Capitolo 1

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 **??** utilizzando gli strumenti e le funzionalità descritte nel capitolo **??** 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 1.1. 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.

1.1 Preprocessing

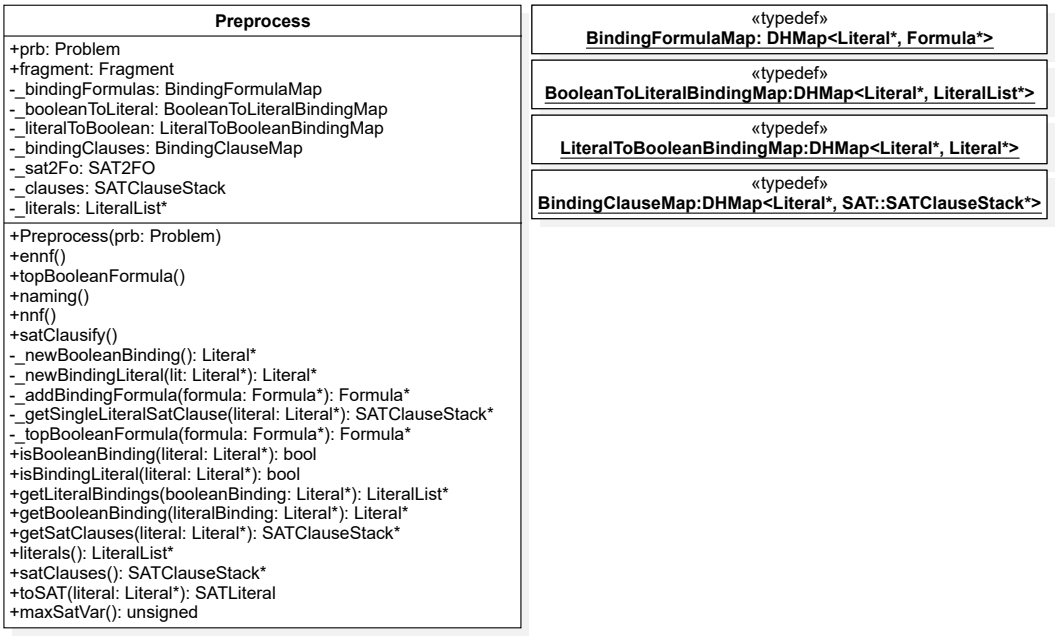


Figura 1.1: Struttura del Preprocessing

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 né la clausificazione né il *naming* sono processi conservativi rispetto ai frammenti. Ad esempio, la semplice formula $\forall x_1(p_1(x_1)) \vee p_2$ del frammento *1B* diventa, una volta portata in forma causale, $\{\{p_1(x_1), p_2\}\}$ che fa parte del frammento DB . L'approccio utilizzato è stato quello di creare prima una nuova formula ground che rappresenta la struttura booleana esterna della formula originale, successivamente 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 arietà 0 saranno chiamati *booleanBinding* e saranno associati a una formula del frammento *1B* o *CB* . I predicati di Σ_b con arietà $n > 0$ saranno chiamati *literalBindign* e fungeranno da rappresentati dei τ-Binding delle formule *1B* . Il preprocessing seguirà pressoché 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 associate ai 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 1.

Algorithm 1: 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);
    case $A[\wedge, \vee]B$ **do**
        return new JunctionFormula(topBooleanFormula(A), connective of φ,
         topBooleanFormula(B));
    case $\neg A$ **do**
        return new NegatedFormula(topBooleanFormula(A));
    case $[\forall, \exists]A$ **do**
        $b = newBooleanBinding()$;
        $bindingFormulas[b] := \varphi$;
        return new AtomicFormula(b);
    case $A[\leftrightarrow, \rightarrow, \oplus]B$ **do**
        return new BinaryFormula(A, connective of φ, B);

