	6.854ms	11.0ms	**0.352ms** (g)
271	6.781ms	11.0ms	**0.337ms** (g)
272	6.934ms	11.0ms	**0.337ms** (g)
273	0.315ms	0.373ms	**0.051ms** (g)
274	0.309ms	0.418ms	**0.28ms**
275	0.279ms	0.306ms	**0.278ms**
276	10.0ms	5.991ms	**0.745ms** (g)
277	11.0ms	7.504ms	**0.867ms** (g)
278	13.0ms	8.969ms	**1.018ms** (g)
279	13.0ms	7.83ms	**0.881ms** (g)
280	32.0ms	25.0ms	**1.586ms** (g)
281	31.0ms	22.0ms	**1.618ms** (g)
282	21.0ms	11.0ms	**1.022ms** (g)
283	36.0ms	25.0ms	**1.82ms** (g)
284	25.0ms	13.0ms	**1.25ms** (g)
285	37.0ms	25.0ms	**1.894ms** (g)
286	50.0ms	40.0ms	**2.196ms** (g)
287	55.0ms	45.0ms	**2.397ms** (g)
288	30.0ms	15.0ms	**1.336ms** (g)

N°	**Vampire**	**1b naif**	**1b**
289	28.0ms	15.0ms	**1.378ms** (g)
290	30.0ms	14.0ms	**1.403ms** (g)
291	30.0ms	15.0ms	**1.229ms** (g)
292	30.0ms	15.0ms	**1.398ms** (g)
293	48.0ms	38.0ms	**2.33ms** (g)
294	29.0ms	15.0ms	**1.415ms** (g)
295	29.0ms	16.0ms	**1.507ms** (g)
296	29.0ms	15.0ms	**1.383ms** (g)
297	29.0ms	15.0ms	**1.357ms** (g)
298	29.0ms	16.0ms	**1.284ms** (g)
299	50.0ms	40.0ms	**2.196ms** (g)
300	28.0ms	15.0ms	**1.287ms** (g)
301	29.0ms	15.0ms	**2.033ms** (g)
302	29.0ms	15.0ms	**1.376ms** (g)
303	55.0ms	43.0ms	**2.214ms** (g)
304	29.0ms	16.0ms	**1.305ms** (g)
305	29.0ms	13.0ms	**1.29ms** (g)
306	49.0ms	35.0ms	**1.967ms** (g)
307	29.0ms	14.0ms	**1.278ms** (g)
308	29.0ms	15.0ms	**1.321ms** (g)
309	29.0ms	16.0ms	**1.38ms** (g)
310	51.0ms	40.0ms	**2.227ms** (g)
311	33.0ms	17.0ms	**1.53ms** (g)
312	33.0ms	17.0ms	**1.414ms** (g)
313	33.0ms	17.0ms	**1.378ms** (g)
314	32.0ms	17.0ms	**1.377ms** (g)
315	32.0ms	15.0ms	**1.346ms** (g)
316	33.0ms	18.0ms	**1.519ms** (g)
317	36.0ms	18.0ms	**1.458ms** (g)
318	35.0ms	18.0ms	**1.602ms** (g)
319	34.0ms	18.0ms	**1.528ms** (g)
320	36.0ms	18.0ms	**1.567ms** (g)
321	35.0ms	18.0ms	**1.57ms** (g)
322	35.0ms	18.0ms	**1.542ms** (g)
323	33.0ms	16.0ms	**1.443ms** (g)
324	35.0ms	17.0ms	**1.49ms** (g)
325	37.0ms	18.0ms	**1.569ms** (g)
326	39.0ms	19.0ms	**1.552ms** (g)

