Abduction in first order logic with equality

Sophie Tourret

Communauté
UNIVERSITÉ Grenoble Alpes

THÈSE

Pour obtenir le grade de

DOCTEUR DE L'UNIVERSITÉ DE GRENOBLE

Spécialité : **Informatique**

Arrêté ministériel : le 7 août 2006

Présentée par

Sophie Tourret

Thèse dirigée par **Nicolas Peltier**
et codirigée par **Mnacho Echenim**

préparée au sein du **Laboratoire d'Informatique de Grenoble**
et de l'**Ecole Doctorale MSTII**

Prime Implicate Generation in Equational Logic

Thèse soutenue publiquement le **3 mars 2016**,
devant le jury composé de :

Prof. Dr. Éric Gaussier
Professeur à l'université Joseph Fourier (Grenoble, France), Président
Prof. Dr. Viorica Sofronie-Stokkermans
Professeur à l'université de Coblence-Landau (Allemagne), Rapporteur
Prof. Dr. Stephan Schulz
Professeur à la Baden-Wuerttemberg Cooperative State University Stuttgart (Allemagne), Rapporteur
Dr. Silvio Ranise
Chercheur sénior à la Fondation Bruno Kessler (Trento, Italie), Examinateur
Dr. Nicolas Peltier
Chargé de Recherche CNRS (Grenoble, France), Directeur de thèse
Dr. Mnacho Echenim
Maître de Conférences Grenoble INP - Ensimag (France), Co-Directeur de thèse

Abstract

The work presented in this thesis deals with the generation of prime implicates in ground equational logic, i.e., of the most general consequences of formulæ containing equations and disequations between ground terms. It is divided into three parts. First, two calculi that generate implicates are defined. Their deductive-completeness is proved, meaning they can both generate all the implicates up to redundancy of equational formulæ. Second, a tree data structure to store the generated implicates is proposed along with algorithms to detect redundancies and prune the branches of the tree accordingly. This data structure is adapted to the different kinds of clauses (with and without function symbols, with and without constraints) and to the various formal definitions of redundancy used in the calculi since each calculus uses different – although similar – redundancy criteria. Termination and correction proofs are provided with each algorithm. Finally, an experimental evaluation of the different prime implicate generation methods based on research prototypes written in Ocaml is conducted including a comparison with state-of-the-art prime implicate generation tools. This experimental study is used to identify the most efficient variants of the proposed algorithms. These show promising results overstepping the state of the art.

Résumé

Introduction. Ce mémoire présente le résultat de mon travail de thèse sur la génération d'impliqués premiers en logique équationnelle. En logique formelle, étant donné une formule logique S, un *impliqué* de S est une clause C telle que S a pour conséquence logique C, ce qui est noté $S \models C$. Un impliqué C est dit *premier* si tout autre impliqué C' tel que $C' \models C$ est aussi tel que $C \models C'$. Les applications de la génération d'impliqués sont, entre autre, la minimisation de circuits booléens et la maintenance de la cohérence des bases de données. Une autre application importante est le raisonnement abductif qui, étant donné une théorie $\mathcal{T}$ et une observation O non expliquée par la théorie ($\mathcal{T} \not\models \mathcal{O}$), génère des hypothèses H telles que H satisfait la théorie ($\mathcal{T} \cup \mathcal{H} \not\models \square$) et H explique O ($\mathcal{T} \cup \mathcal{H} \models \mathcal{O}$). Le calcul abductif présenté dans [23] est d'ailleurs à l'origine des travaux exposés dans ce mémoire. Le calcul en question, qui produit des explications construites à base de constantes abducibles, présente le défaut de générer ses résultats sous forme d'impliqués quelconques d'une formule, ce qui peut être inefficace si l'on considère la quantité de clauses équivalentes, quantité qui tend à croître de façon exponentielle avec la taille des clauses en question. Le calcul des impliqués premiers de ces résultats est une solution à ce problème, suggérée dans [23], mais mise en place de façon inefficace. Les travaux présentés dans ce mémoire vont même plus loin puisque le calcul d'impliqués premiers est étendu aux formules incluant des symboles de fonction non-constantes. Ces résultats ont été précédemment publiés dans les articles [24, 25, 26, 27, 28, 29]. La répartition du contenu des articles dans les différents chapitres de ce mémoire est indiquée dans la table 1, page 12.

Chapitre i - État de l'art. Ce chapitre présente un état de l'art de la génération d'impliqués premiers en logique propositionnelle, en logique modale et en logique du premier ordre. Historiquement la notion d'impliqué premier fut introduite dans les années cinquante par Quine [57] pour la logique propositionnelle. Les premières méthodes de calcul d'impliqués étaient basées sur le *minterm* des formules, comme illustré dans l'exemple i.3, page 14. De manière générale, ces méthodes sont trop inefficaces pour être d'une quelconque utilité. Deux nouveaux types de méthodes se sont développées à partir des années soixante-dix, celles basées sur le calcul de résolution [4] et celles utilisant une méthode par décomposition.

Parmi les méthodes se basant sur la résolution [17, 39, 41, 70, 76], celle qui nous intéresse particulièrement, nommée CLTMS, est présentée dans [17]. Celle-ci organise les clauses générées dans une structure de données arborescente appelée un *trie* (cf. figure 4, page 4) telle que :

— ses arêtes sont étiquetées par les littéraux des clauses considérées,
— les arêtes sœurs sortant d'un même nœud sont ordonnées selon leur étiquette,
— l'ensemble des littéraux étiquetant une branche entre la racine et une feuille est la clause qui sert d'étiquette à la feuille terminant la branche.

L'intérêt de cette structure de données est qu'elle permet la détection efficace des redondances. Comme l'ensemble des autres méthodes par résolution, CLTMS utilise ce calcul pour générer les impliqués de la formule. L'usage des tries et algorithmes correspondant lui permet d'accélérer la détection et la suppression des impliqués non-premiers.

Le point commun des algorithmes n'utilisant pas la résolution [9, 15, 30, 33, 38, 40, 43, 47, 48, 52, 55, 56, 58, 59, 67, 71] est qu'ils construisent les impliqués premiers en les décomposant d'une manière ou d'une autre et en fusionnant les résultats obtenus. La plus récente de ces méthodes [56] décompose chaque littéral en deux nouveaux littéraux, un pour les occurrences positives du littéral d'origine et l'autre pour ses occurrences négatives. Cette décomposition est utilisée pour générer une formule équivalente à la première et dont les modèles sont les impliqués premiers de la formule initiale (cf. exemples i.8 et i.9, page 21). Cette méthode est, à notre connaissance, la plus efficace existante actuellement.

Au-delà de la logique propositionnelle, un certain nombre de barrières théoriques, comme la semi-décidabilité du problème en logique du premier ordre, rendent la génération d'impliqués premiers plus délicate. En logique du premier ordre, nous n'avons trouvé que deux méthodes réalistes et comportant des applications pratiques [22, 36]. Cependant, celle de [36] n'est pas capable de gérer efficacement les formules équationnelles pour des raisons intrinsèques. L'autre méthode [22] est spécialisée dans la génération d'impliqués correspondant à des instanciations partielles de variables. Bien qu'elle soit capable de manipuler des formules équationnelles, elle ne peut donc pas générer l'ensemble des impliqués premiers correspondant. En logique modale, le problème principal est de s'accorder sur la définition même d'impliqué premier [8]. Une fois celle-ci fixée, la plupart des méthodes utilisées en logique propositionnelle sont applicables.

Chapitre ii - Logique équationnelle. Dans ce mémoire, deux formalismes différents sont utilisés.

— La logique équationnelle $\mathbb{E}_0$ n'accepte que des formules plates (i.e. ne contenant pas de symboles de fonctions d'arité supérieure ou égale à un) et fermées (i.e. ne contenant pas de variables). Cette logique est donc uniquement bâtie sur des littéraux de la forme $a \simeq b$ et $c \not\simeq d$.

— La logique équationnelle $\mathbb{E}_1$ étend $\mathbb{E}_0$ en autorisant des termes non-plats à apparaître dans ses formules. Par exemple $f(a) \simeq b$ et $g(a, b) \not\simeq c$, qui sont interdits dans $\mathbb{E}_0$, sont acceptés dans $\mathbb{E}_1$.

Les définitions usuelles de la logique équationnelle sont rappelées. Pour les besoins de ce résumé, notons en particulier qu'une clause unitaire est composée d'un unique littéral et qu'une clause atomique est une clause unitaire positive, c'est à dire que le littéral qui la compose est une équation.

Différents ordres sont définis.

— L'ordre $\prec$ est l'ordre usuel des clauses obtenu en étendant l'ordre $\prec$ des termes (cf. définition ii.10, page 29)

— L'ordre $<_\pi$ est similaire à $\prec$ mais considère que n'importe quelle équation est plus grande que toutes les inéquations.

La notion de calcul formel, i.e. un ensemble de règles d'inférence, est présentée ainsi que le calcul de paramodulation standard [4]. Les notions associées de redondance, correction, complétude pour la déduction et saturation sont aussi définies. Une procédure standard de saturation d'un ensemble de clauses par un calcul est donnée. Il s'agit de l'algorithme *Given Clause* [68] (cf. algorithme 1, page 34) dans lequel les clauses sont initialement stockées dans un ensemble de clauses en attente puis extraites une par une de cet ensemble afin d'être utilisées par les règles du calcul avant d'être stockées avec les clauses déjà examinées, les nouvelles clauses générées étant ajoutées à l'ensemble des clauses en attente. Les clauses redondantes sont éliminées au fur et à mesure de leur détection. Cette procédure termine lorsqu'il ne reste plus de clauses en attente.

Partie I - Des calculs pour la génération d'impliqués.

Cette partie du mémoire regroupe les différents calculs de génération d'impliqués qui ont été développés durant ma thèse.

Chapitre 1 - Implication logique et représentation des clauses équationnelles. En logique propositionnelle, la détection de redondances est facile car elle s'apparente à un simple test d'inclusion et deux clauses sont équivalentes si et seulement si elles sont identiques. Ce n'est pas le cas en logique équationnelle. Dans ce contexte, même la détection de clauses équivalentes de façon syntaxique est non-triviale, ainsi il n'est pas forcément clair que les clauses $a \not\simeq b \vee a \simeq c$ et $b \not\simeq a \vee c \simeq b$ sont équivalentes. Pour cette raison, nous avons défini les notions suivantes : soit C, D des clauses, s un terme et l un littéral.

— La *C-classe d'équivalence* de s, notée $[s]_C$, est définie par $\{t \mid \neg C \models s \simeq t\}$. La relation d'équivalence associée est $\equiv_C$.