L'algoritmo prende in input una formula rettificata e restituisce una formula ground, sostituendo le sottoformule quantificate con un nuovo booleanBinding e aggiungendo la sottoformula originale alla mappa bindingFormulas. Da adesso in poi, qualunque modifica fatta alla FBE preserverà l'appartenenza al frammento originale. I passi 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
    $positiveFormula := bindingFormulas[positiveLiteral(l)]$
    $bindingFormulas[l] := newNegatedFormula(positiveFormula)$

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**
        $bindingClauses[l] := newSatClause\{toSat(l)\}$

Per essere clausificate, le formule della mappa bindingFormulas vanno trasformate in NNF, e poi portate in forma di Skolem. Anche in questo caso vengono utilizzate le funzioni standard di Vampire. Ogni booleanBinding è associato a una formula del frammento ConjunctiveBinding. A questo scopo, successivamente alla trasformazione in forma Skolem, il quantificatore universale viene distribuito sul connettivo $\wedge$, in modo da 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 (tali termini sono gli stessi di tutti i letterali della sottoformula). Successivamente, alla formula viene applicata l'operazione SatClausificata. Tale operazione aggiunge alla mappa satClauses la coppia composta dal nuovo LiteralBinding

e le satClausole della sottoformula. Inoltre, 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"
  else
    toDo.add(formula)
  $literalBindings := \emptyset$
  while *todo* $\neq \emptyset$ **do**
    $subformula := todo.pop()$
    $literalBinding := newLiteralBinding(subformula.mostLeftLiteral())$
    $clauses := SatClausifyBindingFormula(subFormula)$

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

  $booleanBindingToLiteral[booleanBinding] := literalBindings$

La funzione SatClausifyBindingFormula è una funzione che prende in input una formula in forma clausale e converte tutte le clausole in SatClausole, in modo che ogni satLetterale abbia lo stesso indice del funtore del predicato associato. Questa operazione 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 passo 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 connettivo '$\rightarrow$':

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

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 *naming* 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 messe in forma di Skolem 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 sul connettivo $\wedge$ e creati due literalBindings. La trasformazione in forma di Skolem della formula associata a $\neg b_3$ porta alla formula:

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

Dove sk_1 è una nuova costante di Skolem. Vengono create così le mappe booleanBindingToLiteral e la sua inversa literalToBooleanBindings, come riportato nella tabella 1.1.

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

Tabella 1.1: Esempio di booleanBindingToLiteral e literalToBooleanBindings

Le formule associate ai literalBindings vengono messe in forma clausale:

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

E successivamente trasformate in clausole proposizionali e associate ai literalBindings:

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

Gli ultimi due passi sono la trasformazione in forma clausale 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 creazione delle corrispondenti clausole proposizionali 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 hashMap bidirezionale che associa ogni satLetterale ad un letterale:

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

1.2 Procedura di Decisione

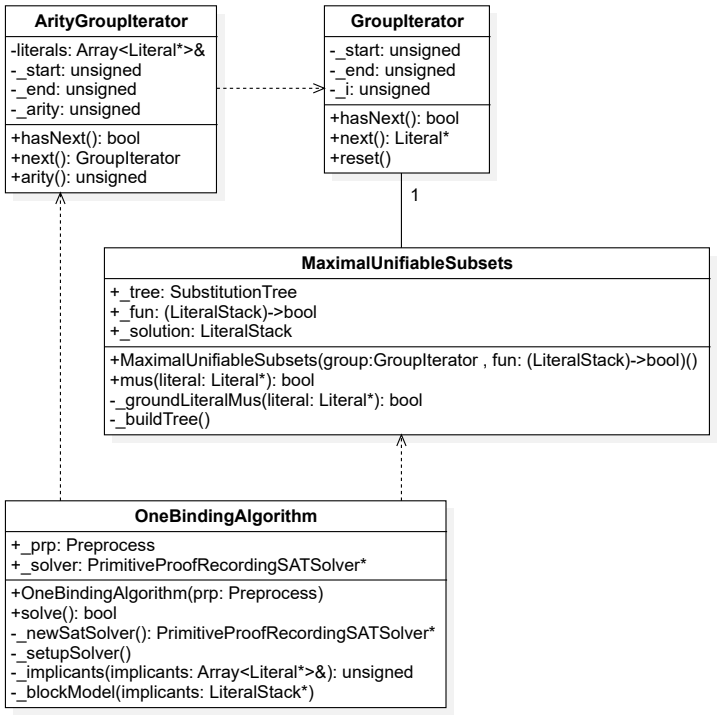


Figura 1.2: Struttura dell'algoritmo di decisione

In questa sezione verrà descritta l'implementazione dell'algoritmo **??** per la decisione dei per frammenti Binding descritto nel capitolo **??**. 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 preparata tramite il *preprocessing* sopra descritto, con tutte le strutture ausiliarie.

La ricerca degli implicanti è la parte più facile da implementare (sat esterna). Data la FBE in forma di SatClausole, 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 **??** 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 arietà. È quindi possibile ordinare l'insieme di implicanti in base all'arietà e ricercare, per ogni 'Gruppo di Arietà', tutti i sottoinsiemi unificabili. Già in questo modo si riduce notevolmente lo spazio di ricerca, eliminando tutti quei sottoinsiemi composti da letterali di arietà 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 a 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 unificabile di letterali. 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 x fa parte della soluzione, se $S[x] = 0$, allora non fa parte della soluzione, infine, se $S[x] = -1$, allora vuol dire che non fa parte della soluzione e deve essere escluso dalle ricerche future (Per $S[x]$ si intende il valore associato ad x nella mappa S). Prima di iniziare la ricerca, va impostato l'ambiente in modo tale che il SubstitutionTree contenga tutti i letterali del gruppo di arietà corrente e S associ tutti i letterali a 0. Viene fornita anche una funzione *fun*, che prende in input l'insieme soluzione e restituisce un booleano. La funzione calcolata dall'algoritmo 2 è la funzione principale che inizia la catena di chiamate ricorsive.

Algorithm 2: 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

```
1 if S[literal] ≠ 0 then
  |  return ⊤;
2 if literal is ground then
  |  return groundLiteralMus(literal);
  S[literal] = 1;
  tmpToFree := ∅;
  res := mus(literal, tmpToFree);
  foreach i ∈ tmpToFree do
  |  S[i] = −1;
  S[literal] = −1;
  return res;
```

Inizialmente verifica se il letterale è già stato esplorato e, in tal caso, restituisce $\top$. Se il letterale è ground, allora chiama l'algoritmo 4, che è un'ottimizzazione pensata per semplificare la ricerca per 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, successivamente, viene richiamato l'algoritmo 3, che prende in input il letterale e un insieme di letterali.

Algorithm 3: Maximal Unifiable Subsets

Firma: mus($literal$, $FtoFree$)
Input: $literal$ un puntatore ad un letterale, $FtoFree$ un puntatore ad una lista di letterali
Output: $\top$ o $\bot$
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};
            else
                toFree := toFree ∪ {u};
        else
            isMax = ⊥;
            tmpToFree := ∅;
            if ¬mus(l, tmpToFree) then
                return ⊥;
            S[u] = −1;
            foreach i ∈ tmpToFree do
                S[i] = −1;
            toFree := toFree ∪ {u} ∪ tmpToFree;
if isMax then
    if ¬fun({x | S[x] = 1}) then
        return ⊥;
while toFree ≠ ∅ do
    S[toFree.pop()] = 0;
return ⊤;
```

La funzione descritta dall'algoritmo 3 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. Quindi, per 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 3, dopo aver inizializzato la lista $toFree$ si itera su tutti i letterali che unificano con il letterale *literal* in input. Per ogni letterale u, se esso è già contenuto nella soluzione o è stato bloccato, allora viene ignorato, altrimenti u 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 perché u ha esattamente gli stessi termini di *literal*, quindi tutti i mus che contengono u contengono anche *literal*. Il letterale 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, u 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 con 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 $\top$.

La lista $toFree$ è quindi la lista di letterali che devono essere sbloccati alla fine dell'esecuzione del livello corrente. La lista $tmpToFree$ è una lista temporanea che viene passata in input alla chiamata ricorsiva. Alla fine della chiamata ricorsiva, $tmpToFree$ contiene tutti i letterali che hanno la stessa lista di termini di $literal^{\sigma}$. Questi letterali vengono bloccati e aggiunti a $toFree$ per essere sbloccati alla fine dell'esecuzione del livello corrente. La lista $FtoFree$ rappresenta la lista $tmpToFree$ passata in input dal livello superiore.

Algorithm 4: 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 ⊤;
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;
        solution := solution ∪ {u};
return fun(solution);
```