N°	**Vampire**	**1b naif**	**1b**
327	35.0ms	18.0ms	**1.583ms** (g)
328	38.0ms	18.0ms	**1.566ms** (g)
329	34.0ms	17.0ms	**1.585ms** (g)
330	37.0ms	18.0ms	**1.48ms** (g)
331	36.0ms	18.0ms	**1.48ms** (g)
332	38.0ms	19.0ms	**1.665ms** (g)
333	33.0ms	17.0ms	**1.605ms** (g)
334	36.0ms	18.0ms	**1.636ms** (g)
335	37.0ms	19.0ms	**1.599ms** (g)
336	5.554ms	2.897ms	**0.558ms** (g)
337	5.19ms	2.021ms	**0.405ms** (g)
338	5.176ms	2.769ms	**0.446ms** (g)
339	5.06ms	2.417ms	**0.463ms** (g)
340	4.927ms	2.666ms	**0.464ms** (g)
341	5.501ms	3.787ms	**0.574ms** (g)
342	5.0ms	2.329ms	**0.428ms** (g)
343	5.313ms	3.073ms	**0.516ms** (g)
344	37.0ms	18.0ms	**1.679ms** (g)
345	38.0ms	19.0ms	**1.641ms** (g)
346	40.0ms	19.0ms	**1.501ms** (g)
347	37.0ms	18.0ms	**1.517ms** (g)
348	37.0ms	23.0ms	**16.0ms**
349	38.0ms	19.0ms	**1.738ms** (g)
350	39.0ms	19.0ms	**1.52ms** (g)
351	37.0ms	19.0ms	**1.671ms** (g)
352	37.0ms	17.0ms	**1.654ms** (g)
353	36.0ms	18.0ms	**1.686ms** (g)
354	37.0ms	19.0ms	**1.773ms** (g)
355	40.0ms	19.0ms	**2.036ms** (g)
356	36.0ms	19.0ms	**1.821ms** (g)
357	38.0ms	20.0ms	**1.548ms** (g)
358	36.0ms	19.0ms	**1.566ms** (g)
359	42.0ms	30.0ms	**2.01ms** (g)
360	49.0ms	39.0ms	**2.052ms** (g)
361	0.756ms	0.567ms	**0.479ms**
362	0.916ms	0.305ms	**0.283ms**
363	**0.342ms**	0.373ms	0.363ms
364	0.468ms	0.408ms	**0.282ms**

N°	**Vampire**	**1b naif**	**1b**
365	0.632ms	**0.397ms**	0.435ms
366	0.644ms	0.394ms	**0.349ms**
367	0.587ms	0.401ms	**0.313ms**
368	0.648ms	0.416ms	**0.31ms**
369	0.701ms	0.419ms	**0.331ms**
370	0.697ms	0.398ms	**0.317ms**
371	4.59ms	0.679ms	**0.1ms** (g)
372	4.636ms	0.695ms	**0.1ms** (g)
373	4.484ms	0.701ms	**0.1ms** (g)
374	4.521ms	0.678ms	**0.099ms** (g)
375	4.538ms	1.257ms	**0.166ms** (g)
376	5.16ms	0.942ms	**0.125ms** (g)

Tabella B.7: Tempi di esecuzione in millisecondi dei problemi Disjunctive Binding di Vampire, 1b naif e 1b