— Le C-*représentant* de s, l ou D, aussi appelé la *projection* de C sur cet élément, est défini par :

$$\begin{aligned} s_{\downharpoonright C} &\stackrel{\text{def}}{=} \min_{\prec}([s]_C), \\ l_{\downharpoonright C} &\stackrel{\text{def}}{=} s_{\downharpoonright C} \bowtie t_{\downharpoonright C}, \text{ pour } l = s \bowtie t, \text{ et} \\ D_{\downharpoonright C} &\stackrel{\text{def}}{=} \{l_{\downharpoonright C} \mid l \in D\}. \end{aligned}$$

— Dans $\mathbb{E}_0$ la *forme normale* de C, notée $C_\downarrow$, est telle que chaque littéral de $C_\downarrow$ est unique et :

$$C_\downarrow \stackrel{\text{def}}{=} \bigvee_{a \in \Sigma_0, a \neq a_{\downharpoonright C}} a \not\simeq a_{\downharpoonright C} \vee \bigvee_{a \simeq b \in C} a_{\downharpoonright C} \simeq b_{\downharpoonright C}$$

— Dans $\mathbb{E}_1$ les conditions suivantes décrivent une clause en forme normale :

1. chaque littéral négatif l de C est tel que $l_{\downharpoonright C \setminus l} = l$;
2. chaque littéral $t \simeq s \in C$ est tel que $t = t_{\downharpoonright C}$ et $s = s_{\downharpoonright C}$;
3. il n'existe pas deux littéraux positifs distincts l, m dans C tels que $m_{\downharpoonright l^c \vee C^-}$ est une tautologie ;
4. C ne contient aucun littéral de la forme $t \not\simeq t$ ou $t \simeq t$;
5. les littéraux de C sont distinct deux à deux.

Ce sont les points 1 et 3 qui contraignent plus cette définition que celle dans $\mathbb{E}_0$.

La preuve de l'unicité de la forme normale des clauses équivalentes est fournie dans $\mathbb{E}_0$ et $\mathbb{E}_1$. Grâce à celle-ci, le problème de la multiplicité des représentations des clauses équivalentes est résolu.

Dans $\mathbb{E}_0$, nous avons également défini des méthodes permettant de déterminer de façon syntaxique qu'une clause en subsume une autre :

— la E-*subsomption* : étant donné deux clauses C et D, la clause D E-subsume C, noté $D \leq_{\text{E}} C$, si et seulement si les deux conditions suivantes sont vérifiées,
  — $\equiv_D \subseteq \equiv_C$,
  — pour chaque littéral positif $l \in D$, il existe un littéral $l' \in C$ tel que $l \equiv_C l'$.

  Si S et S' sont des ensembles de clauses, $S \leq_{\text{E}} C$ est vrai si $\exists D \in S$, tels que $D \leq_{\text{E}} C$ et $S \leq_{\text{E}} S'$ est vrai si $\forall C \in S', S \leq_{\text{E}} C$.

— la I-*subsomption* : étant donné deux clauses C et D, la clause D I-subsume C, noté $D \leq_{\text{I}} C$, si et seulement si les conditions suivantes sont vérifiées,
  — $\equiv_D \subseteq \equiv_C$,
  — il existe une fonction *injective* γ de D^+ à C^+ telle que pour chaque littéral $l \in D^+$, $l \equiv_C \gamma(l)$.

  Cette notion est étendue aux ensembles de clauses de la même façon que pour la E-subsomption.

Cette deuxième définition est plus contraignante. Elle impose la projection des littéraux de $D_{\downharpoonright C}$ sur des littéraux distincts de $C_{\downharpoonright C}$. La relation entre ces notions de subsomption et celle de conséquence logique est la suivante :

$$\text{si } C \leq_{\text{I}} D \text{ alors } C \leq_{\text{E}} D \text{ et } C \leq_{\text{E}} D \text{ est équivalent à } C \models D.$$

Les deux notions de subsomption sont adaptées à $\mathbb{E}_1$ de façon à préserver les propriétés décrites précédemment. Grâce à ces définitions, il est possible de tester de façon syntaxique la redondance des clauses dans $\mathbb{E}_0$ et $\mathbb{E}_1$, comme c'est le cas en logique propositionnelle grâce à l'équivalence de la redondance avec l'inclusion.

Chapitre 2 - Le calcul de $\mathcal{K}$-paramodulation et ses variantes. Ce chapitre présente un calcul inspiré du calcul de paramodulation et deux variantes de celui-ci dont le principe est de simplifier la formule initiale en réduisant le nombre de constantes qu'elle comporte en utilisant ses impliqués atomiques (clauses unitaires positives).

Les règles de la $\mathcal{K}$-paramodulation, calcul défini uniquement dans $\mathbb{E}_0$, sont les suivantes :

$$\text{Paramodulation } (\mathtt{P}^+) : \quad \frac{a \simeq b \vee C \qquad a' \simeq c \vee D}{a \not\simeq a' \vee b \simeq c \vee C \vee D}$$

$$\text{Factorisation } (\mathtt{F}) : \quad \frac{a \simeq b \vee a' \simeq b' \vee C}{a \simeq b \vee a \not\simeq a' \vee b \not\simeq b' \vee C}$$

$$\text{Multi-Paramodulation Négative } (\mathtt{M}) : \quad \frac{\bigvee_{i=1}^{n}(a_i \not\simeq b_i) \vee C \qquad c \simeq d \vee D}{\bigvee_{i=1}^{n}(a_i \not\simeq c \vee d \not\simeq b_i) \vee C \vee D}$$

Les clauses générées sont systématiquement normalisées. La règle $\mathtt{P}^+$ est similaire à la règle de paramodulation standard, à la seule différence qu'au lieu de ne l'appliquer que lorsque a et a' sont identiques, cette règle est appliquée même lorsque a et a' sont différents et le littéral $a \not\simeq a'$ est ajouté à la clause produite, ce qui contraint les constantes a et a' à être sémantiquement équivalentes dans la clause générée. De la même manière, la règle $\mathtt{F}$ factorise les littéraux $a \simeq b$ et $a' \simeq b'$ en assurant l'égalité sémantique entre a et a', et entre b et b'. La règle $\mathtt{M}$ est construite sur le même principe que la règle $\mathtt{P}^+$, mais autorise la paramodulation simultanée dans différentes inéquations. Cette règle est définie de la sorte pour préserver la complétude du calcul lorsqu'il est combiné à la délétion systématique des clauses redondantes. Ce calcul est prouvé complet pour la génération d'impliqués.

Les résultats expérimentaux présentés dans le chapitre 10 suggèrent qu'un des points faibles de la $\mathcal{K}$-paramodulation est la manipulation des impliqués atomiques, i.e. qui sont des clauses unitaires positives. Une façon efficace de gérer ces impliqués est de remplacer systématiquement l'une des deux constantes par l'autre dans l'ensemble des clauses générées. Les deux variantes du calcul de $\mathcal{K}$-paramodulation présentées sont basées sur ce principe.

La première variantes décompose le problème en trois étapes :

1. génération de l'ensemble des impliqués atomiques de la formule initiale à l'aide du calcul de paramodulation non-ordonné et réécriture de la formule ;
2. génération des impliqués premiers de la formule réécrite par le calcul de $\mathcal{K}$-paramodulation ;
3. reconstruction des impliqués premiers de la formule initiale à l'aide de ceux de la formule réécrite et des impliqués atomiques.

La paramodulation non-ordonnée utilisée dans le premier point contient les règles suivantes :

$$\text{Résolution } (\texttt{R}) : \quad \frac{u \simeq v \vee C \qquad u \not\simeq v \vee D}{C \vee D},$$

avec C une clause positive et $u \not\simeq v = sel(u \not\simeq v \vee D)$;

$$\text{Paramodulation Positive } (\texttt{P}) : \quad \frac{u \simeq v \vee C \qquad u \simeq v' \vee D}{v \simeq v' \vee C \vee D},$$

avec C et D deux clauses positives.

La preuve de la complétude de ce calcul pour la génération des impliqués atomiques d'une formule équationnelle est donnée dans le mémoire. Une fois les impliqués atomiques calculés, la formule initiale est simplifiée grâce à la réécriture des constantes apparaissant dans ceux-ci. Le calcul de $\mathcal{K}$-paramodulation est ensuite utilisé pour générer les impliqués premiers de la nouvelle formule (deuxième point). Finalement, une restriction du calcul de $\mathcal{K}$-paramodulation présentée ci-dessous est utilisée pour reconstruire les impliqués premiers de la formule initiale à l'aide des impliqués atomiques et des impliqués premiers de la formule simplifiée.

$$\text{Paramodulation } (\texttt{P}^+) : \quad \frac{a' \simeq c \vee C \qquad a \simeq b}{a \not\simeq a' \vee b \simeq c \vee D}$$

$$\text{Multi-Paramodulation Négative } (\texttt{M}) : \quad \frac{\bigvee_{i=1}^{n}(a_i \not\simeq b_i) \vee C \qquad a \simeq b}{\bigvee_{i=1}^{n}(a_i \not\simeq a \vee b \not\simeq b_i) \vee C}$$

L'intérêt de cette restriction est qu'elle autorise la génération d'impliqués uniquement lorsque ceux-ci ont pour parent un impliqué atomique. La complétude de cette méthode est démontrée, ce qui signifie que cette restriction est suffisante pour reconstruire l'ensemble des impliqués premiers de la formule initiale en partant de la deuxième étape.

La deuxième variante de la $\mathcal{K}$-paramodulation est basée sur le même principe que la variante précédente, mais les deux premiers points de celle-ci sont combinés. Ainsi la $\mathcal{K}$-paramodulation est directement utilisée sur la formule initiale pour générer l'ensemble des impliqués premiers mais dès qu'un impliqué atomique est généré, celui-ci est utilisé pour réécrire l'intégralité des clauses déjà générées. L'application naïve de cette technique de réécriture ne permet pas de

garantir sa complétude. Le problème vient de l'interférence entre la réécriture et la procédure de saturation. Certaines clauses déjà utilisées doivent en effet l'être à nouveau après réécriture pour garantir que toutes les conséquences sont bien générées. Afin d'identifier un sur-ensemble de ces clauses, un critère de collision syntaxique entre réécriture et saturation est défini. De cette façon, les clauses nécessitant un nouvel examen après réécriture sont bien réexaminées, assurant ainsi la complétude pour la génération d'impliqués de cette méthode.

Chapitre 3 - Calcul de superposition contraint. Dans ce chapitre, le problème de la génération d'impliqués premiers est abordé différemment. Un nouveau calcul est présenté, le calcul de superposition contraint, ou c$\mathcal{SP}$. Ce calcul manipule des clauses contraintes, notées $[C \,|\, \mathcal{X}]$ avec C une clause et $\mathcal{X}$ la contrainte associée, qui est une conjonction de littéraux. Une clause contrainte $[C \,|\, \mathcal{X}]$ est sémantiquement équivalente à la clause $C \vee \neg\mathcal{X}$. Le calcul c$\mathcal{SP}$ est constitué des règles suivantes :

$$\text{Superposition :} \quad \frac{[l \simeq r \vee C \,|\, \mathcal{X}] \quad [l \bowtie u \vee D \,|\, \mathcal{Y}]}{[r \bowtie u \vee C \vee D \,|\, \mathcal{X} \wedge \mathcal{Y}]} \ (1.),$$

$$\text{Factorisation :} \quad \frac{[t \simeq u \vee t \simeq v \vee C \,|\, \mathcal{X}]}{[t \simeq v \vee u \not\simeq v \vee C \,|\, \mathcal{X}]} \ (2.),$$

$$\text{Assertion Positive :} \quad \frac{[t \bowtie s \vee C \,|\, \mathcal{X}]}{[u \bowtie s \vee C \,|\, \mathcal{X} \wedge t \simeq u]} \ (3.),$$

$$\text{Assertion Négative :} \quad \frac{[t \simeq s \vee C \,|\, \mathcal{X}]}{[s \not\simeq u \vee C \,|\, \mathcal{X} \wedge t \not\simeq u]} \ (4.),$$

accomagnées des conditions suivantes :

1. $l \succ r$, $l \succ u$, et $(l \simeq r)$ et $(l \bowtie u)$ sont sélectionnés dans $(l \simeq r \vee C)$ et $(l \bowtie u \vee D)$ respectivement,
2. $t \succ u$, $t \succ v$ et $(t \simeq u)$ est sélectionné dans $t \simeq u \vee t \simeq v \vee C$,
3. $t \succ s$, $t \succ u$ et $t \bowtie s$ est sélectionné dans $t \bowtie s \vee C$,
4. $t \succ u$, $t \succ s$ et $t \simeq s$ est sélectionné dans $t \simeq s \vee C$.