L'algoritmo 4 è un'ottimizzazione dell'algoritmo 3 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 questo algoritmo è pari al costo della visita nel SubstitutionTree che viene effettuata tramite la funzione *getUnifications* e l'iteratore *uit*, più il costo della chiamata della funzione *fun*. In termini di quante volte viene visitato interamente il SubstitutionTree, l'algoritmo 4 visita l'albero esattamente una sola volta, mentre l'algoritmo 3 potrebbe visitare l'albero per intero ad ogni chiamata ricorsiva. Il numero di chiamate ricorsive effettuabili dall'algoritmo 3 è limitato superiormente dall'equazione di ricorrenza riportata di seguito.

$$T(n) = \begin{cases} 1 & \text{se } n = 1 \\ \sum_{i=1}^{n-1} T(n-i) & \text{altrimenti} \end{cases}$$

Dove n è il numero di letterali nel gruppo di arietà. Si può notare che il secondo caso si può riscrivere come: $T(n) = \sum_{i=1}^{n-1} T(i)$. Si nota inoltre che $T(n) = \sum_{i=1}^{n-2} T(i) + T(n-1)$ e $T(n-1) = \sum_{i=1}^{n-2} T(i)$ quindi $T(n) = 2T(n-1)$. Applicando il metodo iterativo si ottiene che $T(n) = O(2^n)$. In conclusione, l'algoritmo 3 potrebbe visitare il SubstitutionTree un numero di volte esponenziale, mentre l'algoritmo 4 lo visita esattamente una sola volta.

Procedura di decisione

Algorithm 5: Algoritmo di decisione

Firma: solve(prp)
Input: *prp* il problema pre-processato
Output: $\top$ o $\bot$

```
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 ⊤;
    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;
3       if res = ⊤ then
            return ⊤;
return ⊥;
```

Dato il problema già sottoposto a *preprocessing* l'algoritmo 5 inizializza 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 riportata nell'algoritmo 6. Dopo aver ottenuto l'insieme di implicanti, l'insieme viene ordinato in base all'arietà dei letterali tramite la funzione sortImplicants. La funzione sortImplicants è stata realizzata incorporando varie euristiche. La prima euristica che è stata implementata è quella di anticipare i letterali ground all'inizio di ogni gruppo di arietà in modo da utilizzare l'algoritmo 4 per ridurre il numero di chiamate ricorsive che verrebbero fatte dall'algoritmo

3. La seconda euristica è quella di ordinare i letterali in base ai sottotermini, in modo che sequenze di letterali che hanno stessi termini siano vicini tra loro, così da poter sfruttare l'ottimizzazione vista nell'algoritmo 3.

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

Altrimenti, per ogni gruppo di arietà, si imposta l'ambiente per la ricerca dei mus e viene chiamata la funzione mus dell'algoritmo 2 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 $\bot$.