$N°$	**Vampire**	**1b naif**	**1b**
137	384Kb	382Kb	**382Kb**
138	548Kb	527Kb	**524Kb** (g)
139	4435Kb	4101Kb	**3920Kb** (g)
140	24297Kb	24585Kb	**23254Kb** (g)
141	571Kb	531Kb	**528Kb** (g)
142	4834Kb	4074Kb	**3893Kb** (g)
143	27971Kb	24575Kb	**23244Kb** (g)
144	520Kb	499Kb	**497Kb** (g)
145	4435Kb	4101Kb	**3920Kb** (g)
146	24297Kb	24585Kb	**23254Kb** (g)
147	577Kb	534Kb	**532Kb** (g)
148	4835Kb	4074Kb	**3909Kb** (g)
149	27971Kb	24576Kb	**23245Kb** (g)
150	571Kb	531Kb	**528Kb** (g)
151	4818Kb	4058Kb	**3877Kb** (g)
152	27842Kb	24447Kb	**23132Kb** (g)
153	579Kb	534Kb	**531Kb** (g)
154	4835Kb	4074Kb	**3909Kb** (g)
155	27971Kb	24576Kb	**23245Kb** (g)
156	1108Kb	887Kb	**883Kb** (g)
157	4834Kb	4074Kb	**3893Kb** (g)
158	27971Kb	24575Kb	**23244Kb** (g)
159	633Kb	537Kb	**534Kb** (g)
160	4834Kb	4074Kb	**3893Kb** (g)
161	27971Kb	24575Kb	**23244Kb** (g)
162	994Kb	887Kb	**884Kb** (g)
163	4834Kb	4074Kb	**3893Kb** (g)
164	27971Kb	24575Kb	**23244Kb** (g)
165	961Kb	798Kb	**795Kb** (g)
166	4834Kb	4074Kb	**3893Kb** (g)
167	27971Kb	24575Kb	**23244Kb** (g)
168	639Kb	540Kb	**538Kb** (g)
169	4834Kb	4074Kb	**3893Kb** (g)
170	27971Kb	24575Kb	**23244Kb** (g)
171	553Kb	532Kb	**529Kb** (g)
172	4435Kb	4101Kb	**3920Kb** (g)
173	24297Kb	24585Kb	**23254Kb** (g)
174	731Kb	668Kb	**665Kb** (g)

$N°$	**Vampire**	**1b naif**	**1b**
175	4818Kb	4058Kb	**3877Kb** (g)
176	27842Kb	24447Kb	**23132Kb** (g)
177	610Kb	538Kb	**535Kb** (g)
178	4435Kb	4101Kb	**3920Kb** (g)
179	24297Kb	24585Kb	**23254Kb** (g)
180	503Kb	432Kb	**429Kb** (g)
181	4834Kb	4074Kb	**3893Kb** (g)
182	27971Kb	24575Kb	**23244Kb** (g)
183	1117Kb	888Kb	**885Kb** (g)
184	4834Kb	4074Kb	**3893Kb** (g)
185	27971Kb	24575Kb	**23244Kb** (g)
186	758Kb	675Kb	**672Kb** (g)
187	4834Kb	4074Kb	**3893Kb** (g)
188	27971Kb	24575Kb	**23244Kb** (g)
189	762Kb	703Kb	**684Kb** (g)
190	4834Kb	4074Kb	**3893Kb** (g)
191	27971Kb	24575Kb	**23244Kb** (g)
192	505Kb	485Kb	**482Kb** (g)
193	4834Kb	4074Kb	**3893Kb** (g)
194	27971Kb	24575Kb	**23244Kb** (g)
195	571Kb	531Kb	**528Kb** (g)
196	4818Kb	4058Kb	**3877Kb** (g)
197	27842Kb	24447Kb	**23132Kb** (g)
198	530Kb	499Kb	**496Kb** (g)
199	4834Kb	4074Kb	**3893Kb** (g)
200	27971Kb	24575Kb	**23244Kb** (g)
201	875Kb	767Kb	**716Kb** (g)
202	4834Kb	4074Kb	**3893Kb** (g)
203	27971Kb	24575Kb	**23244Kb** (g)
204	894Kb	786Kb	**783Kb** (g)
205	4818Kb	4058Kb	**3877Kb** (g)
206	27842Kb	24447Kb	**23132Kb** (g)
207	730Kb	668Kb	**666Kb** (g)
208	4818Kb	4058Kb	**3877Kb** (g)
209	27842Kb	24447Kb	**23132Kb** (g)
210	571Kb	531Kb	**528Kb** (g)
211	4834Kb	4074Kb	**3893Kb** (g)
212	27971Kb	24575Kb	**23244Kb** (g)