Comme dans le cas de la $\mathcal{K}$-paramodulation, l'assertion d'égalités est autorisée, mais cette fois ces égalités sont associées sous formes de contraintes aux clauses générées, comme indiqué dans les règles d'assertion. Les règles classiques du calcul de superposition sont étendues aux clauses contraintes de façon intuitive. Ce calcul est prouvé complet pour la génération d'impliqués.

D'un point de vue théorique, cette approche risque de fortement augmenter l'espace de recherche car une clause de taille n est équivalente à 2^n clauses contraintes différentes. Cependant, ce calcul présente aussi les avantages suivants :

— toutes les contraintes d'ordre usuelles et les stratégies de sélection du calcul de superposition peuvent lui être appliqué, ce qui n'est pas le cas pour la $\mathcal{K}$-paramodulation,

— d'autre part, cette approche offre un contrôle fin du type d'impliqués engendrés, que ce soit en terme du nombre maximal de littéraux autorisés dans les contraintes ou en ce qui concerne la nature de ces littéraux, des termes qu'ils contiennent ou encore de leur profondeur.

Cette méthode présente aussi l'avantage d'être trivialement extensible de $\mathbb{E}_0$ à $\mathbb{E}_1$, où ses règles sont les suivantes :

$$\text{Superposition :} \quad \frac{[r \simeq l \vee C \,|\, \mathcal{X}] \quad [u \bowtie v \vee D \,|\, \mathcal{Y}]}{[u[l] \bowtie v \vee C \vee D \,|\, \mathcal{X} \wedge \mathcal{Y}]} \text{ (1.)},$$

$$\text{Factorisation :} \quad \frac{[t \simeq u \vee t \simeq v \vee C \,|\, \mathcal{X}]}{[t \simeq v \vee u \not\simeq v \vee C \,|\, \mathcal{X}]} \text{ (2.)},$$

$$\text{Assertion Positive :} \quad \frac{[u \bowtie v \vee C \,|\, \mathcal{X}]}{[u[s] \bowtie v \vee C \,|\, \mathcal{X} \wedge t \simeq s]} \text{ (3.)},$$

$$\text{Assertion Négative :} \quad \frac{[t \simeq s \vee C \,|\, \mathcal{X}]}{[u[s] \bowtie v \vee C \,|\, \mathcal{X} \wedge u \bowtie v]} \text{ (4.)},$$

avec les conditions :

1. $u_{|p} = r$, $r \succ l$, $u \succ v$, et $(r \simeq l)$ et $(u \bowtie v)$ sont sélectionnés dans $(r \simeq l \vee C)$ et $(u \bowtie v \vee D)$ respectivement,
2. $t \succ u$, $t \succ v$ et $(t \simeq u)$ est sélectionné dans $t \simeq u \vee t \simeq v \vee C$,
3. $u_{|p} = t$, $t \succ s$, $u \succ v$ et $(u \bowtie v)$ est sélectionné dans $(u \bowtie v \vee C)$,
4. $u_{|p} = t$, $t \succ s$, $u \succ v$ et $(t \simeq s)$ est sélectionné dans $(t \simeq s \vee C)$.

La correction et la complétude pour la génération d'impliqués de c$\mathcal{SP}$ sont prouvées dans $\mathbb{E}_0$ et dans $\mathbb{E}_1$.

Partie II - Détection de redondances.

Cette partie du mémoire présente une structure de données inspirée des tries de CLTMS qui permettent le stockage et la manipulation des clauses dans $\mathbb{E}_0$ et $\mathbb{E}_1$.

Chapitre 4 - Arbres de clauses. Dans $\mathbb{E}_0$ comme dans $\mathbb{E}_1$, un arbre de clauses est défini inductivement soit par $\Box$, soit par un ensemble de couples (l, T') avec l un littéral et T' un arbre de clauses. L'intuition derrière cette définition est qu'un couple (l, T') représente une arrête reliant la racine de l'arbre au sous-arbre T'. Une illustration est donnée dans l'exemple 4.5, page 92. Un arbre de clauses T tel que $(l, T') \in T$ doit de plus respecter les conditions suivantes :

— pour tout l' apparaissant dans T', $l <_\pi l'$,
— il n'existe pas d'arbres T'' tel que $T'' \neq T'$ et $(l, T'') \in T$.

L'ensemble des clauses représentées par un arbre de clauses T est noté $\mathcal{C}(T)$ et est défini inductivement par :

$$\mathcal{C}(T) = \begin{cases} \{\square\} & \text{si } T = \square, \\ \bigcup_{(l,T')\in T} \left(\bigcup_{D\in\mathcal{C}(T')} l \vee D \right) & \text{dans tous les autres cas.} \end{cases}$$

Dans $\mathbb{E}_0$ si un arbre de clauses respecte les conditions ci-dessous, alors il est normal. Étant donné un couple $(l, T') \in T$, il faut que :

— l n'est ni de la forme $a \simeq a$, ni de la forme $a \not\simeq a$,

— si $l = a \not\simeq b$ avec $a \prec b$, alors b n'apparaît pas dans T',

— T' est un arbre de clauses normal.

Dans toute la suite de ce résumé, les arbres de clauses sont implicitement normaux.

Il y a trois opérations fondamentales en ce qui concerne la manipulation d'arbres de clauses. La première consiste à tester si une clause donnée est redondante par rapport à l'une des clauses contenue dans un arbre. La deuxième opération est la coupe des branches d'un arbre représentant des clauses redondantes par rapport à une clause donnée. La troisième opération est l'insertion d'une nouvelle clause dans un arbre de clauses. Des algorithmes effectuant les deux premières opérations sur des tries en logique propositionnelle extrais de la méthode CLTMS sont décrit page 94 (cf. algorithme 2 et 3).

Chapitre 5 - Subsomption par un arbre de clauses. Ce chapitre présente les différents algorithmes capables de détecter si une clause est subsumée par l'une des clauses stockées dans un arbre. Quelle que soit la logique dans laquelle on se place ($\mathbb{E}_0$ ou $\mathbb{E}_1$), le critère de redondance utilisé (E-subsomption ou I-subsomption) et le type de clauses manipulées (contraintes ou standard), le même principe est appliqué. Pour tester si une clause donnée est subsumée par une clause de $\mathcal{C}(T)$, l'algorithme effectue un parcours en profondeur de T et tente de projeter chaque littéral rencontré sur C. Si la projection échoue, l'exploration du sous-arbre correspondant est inutile, aussi l'algorithme passe directement au sous-arbre suivant. Dès qu'une clause qui subsume C est découverte, le parcours de l'arbre s'arrête.

Cette méthode est tout d'abord appliquée à des clauses standards dans $\mathbb{E}_0$ en utilisant la I-subsomption. L'algorithme résultant est appelé isEntailed$_{i0}$ (cf. algorithme 4 page 97). La spécificité de la I-subsomption, à savoir l'injectivité entre les littéraux positifs des clauses subsumée et subsumante, y apparaît à la ligne 14. En remplaçant dans cette ligne le terme $C \setminus \{l\}$ par C, l'algorithme isEntailed$_{e0}$, basé sur la E-subsomption, est défini. En raison des réécritures dues aux projections effectuées dans les appels récursifs de l'algorithme la notion d'arbre normal y est relâchée pour permettre les occurrences multiples et désordonnées de littéraux positifs. Cet algorithme termine et est prouvé correct. Cette méthode est ensuite présentée pour des clauses standards dans $\mathbb{E}_1$ en utilisant la E-subsomption. En raison de la plus grande complexité

des opérations de détection de redondance dans $\mathbb{E}_1$, l'algorithme isEntailed$_{e1}$ diffère de isEntailed$_{e0}$ par le fait qu'il lui est nécessaire de conserver tous les littéraux de la branche en cours d'exploration. Les preuves de terminaison et de correction de isEntailed$_{e1}$ sont présentées. Les complexités dans le pire des cas de isEntailed$_{i0}$ et isEntailed$_{e1}$ sont estimées respectivement en $\mathcal{O}(\text{size}(\mathcal{C}(T)) + |C| \times |\mathcal{C}(T)|)$ et $\mathcal{O}(d \times |C| \times \text{size}(\mathcal{C}(T)) + |C^-| \times |\mathcal{C}(T)|)$ avec d la profondeur maximale des termes apparaîssant dans C et T. Bien que cette estimation soit légèrement meilleure pour isEntailed$_{e1}$en raison du remplacement de $|C|$ par $|C^-|$, en pratique isEntailed$_{e1}$ est moins efficace car les opérations de comparaison et de réécriture ont un coût bien plus élevés dans $\mathbb{E}_1$.

Chapitre 6 - Coupes dans un arbre de clauses. La deuxième opération fondamentale à la manipulation des arbres de clauses est la suppression des branches de T représentant des clauses subsumées par une clause C donnée, en supposant que cette clause n'est elle-même pas subsumée par une des clauses de T. Comme pour les algorithmes précédemment décris, cette opération s'effectue à l'aide d'un parcours en profondeur de l'arbre durant lequel sont effectuées les projections nécessaires à la détection de redondances. La différence principale de ces algorithmes avec la famille d'algorithmes isEntailed est que les rôles de C et T sont inversés. À cela s'ajoute le fait que lorsqu'une redondance est détectée, au lieu de retourner 'Vrai', l'algorithme coupe les branches correspondantes et poursuit son parcours de T.

La structure de ce chapitre est semblable à celle du précédent. Tout d'abord, l'algorithme pruneEntailed$_{i0}$ est présenté, qui manipule des clauses standards de $\mathbb{E}_0$ et utilise la i-subsomption. Pour passer de la i-subsomption à la e-subsomption, il suffit de modifier le comportement de l'algorithme en ce qui concerne les littéraux positifs, pour que l'association entre ces littéraux dans les clauses considérées ne soit pas injective. Cette fois, afin d'assurer la correction de l'algorithme, ce n'est pas la notion d'arbre normal qui doit être relâchée mais celle de clause normale. Dans un deuxième temps, l'algorithme pruneEntailed$_{e1}$ est présenté qui manipule des clauses standards dans $\mathbb{E}_1$ et utilise la e-subsomption. Comme précédemment, il est nécessaire de conserver les littéraux déjà rencontrés dans la clause et dans la branche couramment explorée pour effectuer les tests de redondance dans $\mathbb{E}_1$. Les algorithmes de la famille pruneEntailed ont une complexité en $\mathcal{O}(\mathcal{C}(T))$ dans le pire des cas.

Chapitre 7 - Manipulation d'arbres de clauses contraintes. Les algorithmes considérés précédemment peuvent être étendus aux clauses contraintes. Pour cela, les adaptations suivantes sont nécessaires.