Algorithm 6: 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);
        else
            implicants := implicants ∪ {l};
return implicants;
```

La funzione getImplicants, rappresentata nell'algoritmo 6, viene chiamata per ottenere l'insieme degli implicanti dopo aver ottenuto un risultato positivo dal satSolver. Per ogni letterale del problema viene recuperato il corrispondente satLetterale. Se il satLetterale è soddisfatto dall'assegnamento, allora il letterale corrispondente o è un booleanBinding o è un letterale ground. Nel primo caso vengono aggiunti all'insieme di implicanti tutti i literalBindings associati al booleanBinding. Nel secondo caso, invece, viene aggiunto direttamente il letterale all'insieme di implicanti.

Algorithm 7: Sat interna

```
Firma: internalSat(literals)
Input: literals una lista di letterali
Output: ⊤ o ⊥
if literals.length = 1 And getSatClauses(literals.top()).length = 1 then
    return ⊤;
satSolver := newSatSolver();
foreach l ∈ literals do
    satSolver.addClause(getSatClauses(l));
return satSolver.solve() = SATISFIABLE;
```

La funzione internalSat, rappresentata nell'algoritmo 7, viene chiamata ogni volta che l'algoritmo 3 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 contenere contraddizioni, 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 $\bot$.

Algoritmo ottimizzato e algoritmo *naif*

Nel corso della progettazione e 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 *naif* ci si riferirà all'algoritmo che non sfrutta le euristiche descritte nelle sezioni precedenti. In particolare l'algoritmo *naif* non include i blocchi di codice numerati (1) e (2) nell'algoritmo 2 e il blocco numerato con (1) nell'algoritmo 3. Inoltre, nell'algoritmo 5 il blocco numerato con (1) era posto fuori dal ciclo while e veniva controllato se tutta la formula fosse ground. In tal caso, veniva restituito direttamente il valore della funzione solve del SatSolver. L'ordinamento degli implicanti nell'algoritmo *naif* è effettuato solo in base all'arietà dei letterali, mentre nell'algoritmo ottimizzato vengono spinti i letterali ground all'inizio di ogni gruppo di arietà e vengono ordinati in base ai sottotermini. Come ultima modifica, sempre sull'algoritmo 5, 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.

1.3 Algoritmo di Classificazione

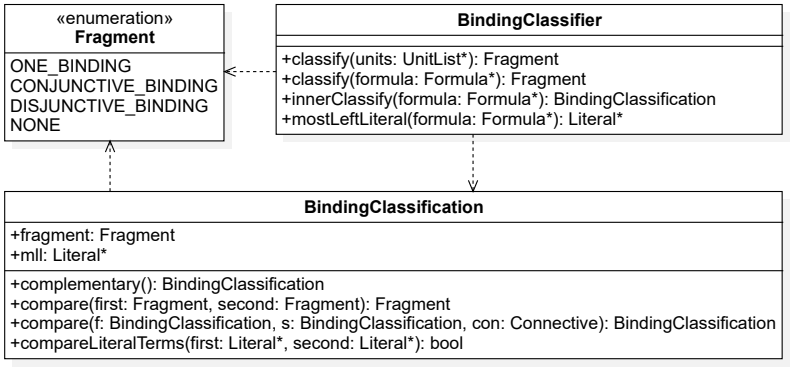


Figura 1.3: Classificatore

La precondizione più importante per la correttezza dell'algoritmo di decisione è che la formula originale in ingresso 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, per capire a quale linguaggio generato, dalle grammatiche descritte nella sezione **??**, appartine. Per questo scopo, sono state create due funzioni, chiamate Classificatore Esterno (Algoritmo 8) e Classificatore Interno (Algoritmo 10). La prima opera sulla struttura booleana esterna alle quantificazioni, corrispondente a una combinazione booleana di proposizioni quantificate, mentre la seconda si occupa delle sottoformule contenenti i quantificatori e confronta i termini contenuti nei letterali. Entrambi gli algoritmi seguono la struttura di una visita in *post-order* sull'albero sintattico della formula e hanno una complessità lineare rispetto alla dimensione della formula.

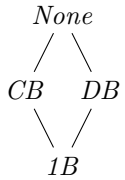
Algorithm 8: Classificatore esterno

Firma: classify(φ) **Input:** φ Una formula rettificata
Output: Un elemento dell'enumerazione Fragment
switch φ **do**
  case *Literal* **do**
    return ONE_BINDING;
  case $A[\wedge, \vee]B$ **do**