$N°$	**Vampire**	**1b naif**	**1b**
213	570Kb	531Kb	**528Kb** (g)
214	4834Kb	4074Kb	**3893Kb** (g)
215	27971Kb	24575Kb	**23244Kb** (g)
216	503Kb	432Kb	**429Kb** (g)
217	4834Kb	4074Kb	**3893Kb** (g)
218	27971Kb	24575Kb	**23244Kb** (g)
219	662Kb	602Kb	**551Kb** (g)
220	4834Kb	4074Kb	**3893Kb** (g)
221	27971Kb	24575Kb	**23244Kb** (g)
222	638Kb	540Kb	**537Kb** (g)
223	4834Kb	4074Kb	**3893Kb** (g)
224	27971Kb	24575Kb	**23244Kb** (g)
225	610Kb	538Kb	**535Kb** (g)
226	4435Kb	4101Kb	**3920Kb** (g)
227	24297Kb	24585Kb	**23254Kb** (g)
228	577Kb	534Kb	**532Kb** (g)
229	4835Kb	4074Kb	**3909Kb** (g)
230	27971Kb	24576Kb	**23245Kb** (g)
231	460Kb	471Kb	**420Kb** (g)
232	424Kb	414Kb	**414Kb**
233	541Kb	502Kb	**502Kb**
234	428Kb	422Kb	**422Kb**
235	541Kb	502Kb	**502Kb**
236	447Kb	429Kb	**429Kb**
237	541Kb	502Kb	**502Kb**
238	433Kb	422Kb	**422Kb**
239	541Kb	502Kb	**502Kb**
240	446Kb	429Kb	**429Kb**
241	541Kb	502Kb	**502Kb**
242	446Kb	429Kb	**429Kb**
243	541Kb	502Kb	**502Kb**
244	389Kb	383Kb	**381Kb** (g)
245	390Kb	383Kb	**381Kb** (g)
246	395Kb	390Kb	**387Kb** (g)
247	395Kb	391Kb	**388Kb** (g)
248	396Kb	391Kb	**388Kb** (g)
249	399Kb	391Kb	**388Kb** (g)
250	397Kb	391Kb	**387Kb** (g)

$N°$	**Vampire**	**1b naif**	**1b**
251	398Kb	392Kb	**389Kb** (g)
252	399Kb	392Kb	**389Kb** (g)
253	399Kb	392Kb	**389Kb** (g)
254	402Kb	392Kb	**390Kb** (g)
255	416Kb	393Kb	**390Kb** (g)
256	417Kb	409Kb	**390Kb** (g)
257	420Kb	410Kb	**407Kb** (g)
258	388Kb	383Kb	**380Kb** (g)
259	383Kb	383Kb	**383Kb**
260	384Kb	384Kb	**384Kb**
261	781Kb	784Kb	**729Kb** (g)
262	835Kb	801Kb	**730Kb** (g)
263	6531Kb	4981Kb	**4944Kb** (g)
264	6403Kb	4576Kb	**4524Kb** (g)
265	6403Kb	4576Kb	**4524Kb** (g)
266	417Kb	407Kb	**404Kb** (g)
267	380Kb	380Kb	**377Kb** (g)
268	383Kb	382Kb	**379Kb** (g)
269	666Kb	623Kb	**619Kb** (g)
270	666Kb	623Kb	**619Kb** (g)
271	666Kb	623Kb	**619Kb** (g)
272	666Kb	623Kb	**619Kb** (g)
273	379Kb	380Kb	**376Kb** (g)
274	**379Kb**	380Kb	380Kb
275	**378Kb**	379Kb	379Kb
276	658Kb	583Kb	**580Kb** (g)
277	708Kb	582Kb	**579Kb** (g)
278	744Kb	699Kb	**616Kb** (g)
279	785Kb	702Kb	**667Kb** (g)
280	1321Kb	945Kb	**925Kb** (g)
281	1287Kb	925Kb	**922Kb** (g)
282	986Kb	897Kb	**846Kb** (g)
283	1450Kb	1102Kb	**1051Kb** (g)
284	1048Kb	930Kb	**927Kb** (g)
285	1461Kb	1152Kb	**1068Kb** (g)
286	1775Kb	1313Kb	**1246Kb** (g)
287	1852Kb	1318Kb	**1250Kb** (g)
288	1246Kb	1051Kb	**1048Kb** (g)