- La détection de redondances utilise la c-subsomption définie par : $[C \mid \mathcal{X}] \leq_{\text{c}} [D \mid \mathcal{Y}]$ si et seulement si $C \leq_{\text{e}} D$, $C \preceq D$ et $\mathcal{X} \subseteq \mathcal{Y}$.
- La définition des arbres de clauses est étendue aux arbres de clauses contraintes. Pour cela, on défini tout d'abord les arbres de contraintes, de la même façon que les arbres de clauses, mais avec l'interprétation $\mathcal{C}'_c(T)$, de façon à ce que les formules stockées soient bien des contraintes

et non des clauses.

$$\mathcal{C}'_c(T) = \begin{cases} \{\top\} & \text{si } T = \square, \\ \bigcup_{(l,T')\in T} \left(\bigcup_{\mathcal{X}\in\mathcal{C}'_c(T')} \mathcal{X} \wedge l \right) & \text{dans le cas contraire.} \end{cases}$$

Ces structures nous permettent de définir les arbres de clauses contraintes inductivement par un ensemble de couples littéral-arbre de clause contraint (l, T'), qui sont éventuellement concaténés à un couple $(\boxdot, T')$ avec T' un arbre de contraintes qui peut être vide. De plus, un arbre de clauses contraint T doit respecter les conditions suivantes :

— l'étiquette $\boxdot$ apparaît exactement une fois dans chaque branche de T,

— pour chaque couple $(l, T') \in T$,
  - chaque littéral l' qui apparaît dans T' est tel que $l <_\pi l'$,
  - il n'existe pas d'arbres T'' tel que $T'' \neq T'$ et $(l, T'') \in T$.

L'intuition derrière ces définitions d'arbres est que les arbres de contraintes sont accrochés aux feuilles des arbres de clauses pour former les arbres de clauses contraintes.

Les algorithmes de manipulation d'arbres sont adaptés aux arbres de clauses contraintes. En ce qui concerne isEntailed$_{cons}$ (cf. algorithme 8, page 119), l'algorithme isEntailed$_{e1}$ est utilisé pour tester la e-subsomption. Si celle-ci est confirmée lors de l'exploration d'une branche, les autres critères de c-subsomption sont testés. Le test sur l'ordre $\prec$ est réduit à une simple comparaison entre clauses standards, et le test d'inclusion des contraintes nécessite le parcours en profondeur de l'arbre à contrainte correspondant à la branche explorée. Ce dernier test est réalisé par une version simplifiée de l'algorithme isEntailed$_{e1}$. La terminaison et la correction de ces tests est prouvée.

En ce qui concerne l'opération de coupe des branches redondantes, le passage aux arbres de clauses contraintes s'effectue suivant le même principe que pour isEntailed$_{cons}$. Ainsi, pruneEntailed$_{cons}$ commence par appliquer l'algorithme pruneEntailed$_{e1}$ et dans le cas où celui-ci atteint le symbole $\boxdot$ (et n'a donc pas détecté de redondance et coupé la branche en cours d'exploration), le test sur l'ordre $\prec$ est effectué, suivi, si celui-ci non plus n'a pas détecté de redondances, par l'algorithme pruneIncluded (cf. algorithme 11, page 123) qui détecte l'inclusion des contraintes et supprime celles qui sont redondantes, suivant le même schéma que les algorithmes de la famille pruneEntailed en plus simple. Des preuves de terminaison et de correction de isEntailed$_{cons}$ et pruneEntailed$_{cons}$ sont données.

Dans ces chapitres sur la manipulation des arbres de clauses, les algorithmes sont présentés de façon abstraite. Leur implémentation concrète peux bénéficier d'améliorations techniques comme l'évaluation paresseuse de expressions. Si ces améliorations n'ont pas d'impact sur la complexité théorique des algorithmes dans le pire des cas, elles peuvent en revanche considérablement améliorer les temps d'exécution en général.

Partie III - Résultats expérimentaux.

Chapitre 8 - Détails d'implémentation. Durant cette thèse, deux prototypes de générateurs d'impliqués premiers ont été réalisés.

- `Kparam` implémente le calcul de $\mathcal{K}$-paramodulation, ses variantes et une première version de c$\mathcal{SP}$ restreinte à $\mathbb{E}_0$.
- `cSP` implémente le calcul c$\mathcal{SP}$ dans $\mathbb{E}_1$.

Le code de ces programmes ainsi que les scripts et les jeux de tests utilisés pour réaliser les expériences sont disponibles sur la page web `http://lig-membres.imag.fr/tourret/` dans l'onglet `tools`. Le format d'entrée de ces deux programmes est un sous-ensemble de la syntaxe TPTP v5.4.0.0 [75] décrit dans l'archive contenant le code.

Les détails d'implémentation notables de `Kparam` sont les suivants.

- L'ordre $\prec$ des constantes est calculé automatiquement lorsque la formule d'entrée est parsée. Il correspond à l'ordre d'apparition des constantes dans le fichier d'entrée. La représentation interne des termes dans `Kparam` est basée sur cet ordre. Chaque terme est représenté par un entier. Les noms des constantes sont conservés dans un tableau et utilisés uniquement lors de la génération du fichier contenant les résultats. Cela permet un gain de temps dans les (nombreuses) opérations qui nécessitent des comparaisons de termes.
- La normalisation des clauses est réalisée par une implémentation en Ocaml de l'algorithme *Union-Find* sur une structure de tableau persistant [14] qui permet le calcul des classes d'équivalences associées aux clauses.
- Le stockage des clauses utilise la structure d'arbre de clauses et les manipulations associées décrites dans les précédents chapitres.
- Les différentes options de `Kparam` couvrent les variantes du calcul de $\mathcal{K}$-paramodulation présentées dans le chapitre 2. En ce qui concerne la deuxième variante, deux versions du critère de collision sont proposées. La première concorde avec la définition fournie au chapitre 2 et la seconde est une sur-approximation de ce critère, effectuant des opérations moins coûteuses.
- En ce qui concerne la première version du calcul c$\mathcal{SP}$ restreinte à $\mathbb{E}_0$, différentes méthodes de sélection des clauses contraintes dans l'ensemble des clauses en attente sont disponibles ainsi qu'un index pour accélérer l'identification des clauses compatibles pour l'application des règles de calcul. De plus, deux options qui permettent de filtrer le type d'impliqués générés sont proposées. Ces filtres assurent respectivement la génération des impliqués qui ont pour conséquence l'une des clauses de la formule initiale et qui ont une taille inférieure ou égale à une valeur donnée. Cette dernière option est aussi proposée combinée avec l'index précédemment décrit.

Deux outils qui transforment des formules TPTP pour les rendre compatibles avec `Kparam` sont inclus dans l'archive. `Equationalizer` transforme les formules contenant des bouléens en formules de $\mathbb{E}_0$ et `Flattener_for_Kparam` aplatit les

termes fonctionnels des formules de $\mathbb{E}_1$ pour en faire des équivalents dans $\mathbb{E}_0$.

En ce qui concerne `cSP` les détails d'implémentation notables sont les suivants.

— La bibliothèque `LogTk` [16] est utilisée pour tout ce qui a trait à la représentation et à la manipulation des termes. Par exemple ceux-ci sont ordonnés à l'aide de l'ordre KBO [2] implémenté dans `LogTk`. De même, pour la normalisation, un algorithme de clôture de congruence inclus dans `LogTk` est utilisé. Ces changements coûteux sont nécessaires pour gérer la complexité des termes non-plat.

— Les différentes options de `cSP` sont des filtres sur la taille des impliqués générés, le nombre de littéraux négatifs qu'ils sont autorisés à contenir et la profondeur maximale de leurs termes. Comme dans la version restreinte à $\mathbb{E}_0$, un filtre qui ne conserve que les impliqués impliquant eux-mêmes une des clauses de la formule d'origine est aussi proposé. Enfin, une option permettant la comparaison des clauses générées avec les clauses initiales lorsque la limite de temps est atteinte avant la fin de l'exécution de l'algorithme est disponible. Ces différentes options peuvent être combinées les unes aux autres.

— Les termes introduits dans les contraintes par les règles d'assertion du calcul c$\mathcal{SP}$ sont en nombre fini pour assurer la terminaison de l'algorithme. Par défaut, il s'agit des termes apparaissant dans la formule initiale, mais il est aussi possible de les spécifier dans un fichier à part.

Un outil pour traiter les formules fermées de la théorie des tableaux en les convertissant en formules de $\mathbb{E}_1$, basé sur la méthode décrite dans [11] est fourni. Les tables 2 et 3, page 134 récapitulent les options disponible dans `Kparam` et `cSP` ainsi que leurs effets.

Chapitre 9 - Contexte expérimental. Contrairement à ce que pourrait laisse supposer la littérature plutôt dense sur le sujet, il n'existe à notre connaissance que très peu d'outils de génération automatique d'impliqués premiers en logique propositionnelle. À l'époque où les premières expériences ont été réalisées, nous n'en avions trouvé que deux : `ritrie` [47] and `Zres` [70]. Le second étant clairement plus performant sur nos jeux de tests, ce fut lui qui fut retenu pour servir de référence dans nos expériences. Depuis, un troisième outil (`primer` [56]) est disponible. En logique du premier ordre, les outils existant à notre connaissance sont `SOLAR` [50] et `Mistral` [22]. Ce dernier ne permettant pas la génération de l'ensemble des impliqués premiers, `SOLAR` est le seul outil qui puisse nous servir de référence, malgré le fait que la méthode sur laquelle il se base, les tableaux sémantiques [32], est notoirement inefficace quand il s'agit de gérer des formules équationnelles.

Les différents jeux de tests que nous avons utilisés sont les suivants.

Formules plates aléatoires. Ce jeu de tests contient un millier de formules de $\mathbb{E}_0$ générées aléatoirement. Chaque formule est composée de six clauses ayant entre un et cinq littéraux construits à partir de huit constantes. Des formules équivalentes en logique propositionnelle sont

aussi générées en instanciant les axiomes de l'égalité lorsque c'est nécessaire.

Formules non-plates aléatoires. Ce jeu de tests contient cent quatorze formules de taille variable et construit sur des termes de profondeurs variables.

SMT-LIB QF_AX. Ce jeu de tests est constitué d'un ensemble de formules fermées sans quantificateurs de la théorie des tableaux avec extensionalité, extrait de [1].

SMT-LIB QF_UF. Ce jeu de tests comporte huit formules extraites de la famille QF_UF de SMT-LIB [5] parmi les formules plates et satisfiables, contenant entre une centaine et un millier de clauses chacune.

Le premier jeu de tests présenté nous sert à évaluer les capacités de nos prototypes dans $\mathbb{E}_0$ et le deuxième dans $\mathbb{E}_1$. Le troisième nous sert à tester les filtres de `cSP` et le dernier à évaluer le passage à l'échelle.