    return $compare(classify(A), classify(B))$;
  case $\neg A$ **do**
    return $classify(A).complementary()$;
  case $[\forall, \exists]A$ **do**
    $sub := \varphi$;
    connective := connective of φ;
    repeat
      sub := subformula of sub;
      connective := connective of sub;
    until *connective* $\notin \{\forall, \exists\}$;
    $(fragment, \_) := innerClassify(sub)$;
    return $fragment$;
  case $A \leftrightarrow B$ **do**
    return $compare(classify(A \rightarrow B), classify(B \rightarrow A))$;
  case $A \oplus B$ **do**
    return $classify(A \leftrightarrow B).complementary()$;
  case $A \rightarrow B$ **do**
    return $compare(classify(\neg A), classify(B))$;

Algorithm 9: 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;
if $One\_Binding \notin \{A, B\}$ **then**
  return $None$;
return $max(A, B)$;

Il classificatore esterno si appoggia a 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 la seguente struttura a rombo:



In questo ordinamento parziale, *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 in ingresso. 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 10: Classificatore interno

Firma: innerClassify(φ) **Input:** φ Una formula rettificata
Output: Una coppia (Fragment, Literal)
switch φ **do**
  case *Literal* l **do**
    return $(ONE\_BINDING, l)$;
  case $A[\wedge, \vee]B$ **do**
    return $innerCompare(innerClassify(A), innerClassify(B), \text{connective of } \varphi)$;
  case $\neg A$ **do**
    return $innerClassify(A).complementary()$;
  case $A[\rightarrow, \leftrightarrow, \oplus]B$ **do**
    return $innerCompare(innerClassify(A), innerClassify(B), \text{connective of } \varphi)$;
  else
    return $(None, null)$;

La struttura del classificatore interno (*innerCompare*) è molto simile a quella del classificatore esterno, mentre il comparatore interno, è leggermente più complesso. Il caso base si ha quando la formula è un singolo letterale, che è sempre un *1B* . La visita in *post-order* 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 un riferimento alla lista di termini delle formule del frammento *1B* .

Algorithm 11: 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;
    else if $conn = \wedge$ **then**
      return $(Conjunctive\_Binding, null)$;
    else if $conn = \vee$ **then**
      return $(Disjunctive\_Binding, null)$;
  case *[One_Binding, Conjunctive_Binding | Conjunctive_Binding, One_Binding],* $\wedge$ **do**
    return (Conjunctive_Binding, null);
  case *[One_Binding, Disjunctive_Binding | Disjunctive_Binding, One_Binding],* $\vee$ **do**
    return (Disjunctive_Binding, null);
  case *Conjunctive_Binding, Conjunctive_Binding,* $\wedge$ **do**
    return (Conjunctive_Binding, null);
  case *Disjunctive_Binding, Disjunctive_Binding,* $\vee$ **do**
    return (Disjunctive_Binding, null);
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. La congiunzione di formule del frammento *CB* rimane nel frammento *CB* . Analogamente, la congiunzione

di una formula del frammento *1B* e una di quello *CB* , fa parte del frammento *CB* . Stesso discorso per i *DB* e il connettivo $\vee$. In tutti gli altri casi, la combinazione è *None*. Nell'algoritmo 11 sono stati omessi i casi con connettivi $\rightarrow, \leftrightarrow$ e $\oplus$, in quanto sono riconducibili a formule composte da $\wedge$ e $\vee$, come è stato fatto ad esempio nell'algoritmo 8. Con la funzione *complementary* applicata ad una coppia, cioè (Fragment, Literal).complementary(), si intende la coppia (Fragment.complementary(), Literal).