$N°$	**Vampire**	**1b naif**	**1b**
289	1245Kb	1039Kb	**1036Kb** (g)
290	1287Kb	1184Kb	**1053Kb** (g)
291	1292Kb	1118Kb	**1051Kb** (g)
292	1299Kb	1133Kb	**1050Kb** (g)
293	1680Kb	1309Kb	**1226Kb** (g)
294	1284Kb	1118Kb	**1051Kb** (g)
295	1253Kb	1042Kb	**1038Kb** (g)
296	1240Kb	1052Kb	**1049Kb** (g)
297	1235Kb	1117Kb	**1049Kb** (g)
298	1291Kb	1117Kb	**1050Kb** (g)
299	1737Kb	1311Kb	**1227Kb** (g)
300	1235Kb	1037Kb	**1034Kb** (g)
301	1247Kb	1052Kb	**1049Kb** (g)
302	1247Kb	1041Kb	**1038Kb** (g)
303	1782Kb	1314Kb	**1246Kb** (g)
304	1252Kb	1051Kb	**1047Kb** (g)
305	1260Kb	1168Kb	**1053Kb** (g)
306	1712Kb	1314Kb	**1231Kb** (g)
307	1285Kb	1118Kb	**1051Kb** (g)
308	1247Kb	1040Kb	**1037Kb** (g)
309	1239Kb	1052Kb	**1049Kb** (g)
310	1732Kb	1313Kb	**1229Kb** (g)
311	1403Kb	1223Kb	**1124Kb** (g)
312	1405Kb	1224Kb	**1125Kb** (g)
313	1480Kb	1242Kb	**1127Kb** (g)
314	1313Kb	1184Kb	**1069Kb** (g)
315	1413Kb	1225Kb	**1126Kb** (g)
316	1437Kb	1243Kb	**1128Kb** (g)
317	1514Kb	1296Kb	**1293Kb** (g)
318	1505Kb	1297Kb	**1246Kb** (g)
319	1486Kb	1295Kb	**1180Kb** (g)
320	1512Kb	1296Kb	**1245Kb** (g)
321	1499Kb	1296Kb	**1245Kb** (g)
322	1493Kb	1245Kb	**1178Kb** (g)
323	1468Kb	1243Kb	**1128Kb** (g)
324	1403Kb	1241Kb	**1126Kb** (g)
325	1515Kb	1319Kb	**1299Kb** (g)
326	1607Kb	1321Kb	**1318Kb** (g)

$N°$	**Vampire**	**1b naif**	**1b**
327	1485Kb	1243Kb	**1176Kb** (g)
328	1569Kb	1321Kb	**1318Kb** (g)
329	1497Kb	1296Kb	**1181Kb** (g)
330	1512Kb	1301Kb	**1298Kb** (g)
331	1504Kb	1299Kb	**1296Kb** (g)
332	1567Kb	1320Kb	**1301Kb** (g)
333	1340Kb	1221Kb	**1122Kb** (g)
334	1512Kb	1295Kb	**1244Kb** (g)
335	1522Kb	1317Kb	**1298Kb** (g)
336	515Kb	486Kb	**467Kb** (g)
337	500Kb	477Kb	**474Kb** (g)
338	514Kb	487Kb	**468Kb** (g)
339	504Kb	488Kb	**469Kb** (g)
340	505Kb	487Kb	**469Kb** (g)
341	521Kb	486Kb	**467Kb** (g)
342	504Kb	471Kb	**468Kb** (g)
343	516Kb	487Kb	**484Kb** (g)
344	1515Kb	1318Kb	**1299Kb** (g)
345	1571Kb	1322Kb	**1319Kb** (g)
346	1635Kb	1418Kb	**1415Kb** (g)
347	1510Kb	1317Kb	**1298Kb** (g)
348	1521Kb	1316Kb	**1316Kb**
349	1580Kb	1321Kb	**1318Kb** (g)
350	1578Kb	1321Kb	**1318Kb** (g)
351	1521Kb	1315Kb	**1296Kb** (g)
352	1553Kb	1320Kb	**1317Kb** (g)
353	1503Kb	1316Kb	**1297Kb** (g)
354	1521Kb	1316Kb	**1297Kb** (g)
355	1636Kb	1415Kb	**1412Kb** (g)
356	1497Kb	1294Kb	**1179Kb** (g)
357	1570Kb	1321Kb	**1318Kb** (g)
358	1516Kb	1297Kb	**1294Kb** (g)
359	1561Kb	1238Kb	**1154Kb** (g)
360	1777Kb	1314Kb	**1247Kb** (g)
361	385Kb	381Kb	**381Kb**
362	386Kb	383Kb	**383Kb**
363	**377Kb**	380Kb	380Kb
364	**379Kb**	381Kb	381Kb