Chapitre 10 - Analyse des résultats. Les trois premières expériences impliquent `Kparam` et `Zres`. Elles sont effectuées sur le premier jeu de tests, qui contient les formules plates aléatoires. La première expérience présentée compare `Kparam` avec `Zres` en utilisant la version standard du calcul de $\mathcal{K}$-paramodulation. Les résultats de cette expérience sont présentés figure 12, page 141. Le graphe de dispersion (12a) compare le nombre d'impliqués premiers générés par `Zres` en logique propositionnelle avec celui généré par $\mathcal{K}$-paramodulation dans $\mathbb{E}_0$. Le second est exponentiellement plus petit à cause des nombreuses redondances qui ne peuvent pas être détectées en logique propositionnelle en raison de l'encodage des clauses équationnelles. Le graphe suivant (12b) compare les temps d'exécution de `Kparam` et `Zres` et indique que le premier est globalement plus efficace, mais de peu. En particulier, il est observé que les tests comportant des impliqués atomiques sont beaucoup moins bien gérés par `Kparam` que par `Zres`, ce qui est la raison derrière le développement des variantes du calcul de $\mathcal{K}$-paramodulation. La deuxième expérience présentée compare les deux variantes du calcul de $\mathcal{K}$-paramodulation entre elles. Les résultats (cf. figure 13 page 142) sont sans équivoque en faveur de la seconde variante, celle qui n'utilise que le calcul de $\mathcal{K}$-paramodulation et réécrit les constantes des impliqués atomiques dès que ceux-ci sont générés. La troisième expérience compare la meilleure variante du calcul de $\mathcal{K}$-paramodulation à sa version standard et à `Zres` en mettant en évidence l'efficacité de celle-ci, qui, comme indiqué dans le tableau 4 page 143, est deux fois plus rapide que `Zres` dans 77% du jeu de tests, alors que la version standard ne l'est que pour 48% des formules.

Les expériences suivantes s'intéressent à l'implémentation du calcul c$\mathcal{SP}$ dans $\mathbb{E}_0$. Ces expériences sont aussi réalisées en utilisant le premier jeu de tests. Dans un premier temps, les différents critères de sélection des clauses contraintes dans l'ensemble des clauses en attente sont comparés. Ces résultats sont présentés dans le tableau 5, page 145, et désignent l'ordre lexicographique, qui considère d'abord la taille de la clause puis celle de la contrainte correspondante, comme le plus efficace. C'est cet ordre qui est utilisé dans les autres

expériences impliquant le calcul c$\mathcal{SP}$. Dans un deuxième temps, l'impact de l'index ajouté pour accélérer l'exécution des règles de calcul, est étudié. Comme attendu, celui-ci améliore significativement les temps d'exécution, ce qui est visible sur la figure 15, page 145. Enfin, la version avec index est comparée à la deuxième variante du calcul de $\mathcal{K}$-paramodulation. Les résultats sont présentés dans le tableau 6, page 146, et indiquent que le calcul c$\mathcal{SP}$ est globalement plus efficace que celui de $\mathcal{K}$-paramodulation (sur 68% du jeu de tests).

Le reste des expériences présentés dans ce mémoire concernent `cSP`. Tout d'abord, l'impact du changement de représentation des termes est étudié en faisant tourner `cSP` sur le premier jeu de tests. Comme l'indique la figure 16, page 146, en moyenne `cSP` est dix fois plus lent que son prédécesseur sur les formules de $\mathbb{E}_0$. Ces deux prototypes sont ensuite comparés sur le deuxième jeu de tests, celui qui contient des formules non-plates aléatoires. Cette comparaison inclut aussi `Zres` et `SOLAR` grâce à la conversion des formules d'un format à un autre. Les résultats de cette comparaison sont présentés dans le tableau 7, page 147, et mettent clairement en évidence l'intérêt de `cSP`, qui est l'outil le mieux adapté à la génération d'impliqués premiers des formules de $\mathbb{E}_1$. L'expérience suivante est réalisée sur le jeu de tests QF_UF. Elle sert à évaluer le passage à l'échelle de `cSP`. Comme le montrent les résultats (cf. tableaux 8, page 148), `cSP` n'est pas capable de manipuler des formules d'une telle taille dans des temps raisonnables, ce qui est le cas de tous les générateurs d'impliqués premier à notre connaissance. Les deux dernières expériences utilisent le jeu de tests QF_AX et le filtre de `cSP` qui permet de limiter la taille des impliqués générés. Tout d'abord, cette taille maximale est limitée à zéro, ce qui transforme `cSP` en un démonstrateur de théorèmes par réfutation. Le nombre de clauses générées pour valider ou invalider les formules est comparé au nombre de clauses générées à l'aide du démonstrateur `E` [69]. Cela met en évidence l'intérêt de la normalisation et de la méthode de détection de redondances de `cSP` puisque dans un grand nombre de cas, le nombre de clauses nécessaires à `cSP` est dix fois inférieur à celui nécessaire à `E`. Sur le même jeu de tests, la dernière expérience utilise ce même filtre, cette fois-ci en autorisant la génération d'impliqués unitaires. Le graphique 19, page 151, montre l'évolution des capacités de `cSP` avec l'augmentation de la taille de la formule. Une fois encore, les limites de `cSP` sont apparentes et la complexité du passage à l'échelle est mise en évidence.

Conclusion. Les principales contributions présentées dans ce mémoire sont les suivantes.

- Deux calculs de génération d'impliqués sont définis : la $\mathcal{K}$-paramodulation et le calcul c$\mathcal{SP}$.
- Une structure de données capable de stocker les clauses équationnelles de façon efficace et permettant la suppression des clauses redondantes est présentée.
- Différents prototypes de générateurs d'impliqués premiers basés sur les méthodes ci-dessus sont étudiés expérimentalement.

Ce travail peut être poursuivi en intégrant `cSP` à un démonstrateur mature comme `E` afin de bénéficier des améliorations techniques que celui-ci comporte. D'autre part, l'extension de la génération d'impliqués premiers à des logiques plus expressives (par exemple avec variables) est à considérer.

Contents

List of Algorithms

List of Tables

List of Figures

Aknowledgements

Thanks to the members of the jury for finding time to attend my defense on such a tight schedule and in particular to the reviewers whose insightful comments helped improve this thesis. Many thanks to Mnacho and Nicolas - I simply cannot imagine how they could have done a better job supervising me. The rest of the CAPP research team should not be forgotten. Thank you for the tips, the interesting discussions and the music. Thanks also to all the researchers that shared their work with me (and that I pestered with questions).

Thanks to my friends and family for their support. Especially many thanks to Marie-Dominique and Vincent for their friendship and their unfailing help. Finally, thanks to my husband for staying by my side all this time.

This thesis is dedicated to Alexandre and Héloïse.

Introduction

"Socrates is a human, all humans are mortal, thus Socrates is mortal." This kind of reasoning, one of the most natural to human beings, is called deduction. In formal logic, it is often used to find contradictions. For example, consider the statements "Socrates is a human", "all human are mortals" and "Socrates is a hero". If we add the statement "a hero never dies", then it becomes possible to deduce both "Socrates is mortal" and "Socrates never dies" from our set of statements, which raises a contradiction. Although this might seem counterintuitive, finding contradictions is generally used to prove that properties are true. This is done by negating the statement under consideration before adding it to the other propositions. In this way, the previous contradiction is a proof that "even heroes can die".

The problem that is of interest to us occurs when this sort of proof by contradiction has failed, i.e. when the formula representing the statement admits a model. We can get such a formula by modifying our previous example so that the last statement is "a hero never dies except when he is a sidekick" instead of "a hero never dies". In such a case, where there is no contradiction in the statements, the tools that specialize in theorem proving usually return a model of the formula. Here, for example, it can be "Socrates is a human, a mortal, a hero and a sidekick". Observe that the interesting information is only that "Socrates is a sidekick". The rest of the model was already clear from the original sentences. This sort of interesting consequence is called a prime implicate. Formally, an *implicate* of a formula S is a clause that is a logical consequence of S. It is *prime* if it is minimal w.r.t. logical entailment. In other words, prime implicates are the most general consequences of a formula.

In propositional logic, the problem of generating all the prime implicates of an input formula has been studied since the fifties. The original motivation was formula simplification (also referred to as minimizing boolean circuits). In the eighties, new applications like the truth maintenance of databases or abductive reasoning (the computation of explanations to observed facts) increased the interest toward prime implicate computation. In our previous example, if the aim is to know how to prevent Socrates from being a dead hero, then "Socrates is not a sidekick", the negation of the prime implicate "Socrates is a sidekick", is an interesting statement. It means that if it is possible to prevent Socrates from being a sidekich then it will be impossible for him to be a dead hero. Conversely, if there is a way to prove "Socrates is a sidekick" then the fact that Socrates is

a dead hero stands, no contradiction is raised. Indeed, by duality, any implicate $l_1 \vee \ldots \vee l_n$ of some formula S corresponds to a conjunction $\neg l_1 \wedge \ldots \wedge \neg l_n$ that logically entails $\neg S$.

Nowadays, there exist a lot of different methods to compute prime implicates. In propositional logic, these methods can be sorted into two categories, those based on resolution [4], a calculus that generates consequences of a formula, and those that one way or another use decomposition to solve the problem. Although a lot of directions were (and still are) explored in propositional logic, in more expressive logics like modal or first order logic, to the best of our knowledge, there are only few available methods. They are presented along with the prominent propositional algorithms in Chapter i. This scarcity is particularly true in the domain of equational reasoning where no efficient method to compute prime implicates exists. The generation of prime implicates in equational logic is the problem that was tackled during my PhD. thesis. Equational logic is the logic in which the axioms of equality are implicit. These axioms are [1]:

— reflexivity ($x \simeq x$),
— commutativity ($x \simeq y$ is equivalent to $y \simeq x$),
— transitivity (if $x \simeq y$ and $y \simeq z$ then $x \simeq z$),
— substitutivity (if $x \simeq y$ then $f(x) \simeq f(y)$).

Let us consider the following exchange between Achilles and the tortoise, that I am borrowing from Douglas Hofstadter [35] to illustrate the deduction mechanism in equational logic.

Achilles:	If I am the hero then you are the sidekick.
Tortoise:	Really? I would rather say that either you or Zeno is the sidekick!

This short discussion can be modeled in equational logic. There are five characters: Achilles (a), the tortoise (t), Zeno of Elea (z), the hero (h) and the sidekick (s). Achilles' sentence amounts to the formula $a \simeq h \Rightarrow t \simeq s$, which in clausal form gives $a \not\simeq h \vee t \simeq s$. The tortoise's answer has the logical equivalent $a \simeq s \vee z \simeq s$. We can also add to this dialog information considered as common knowledge, such as the fact that the hero and the sidekick cannot be the same person ($h \not\simeq s$) and that Achilles is the hero ($a \simeq h$). Thus we obtain the formula:

$$
\begin{aligned}
&a \not\simeq h \vee t \simeq s \\
&a \simeq s \vee z \simeq s \\
&h \not\simeq s \\
&a \simeq h
\end{aligned}
$$

An obvious implicate of this formula is $t \simeq s$. Since Achilles is the hero, the tortoise is indeed its sidekick. A less obvious consequence is that Zeno is the

1. Note that we use the symbol '$\simeq$' to denote semantic equality, while '$=$' is used for syntactic equality, in other words '$\simeq$' relates notions while '$=$' relates symbols with each other.

tortoise ($t \simeq z$). It requires all four clauses of the formula plus the transitivity axiom to be deduced.