$N°$	**Vampire**	**1b naif**	**1b**
365	**379Kb**	381Kb	381Kb
366	381Kb	381Kb	**381Kb**
367	381Kb	381Kb	**381Kb**
368	**381Kb**	382Kb	382Kb
369	**381Kb**	382Kb	382Kb
370	382Kb	382Kb	**382Kb**
371	631Kb	618Kb	**612Kb** (g)
372	631Kb	617Kb	**611Kb** (g)
373	630Kb	617Kb	**611Kb** (g)
374	630Kb	617Kb	**611Kb** (g)
375	629Kb	616Kb	**610Kb** (g)
376	697Kb	685Kb	**679Kb** (g)

Tabella B.8: Memoria di esecuzione in kilobyte dei problemi Disjunctive Binding (FOF) di Vampire, 1b naif e 1b

Bibliografia

[1] vampire website. `https://vprover.github.io/`.

[2] Simone Bova and Fabio Mogavero. Herbrand property, finite quasi-herbrand models, and a chandra-merlin theorem for quantified conjunctive queries. In *2017 32nd Annual ACM/IEEE Symposium on Logic in Computer Science (LICS)*, pages 1–12, 2017.

[3] M. Davis. *Il calcolatore universale. Da Leibniz a Turing*. Biblioteca scientifica. Adelphi, 2003.

[4] Krystof Hoder and Andrei Voronkov. Comparing unification algorithms in first-order theorem proving. pages 435–443, 09 2009.

[5] D.R. Hofstadter. *Gödel, Escher, Bach: un'eterna ghirlanda brillante ; una fuga metaforica su menti e macchine nello spirito di Lewis Carroll*. Biblioteca scientifica / Adelphi. Adelphi, 2009.

[6] Laura Kovács and Andrei Voronkov. First-order theorem proving and vampire. In Natasha Sharygina and Helmut Veith, editors, *Computer Aided Verification*, pages 1–35, Berlin, Heidelberg, 2013. Springer Berlin Heidelberg.

[7] Fabio Mogavero and Giuseppe Perelli. Binding Forms in First-Order Logic. In Stephan Kreutzer, editor, *24th EACSL Annual Conference on Computer Science Logic (CSL 2015)*, volume 41 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 648–665, Dagstuhl, Germany, 2015. Schloss Dagstuhl – Leibniz-Zentrum für Informatik.

[8] Nonnengart, Andreas and Rock, Georg and Weidenbach, Christoph. On Generating Small Clause Normal Forms. 2000.

[9] Alexandre Riazanov and Andrei Voronkov. The design and implementation of vampire. *AI Commun.*, 15:91–110, 01 2002.

[10] G. Sutcliffe. The TPTP Problem Library and Associated Infrastructure. From CNF to TH0, TPTP v6.4.0. *Journal of Automated Reasoning*, 59(4):483–502, 2017.

[11] G. Sutcliffe. The Logic Languages of the TPTP World. *Logic Journal of the IGPL*, 2022.

[12] Dirk van Dalen. *Logic and Structure*. Universitext. Springer London, 2012.