Our interest toward the generation of prime implicates in equational logic stems from a method to extract ground abducible implicants (i.e. explanations) of first-order formulæ presented in [23] which was motivated by some applications in program verification. The method works by using a specifically tailored superposition-based calculus [54] which is capable of generating, from a given set of first-order clauses S with equality, a set of *ground* (i.e., with no variables) and *flat* (i.e., with no function symbols) clauses S' such that all abducible prime implicates of S are implicates of S'. If the formula at hand is satisfiable, these implicates can be seen as missing hypotheses explaining the "bad behavior" of the program (if the formula is unsatisfiable then the program is of course error-free). In [23], the running example is that of the successive insertion of two elements b and c in an array, at positions i and j respectively. The desired property of this insertion is that the order in which the elements are inserted must not impact the final result. In the theory of arrays, this problem can be formalized as:

$$select(store(x, z, v), z) \simeq v \tag{1}$$
$$z \simeq w \vee select(store(x, z, v), w) \simeq select(x, w) \tag{2}$$
$$d_1 \simeq store(a, i, b) \tag{3}$$
$$d_2 \simeq store(d_1, j, c) \tag{4}$$
$$d_3 \simeq store(a, j, c) \tag{5}$$
$$d_4 \simeq store(d_3, i, b) \tag{6}$$
$$select(d_2, k) \simeq select(d_4, k) \tag{7}$$

where $select(a, i)$ returns the element stored in cell i of array a and $store(a, i, e)$ inserts the element e in cell i of array a and returns the resulting array. This formula is satisfiable in the theory of arrays, thus the property (represented by its negation, line 7) is not always verified. The calculus from [23] then generates the ground implicates $i \simeq j$ and $i \not\simeq j \vee b \not\simeq c$. However, the proposed calculus is not able to *explicitly* generate the implicates of S'. This task is performed by a post-processing step which consists in translating the clause set S' into a propositional formula by adding relevant instances of the equality axioms, and then using the unrestricted resolution calculus to generate the propositional implicates. In the given example, the post-processing step produces the prime implicates $i \simeq j$ and $b \not\simeq c$. These are the conditions in which the property is not verified, and a solution to make it valid is to ensure that $i \not\simeq j \vee b \simeq d$ is always true, i.e. that the elements are inserted in different cells or that they are the same. This approach is sound, complete and terminating, but it is also very inefficient, in particular due to the fact that a given clause may have several (in general, exponentially many) representatives, that are all equivalent modulo the usual properties of the equality predicate. Computing and storing such a huge set of clauses is time-consuming and of no practical use. The present work addresses this issue. It offers an efficient method for handling the post-processing

1			
		2	
	1		
2		1	

Figure 1 – A mini-sudoku puzzle

step without translation of the problem into propositional logic and avoids the multiple representatives of equivalent clauses, while providing a powerful redundancy detection mechanism. This work also goes beyond this issue with an extension of the presented method for prime implicate generation to non-flat formulæ. The following example is a simple illustration of the kind of abductive problems that can be solved using the present work.

Example .1 (Mini-sudoku) Let us consider a simplified version of the sudoku puzzle where there are only 16 cells to fill with 4 numbers and the usual constraints on lines, columns and sub-squares (of size 4) of the grid, as illustrated with the puzzle from Figure 1.

This puzzle can be formalized in ground equational logic with uninterpreted functions [34] by assigning a constant to each number (here n_1, n_2, n_3 and n_4) and using a binary function `cell` to access the content of each cell by its line and column number. For example, in Figure 1 the cell $(2, 3)$ contains the value 2, which can be expressed using the literal $\texttt{cell}(n_2, n_3) \simeq n_2$.

With this representation the puzzle's constraints can be formalized in a straightforward way. There are four different types of constraints:

— cell (or co-domain) constraints: a cell contains a number ranging from 1 to 4 (e.g. for cell $(1, 1)$, $\texttt{cell}(n_1, n_1) \simeq n_1 \vee \texttt{cell}(n_1, n_1) \simeq n_2 \vee \texttt{cell}(n_1, n_1) \simeq n_3 \vee \texttt{cell}(n_1, n_1) \simeq n_4$),

— line constraints: each number appears exactly once in each line (e.g. for line 1, $\texttt{cell}(n_1, n_1) \not\simeq \texttt{cell}(n_1, n_2) \wedge \cdots \wedge \texttt{cell}(n_1, n_3) \not\simeq \texttt{cell}(n_1, n_4)$),

— column constraints: each number appears exactly once in each column (e.g. for column 1, $\texttt{cell}(n_1, n_1) \not\simeq \texttt{cell}(n_2, n_1) \wedge \cdots \wedge \texttt{cell}(n_3, n_1) \not\simeq \texttt{cell}(n_4, n_1)$),

— sub-square constraints: each number appears exactly once in each sub-square (e.g. for the upper-left sub-square, $\texttt{cell}(n_1, n_1) \not\simeq \texttt{cell}(n_1, n_2) \wedge \cdots \wedge \texttt{cell}(n_2, n_1) \not\simeq \texttt{cell}(n_2, n_2)$).

Each puzzle is then described by the known values of the original grid. The example (Figure 1) corresponds to the formula $\texttt{cell}(n_1, n_1) \simeq n_1 \wedge \texttt{cell}(n_4, n_1) \simeq n_2 \wedge \texttt{cell}(n_2, n_3) \simeq n_1 \wedge \texttt{cell}(n_2, n_3) \simeq n_2 \wedge \texttt{cell}(n_4, n_3) \simeq n_1$. The TPTP formalization of the mini-sudoku problem is given in Appendix A.

Using this formalism makes it possible to use tools like equational theorem provers or prime implicate generators to solve this kind of puzzle. Feeding the axiomatization of the constraints plus the description of the given puzzle into a theorem prover allows to determine if there exists a solution to the puzzle. Since

1	2	4	3
3	4	2	1
4	1	3	2
2	3	1	4

1	2	3	4
4	3	2	1
3	1	4	2
2	4	1	3

Figure 2 – Solutions of the puzzle

most solvers provide a model when the input is satisfiable, adding the predicate $\texttt{grid}(\texttt{cell}(n_1, n_1), \texttt{cell}(n_1, n_2), \ldots, \texttt{cell}(n_4, n_4))$, i.e. the representation of the whole grid, to the formula allows the recovery of the puzzle's solution where the `cell` function calls have been replaced by appropriate values in the `grid` predicate.

With a prime implicate generator, even without the `grid` predicate, it is possible to recover all the clauses assigning a value to a cell (i.e. of the form $\texttt{cell}(x, y) \simeq z$) that are a consequence of the input. This means that contrarily to a theorem prover, a prime implicate generator will directly detect if a puzzle has more than one solution. For sudokus, this is a desirable feature because having several ones is a major source of annoyance to the player.

For example, let us consider the grid of Figure 1. This particular puzzle admits two solutions, that are presented in Figure 2. For a theorem prover to find all solutions, it must be run three times. The first run is used to find the first model, say the one on the left of Figure 2. Then the negation of the corresponding solution, i.e. $\neg\texttt{grid}(n_1, n_2, n_4, n_3, n_3, n_4, n_2, n_1, n_4, n_1, n_3, n_2, n_2, n_3, n_1, n_4)$ is added to the input before the prover is run a second time, generating the second solution. A final run with both negated models is the only way to ensure that all solutions have been found. Note that it is then up to the user to find the differences between models and how to modify the puzzle so that one becomes unreachable. On a mini-sudoku, this seems rather easy, but on a normal sized sudoku or an extra-large one, with more than two solutions, this problem rapidly turns into a headache.

In contrast, a prime implicate generator returns in one run all possible values for each cell (e.g. for cell $(2, 2)$, $\texttt{cell}(n_2, n_2) \simeq n_3 \vee \texttt{cell}(n_2, n_2) \simeq n_4$), but also clearly identifies the dependencies between the values as in the clause $\texttt{cell}(n_2, n_2) \not\simeq n_4 \vee \texttt{cell}(n_2, n_1) \simeq n_3$, which can be read as "if the cell $(2, 2)$ contains 4 then the cell $(2, 3)$ must contain 3". ♣

The results produced during my PhD. were published in several articles [24, 25, 26, 27, 28, 29]. The present thesis covers all the results they present (and more) as indicated in Table 1.

The article [28] is a special case, in that in addition to presenting some algorithms from the chapters 5 and 6, it combines the work of [23] with the calculus of Chapter 3, thus obtaining a calculus that performs abductive reasoning (this result is not presented here).

This thesis is organized in three parts, preceded by two introductory chapters. The first introductory chapter exposes a state of the art of prime implicate

article	calculus	redundancy detection	experiments
[24]	Ch. 2 (1)	Ch. 5 (2) and Ch. 6 (2)	
[25]	Ch. 2 (1)	Ch. 5 (2) and Ch. 6 (2)	Ch. 10 (1.1)
[27]	Ch. 2 (3)	Ch. 5 (1) and Ch. 6 (1)	Ch. 10 (1.3)
[26]	Ch. 3 (2)		Ch. 10, (2.3)
[29]	Ch. 3 (3, 4)	Ch. 7	Ch. 10 (3,4)

Table 1 – Correspondence between the articles and the chapters.

generation. The second formalizes basic notions that are the background necessary to understand the work presented. The first part introduces the different calculi that we developed for the computation of implicates. Soundness and deductive-completeness proofs are provided with each calculus. Chapter 1 is a preliminary chapter introducing important notions used to describe the calculi such as the normal form of clauses or the projection, an operation that allows to syntactically detect redundancy between clauses. The two following chapters present our two calculi, the $\mathcal{K}$-paramodulation and c$\mathcal{SP}$ calculi, and their variants. The former uses transitivity rules to derive new clauses from premises. It is based on a form of "conditional paramodulation", meaning that equality conditions are not checked statically, but *asserted* by adding new disequations to the derived clause. For instance, given a clause $C[a]$ and an equation $a' \simeq b$, the $\mathcal{K}$-paramodulation calculus generates the clause $a \not\simeq a' \vee C[b]$, which can be interpreted as $a \simeq a' \Rightarrow C[b]$ (the condition $a \simeq a'$ is asserted). In contrast, in the latter, a distinction is drawn between the literals that are asserted and the standard ones – the former being attached to the clauses as constraints. For instance, given the same clause $C[a]$, the c$\mathcal{SP}$ calculus produces $C[b]$ along with the constraint $a \simeq b$. Another difference between the $\mathcal{K}$-paramodulation and c$\mathcal{SP}$ calculi is that the former is based on unordered paramodulation [4] while the second benefits from all the restrictions of the superposition calculus [4], which it extends. This second approach generates an important number of equivalent clause-constraint pairs, that have a single equivalent in the $\mathcal{K}$-paramodulation formalism. The second part of this thesis begins with Chapter 4 that presents a clause storage data structure. To do so, we extend the representation mechanism of [17] that uses a trie-based representation of propositional clause sets, in order to handle equational symbols. This extension is not straightforward since we have to encode the axioms of equality in the representation. The two subsequent chapters define algorithms that use the redundancy detection method in such a way to directly manipulate the clause storage data structure. The last chapter of this part adapts these algorithms to constrained clauses, a special kind of clause used in the c$\mathcal{SP}$ calculus. The final part of the thesis presents an experimental evaluation of our different methods. In Chapter 8, the details of our implementations are described and in Chapter 9 we present the other prime implicate generation tools that were used as reference, as well as the benchmarks on which we conducted our experiments. The results are then presented in Chapter 10.

Chapter i

State of the Art

The terminology used in this chapter is standard for propositional and equational logic. The reader who is unfamiliar with the notations and their meaning may refer to any introductory course on formal logic, e.g. [13].

1 Algorithms for prime implicate computation in propositional logic

This section contains a brief survey of existing algorithms for generating prime implicates of propositional formulæ. Here, we do not formally define every algorithm, but we provide some insights on how they work and discuss the main ideas underlying them, emphasizing their common points and differences. We use some simple but interesting examples to illustrate them. A comprehensive survey of the existing algorithms and their applications is available in [46]. First, let us introduce the notion of a prime implicate.

Definition i.1 In propositional logic, a clause C is an *implicate* of a formula S iff $S \models C$ and C is not a tautology. It is a *prime implicate* of S iff C is an implicate of S and for every implicate C' of S, if $C' \models C$ then $C \models C'$. An equivalent definition of a prime implicate is $S \models C$, C is not a tautology and $\forall l \in C$, $S \not\models C \setminus \{l\}$. $\Diamond$

Remark i.2 *There exists a notion that is dual to that of a prime implicate, namely a prime implicant. A clause C is a prime implicate of S iff $\neg C$ is a prime implicant of $\neg S$. In this way, any algorithm computing prime implicants can be used to compute prime implicates and vice versa. Thus from this point on, we will only consider prime implicates, even if some of the algorithms presented were originally designed for prime implicant computation.*

1.1 First approaches

Computing prime implicates is an NP-hard problem: a formula S is unsatisfiable iff $\square$ is a prime implicate of S and S is valid iff S has no prime implicate. This shows that (unless P=NP) the complexity of computing prime implicates is exponential. The notion of a prime implicate was first introduced in [57]. From that point on, a lot of algorithms to compute them were developed. Until the early seventies, they were all based on the *minterm* representation of the formulæ, which is the set of all the models of the formula. Each interpretation in this set is represented as a tuple of truth values 0 (for F) and 1 (for T), giving the value of every variables occurring in the formula, in a given order.

Example i.3 Using minterms, the formula $S_{CNF} = (x \vee y \vee z) \wedge (\neg x \vee y)$ would be represented by the set $\{(0,0,1);(0,1,0);(0,1,1);(1,1,0);(1,1,1)\}$ of minterms, where the variables are considered in the alphabetic order. For instance, the tuple $(0,0,1)$ represents the interpretation $\{x \mapsto \bot, y \mapsto \bot, z \mapsto \top\}$, which is indeed a model of S_{CNF}. ♣

As can be seen, this notation is very space consuming, so even though polynomial algorithms (w.r.t. the minterm representation, which is exponential w.r.t. the size of the input formula) were found to compute prime implicates (see [74]), they were never efficient enough to be used in real-life problems. A detailed listing of these early works can be found in [64].

At the beginning of the seventies, methods directly using the formulæ began to appear. They can be roughly divided into two classes: the resolution-based algorithms and the algorithms based on decomposition.

1.2 Resolution-based algorithms

The *resolution calculus* [4] is an inference method made of one rule, called the resolution rule:

Definition i.4 Let $C_1 = l \vee \alpha$ and $C_2 = l^c \vee \beta$ be two clauses [1]. The *resolution rule* is defined as:

$$\frac{C_1 \quad C_2}{\alpha \vee \beta}$$

The clause $\alpha \vee \beta$ generated (modulo associativity and commutativity of $\vee$) is called a *resolvent*. ◊

All the methods computing prime implicates based on the resolution calculus follow the same schema. The algorithm's input is a formula in CNF, the two steps below are applied recursively:

1. Produce the resolvent of two clauses.
2. Remove the clauses that are subsumed by another clause in the generated clause set.

1. Note that often in propositional logic and particularly here, clauses are seen as sets of literals. Outside of this section, clauses are defined as multisets of literals.

When the procedure leaves the resulting formula unchanged, the remaining set of formulæ is exactly the set of prime implicates of the original formula. This method was introduced by Tison in [76] and an incremental technique called IPIA inspired from it was presented in [41]. IPIA and Tison's method define an order on the literals to limit the number of redundant resolution steps, as illustrated in Figure 3 (taken from [17]). In this example, the chosen order is $x > y > z > t$. All the resolutions that can be applied on one literal are done in one go. Afterwards, even if the clauses generated allow for new resolutions on this literal, these are not carried out since it can be shown that they would be redundant. This is the case here, where the resolution number (2) on z allows for a resolution on y which is not done since y was already used at resolution number (1).

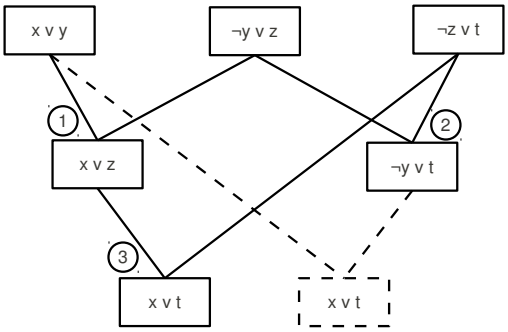


Figure 3 – Example of redundant resolution avoided by Tison's method

Tison's method was also adapted in [39] into an incremental algorithm called PIGLET (incremental version of PIG: Prime Implicate Generator), able to handle the addition of several clauses at a time. This algorithm is more efficient that IPIA because of an additional strategy in the selection of clauses for the resolutions. This strategy gives a higher priority to resolution steps between clauses that have shared literals. This way, literals are merged in the resolvent, which becomes smaller and so less likely to be subsumed later.

Another incremental algorithm inspired from IPIA is de Kleer's CLTMS (Complete Logic-based Truth Maintenance System), introduced in [17]. In addition to IPIA's optimization on the resolution step, CLTMS optimizes the subsumption step by using an efficient data structure to represent the generated prime implicates. This data structure is called a trie or discrimination tree [31]. All the literals of the formula are ordered and the clauses are considered as ordered sets of literals. The trie is then built with respect to this order. It has the following properties:

— It's edges are labelled by literals and its leaves by clauses.

— The edges below each node are ordered by the rank of the literals labeling them.
— The set of literals labeling the path from the root to a leaf is the clause labeling this leaf.

Figure 4 is an example of trie:

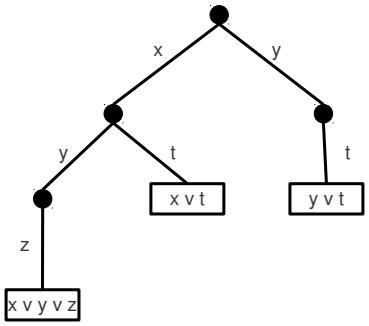


Figure 4 – Trie representation of the set of clauses $\{x \vee y \vee z; x \vee t; y \vee t\}$ with order $x > y > z > t$

CLTMS builds the trie of all prime implicates of a formula in an incremental way. The subsumption tests are performed directly on the trie which is modified in accordance with the results of the tests [2]. This method is a lot more efficient than the original IPIA algorithm. It is the first prime implicate generator fast enough to handle non-trivial problems.

The last algorithm based on Tison's method that is presented here comes from [70]. Zres-tison is an algorithm based on a compact representation of formulæ called ZBDD (Zero-suppressed Binary Decision Diagram) illustrated in Figure 5. A Zero-suppressed BDD represents a clause set. Each clause is represented by a path in the diagram from the root to the terminal node labeled by 1. On Figure 5, the dashed arrows mean that the literal labeling the parent node does not belong to the clause and the full ones that this literal belongs to the clause. Consequently, nodes for which the full arrow leads to 0 may be dismissed (whence the name "Zero-suppressed"). This algorithm is also very efficient. It uses a variant of resolution called multiresolution which may be applied directly on the ZBDD-representation. This technique makes the algorithm's complexity independent of the number of real resolution steps performed, because it allows to perform all the resolutions on the same variable at the same time. The output of Zres-tison is the set of all prime implicates presented as a ZBDD.

2. The detail of the corresponding algorithms is given Chapter 4.

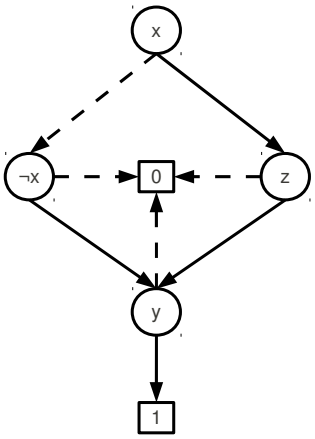


Figure 5 – ZBDD of $S_{CNF} = (x \vee y \vee z) \wedge (\neg x \vee y)$

1.3 Decomposition-based algorithms

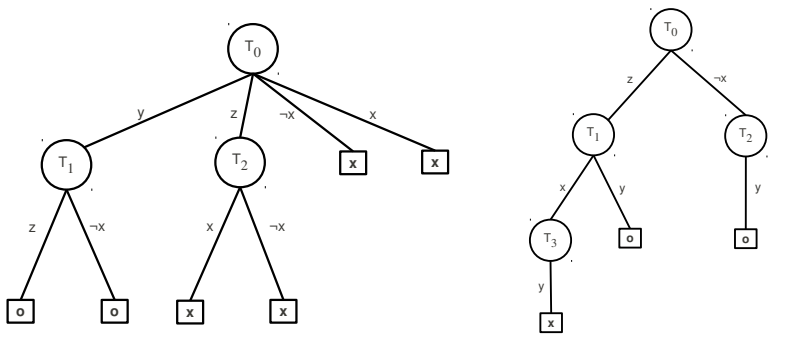
The common point of all the algorithms that do not use an inference rule is that they build the prime implicates of formulæ by recursively decomposing them into smaller formulæ (be it literals, terms [3] or clauses) and merging the results obtained on those formulæ. Contrarily to the resolution method, different algorithms accept different kinds of input formulæ, CNF, DNF or NNF. The oldest method of this kind (to the best of our knowledge) was introduced in [71] and is called Tree Method (TM). It is based on a depth-first search of an ordered semantic tree, but with a slightly different definition of a semantic tree. Here it is a tree where each edge represents a literal and each node a formula in DNF (instead of CNF like in the tries of CLTMS), a leaf being called *failure node* if the term $\top$ belongs to the set of terms it represents, and *success node* if the formula is empty. Starting from a formula in DNF, TM produces a semantic tree where all the prime implicates of the formula appear as paths from the root to a leaf, but the tree can also contain some implicates that are not prime. One of its main advantages compared to previous techniques is that it does not generate the same prime implicate more than once. An improvement of TM based on a generalization of the notion of semantic trees (called Set Enumeration trees) is presented in [67].

In [40], TM is compared with a new algorithm of the same type called MM (for Matrix Method) that also takes DNF formulæ as input. The main difference between the two algorithms is that MM does a breadth-first search of the tree. Another difference is that TM focuses on the literals (each node performs a decomposition according to the maximal literal occurring in the considered

3. In this section only, the word 'term' describe a conjunction of propositional literals.

formula) while MM focuses on the terms (each node corresponds to the maximal remaining term, and the decomposition is performed according to the literal occurring in this term). Jackson and Pais show through an experimental procedure that their new algorithm is more efficient than the previous one. The following example will highlight the differences between the two algorithms.

Example i.5 Let $S_{DNF} = (x \wedge y) \vee (\neg x \wedge y) \vee (\neg x \wedge z) \vee (y \wedge z)$. ($S_{DNF}$ is equivalent to the formula S_{CNF} previously used in Example i.3. Depending on the algorithm's requirements, we will use one or the other in all following examples.) The outputs of TM and MM for S_{DNF} are presented in Figure 6. For both graphs, success nodes are the squares containing a 'o' and failure nodes are those with a '×' inside. For TM, the partial order of the literals is based on their number of occurrences, hence in Figure (6a) we have $y > z, \neg x > x$ and we arbitrarily decide that $z > \neg x$. The algorithm's principle is for each literal to generate a new formula by removing the terms that contain it and deleting all the contrary occurences of this literal in the remaining terms. Starting from $T_0 = \{x \wedge y; \neg x \wedge y; \neg x \wedge z; y \wedge z\}$, deleting all the terms containing y allows to generate $T_1 = \{\neg x \wedge z\}$ and doing the same with terms containing z, plus removing $\neg z$ and the literals greater than z produces $T_2 = \{x; \neg x\}$. The same principle is applied to produce the terminal nodes. If the set is empty after removing terms then the node is a success node and if a term is empty after removing literals then the node is a failure node. For MM the terms themselves are ordered. In Figure (6b), the order used is $\neg x \wedge z > x \wedge y > \neg x \wedge y > y \wedge z$. The extension of paths is done term by term in this order, and subsumption checks are performed at each step to remove the implicates that are not prime (like the first path in this example). ♣



(a) TM output for S_{DNF} (b) MM output for S_{DNF}

Figure 6 – TM and MM output for S_{DNF}

All previous methods need the input formulæ to be in CNF or DNF. This

can prove to be a major inconvenience for some families of formulæ (see [52, 58]). Several techniques were developed to overcome this problem. In [52], the incremental algorithm GEN-PI (for GENeration of Prime Implicates) is introduced. It takes a conjunction of DNF formulæ as an input and generates the set of all prime implicates by closure operations defined in an order-theoretic framework. In fact, this technique stands at the boundary between decomposition-based and resolution-based methods. The closure operation can also be seen as an extension of the resolution rule from clauses to DNFs. Its creator shows that this algorithm is as efficient as CLTMS on most formulæ and a lot more powerful for those that are being difficult to express in normal form. Another method developed with the same objective is the path-dissolution approach of [59, 58]. This extension of MM to NNF formulæ nevertheless has the drawback of needing a lot of subsumption tests. Simply put, it looks for paths (representing prime implicates) through a graph representing a formula in NNF, after having simplified the graph as much as possible.

In [15], the algorithm computing the prime implicates of DNF formulæ is in fact building a BDD representing them, using a special representation of the variables of the formula called meta-product. This representation consists in replacing each variable x_i by two new variables o_i and s_i, respectively denoting whether x_i occurs in the considered term and indicating the sign of x_i (taken into account only if o_i is true). For each variable of the formula, the algorithm has to consider three recursive sub-cases:

— the case where the considered variable does not appear in the prime implicate (o_i is false);
— the case where the variable is positive in the prime implicate (o_i and s_i are both true);
— the case where the variable is negative in the prime implicate (o_i is true and s_i is false).

This encoding, combined with the BDD representation, presents the advantage that only standard BDD operations are needed to compute the prime implicates of a formula. The biggest problem here is to consider the variables in an order that makes the BDD as compact as possible. This is done through the use of heuristics, like the dynamic weight assignment method in [43]. This method orders the variables with regard to their number of occurrences in the formula.

In [43] Coudert and Madre's algorithm [15] is compared with a generally more efficient technique that uses integer linear programming (ILP). This method is introduced in [30]. Contrarily to previous algorithms, in addition to computing all prime implicates, it can also compute one prime implicate at a time and even establish preference criteria about which literals the prime implicate computed first should contain. This algorithm computes the prime implicates of DNF formulæ by converting them into ILP problems as shown in the example.

Example i.6 Let us consider S_{DNF} as defined in Example i.5. Each variable of this formula is associated with two new variables (one for the positive literal and one for the negative literal). x becomes x_1 and x_2, y becomes y_1 and y_2 and z

becomes z_1 and z_2. Then the problem is to solve $min(x_1+x_2+y_1+y_2+z_1+z_2)$ under a set of constraints C which is obtained in the following way: Each term of the formula is converted into an inequation. The literals are converted into the corresponding variables and the $\wedge$ become $+$. The sum obtained must be greater or equal to one. Additional constraints ensuring that a literal and its complement are not selected at the same time are also added, for example $x_1 + x_2 \leq 1$. In the case of S_{DNF} the constraints are:

$$C = \begin{cases} x_1 + y_1 \geq 1 \\ x_2 + y_1 \geq 1 \\ x_2 + z_1 \geq 1 \\ y_1 + z_1 \geq 1 \\ x_1 + x_2 \leq 1 \\ y_1 + y_2 \leq 1 \\ z_1 + z_2 \leq 1 \end{cases}$$

Using ILP, we can successfully recover the two prime implicates $y \vee \neg x$ and $y \vee z$ as the solutions where respectively the variables $\{x_2, y_1\}$ and $\{y_1, z_1\}$ are evaluated to 1 and all others are evaluated to 0. ♣

The advantage of the conversion to ILP is that all the techniques used in ILP solving become available. In fact, any kind of ILP algorithm can solve the problem, but to have an efficient prime implicate generator, it is better to use one specifically tuned for such a problem. For example, in [43], the search algorithm used is one that is inspired from an improvement of the DPLL algorithm called GRASP [53, 44].

Besides the ILP techniques, nearly all the most recent new algorithms for prime implicate generation use the idea of splitting the problem in three subcases for each variable that was introduced in [15]. One of them, in [33], uses a DPLL-like algorithm with three branches for each variable corresponding to these cases. In this algorithm, DNFs are used as input and BDDs are used to store the prime implicates that are found, which allows to deal with formulæ having more than 10^{20} prime implicates. Another recent algorithm based on the same principle but accepting CNF inputs comes from [47], where the notion of trie is reused in the form of pi-trie (or prime implicate trie, meaning the trie of all prime implicates of a formula). The pi-tries, like the BDDs, allow to store a great number of prime implicates at the same time. Moreover, an improvement of the prime implicate trie algorithm [48] allows it to run with a polynomial memory-space occupation. Experiments have shown that this technique is at least twice as fast as those based on resolution.

The penultimate technique presented here is based on [9]. It computes the prime implicates of a DNF formula through a new representation called quantum notation. This technique consists in annotating each literal with the identifiers of each term it belongs to as seen in the following example:

Example i.7 Let us identify the terms of S_{DNF} with numbers.

1. $x \wedge y$

2. $\neg x \wedge y$
3. $\neg x \wedge z$
4. $y \wedge z$

It gives the set of quanta : $\{x^{\{1\}}, \neg x^{\{2,3\}}, y^{\{1,2,4\}}, z^{\{3,4\}}\}$. ♣

To obtain prime implicates of the formula, it suffices to select the smallest sets of quanta that cover all terms. In the previous example, it would be $\{\neg x^{\{2,3\}}, y^{\{1,2,4\}}\}$ and $\{y^{\{1,2,4\}}, z^{\{3,4\}}\}$.

Finally, the most recent technique for the computation of prime implicates that we know of [56] relies on a decomposition similar to that of the ILP technique. It originates from the works presented in [38, 55], and transforms the input formula such that the models of the new formula are the prime implicates of the original input. In [38] the method is described for computing the prime implicants of CNF formulæ. First, an input equisatisfiable formula is created by replacing every literal with a propositional variable and ensuring that the variable and its negation are not both evaluated to true.

Example i.8 Let $P = (\neg x \vee \neg y) \wedge (x \vee \neg y) \wedge (x \vee \neg z) \wedge (\neg y \vee \neg z)$ be the input formula. The method described here generates the equisatisfiable formula $P' \cup Q$ where $P' = (l_{\neg x} \vee l_{\neg y}) \wedge (l_x \vee l_{\neg y}) \wedge (l_x \vee l_{\neg z}) \wedge (l_{\neg y} \vee l_{\neg z})$ and $Q = (\neg l_x \vee \neg l_{\neg x}) \wedge (\neg l_y \vee \neg l_{\neg y}) \wedge (\neg l_z \vee \neg l_{\neg z})$. ♣

The second step of the transformation is to concatenate the formula with encodings of the following statement for each variable: "If the variable l_p is evaluated to true then the rest of the clause C where l_p appears does not need to be evaluated to true", i.e. the formula $l_p \Rightarrow \neg C \setminus \{l_p\}$.

Example i.9 To complete the formula $P' \cup Q$ of the previous example, it must be concatenated with $P'' = (\neg l_{\neg x} \vee \neg l_{\neg y}) \wedge (\neg l_x \vee \neg l_{\neg y}) \wedge (\neg l_x \vee \neg l_{\neg z}) \wedge (\neg l_{\neg y} \vee \neg l_{\neg z})$ because it corresponds to the formulæ:

$$\begin{aligned}
l_x &\Rightarrow \neg l_{\neg y} && (= \neg l_x \vee \neg l_{\neg y}) \\
l_x &\Rightarrow \neg l_{\neg z} && (= \neg l_x \vee \neg l_{\neg z}) \\
l_{\neg x} &\Rightarrow \neg l_{\neg y} && (= \neg l_{\neg x} \vee \neg l_{\neg y}) \\
l_{\neg y} &\Rightarrow \neg l_{\neg x} && (= \neg l_{\neg x} \vee \neg l_{\neg y}) \\
l_{\neg y} &\Rightarrow \neg l_x && (= \neg l_x \vee \neg l_{\neg y}) \\
l_{\neg y} &\Rightarrow \neg l_{\neg z} && (= \neg l_{\neg y} \vee \neg l_{\neg z}) \\
l_{\neg z} &\Rightarrow \neg l_x && (= \neg l_x \vee \neg l_{\neg z}) \\
l_{\neg z} &\Rightarrow \neg l_{\neg y} && (= \neg l_{\neg y} \vee \neg l_{\neg z})
\end{aligned}$$

One model of $P' \cup Q \cup P''$ evaluates l_x and $l_{\neg y}$ to true and the other literals to false, encoding the prime implicant $y \wedge \neg y$ of P. Another one evaluates only $l_{\neg y}$ and $l_{\neg z}$ to true, thus encoding the prime implicant $\neg y \wedge \neg z$ of P. ♣

By negating the formula, applying this method and negating again the results, it is possible to compute the prime implicates of a DNF formula.

Example i.10 Consider the DNF formula S_{DNF}. The relation $\neg S_{DNF} = P$ holds (with P defined in Example i.8), thus using the previously described method, the clauses $\neg x \vee y$ and $y \vee z$, prime implicates of S_{DNF} are computed.♣

To compute all the prime implicates of a formula, the transformed formula is fed to a SAT solver that enumerates its models. An improved version of this method [56] is implemented in the software `primer`[4] where the input formula is only required to be in NNF. Experimental results in [56] indicate that this method is currently the most efficient one available.

2 Beyond propositional logic

Several lines of work have approached the problem of prime implicate generation beyond propositional logic [8, 22, 36, 42, 45, 49, 60, 61, 62, 63, 77]. A first formalization of the notion of a prime implicate in first-order logic is introduced in [45] and several obstacles to their computation, e.g. semi-decidability, are identified. Sub-classes of first-order formulæ for which this computation is possible are presented (e.g. clauses of bounded length, bounded depth). This work does not explicitly provide a computation method for first-order prime implicates. To the best of our knowledge, in first-order logic there are only two works of practical use.

The first one is the SOL calculus [36], a prime implicate generation tool for full first-order logic implemented in the JAVA program SOLAR [50]. This calculus is based on the tableau resolution method for theorem refutation [32] as illustrated in the following example.

Example i.11 Figure 7 is a refutation tableau for the following formula: $S = \{p(X) \vee s(X), \neg s(Y), q(Z) \vee \neg p(Z), \neg p(T) \vee \neg q(T)\}$. These clauses appear as the set of all the direct children of a node. In Figure 7 all the branches are closed, i.e. they contain a literal and its negation, hence the formula is unsatisfiable.♣

To generate implicates, the SOL calculus uses a 'skip' rule to stop developing branches of the tableau without refuting them. The leaves of the skipped branches form an implicate of the original formula. A specificity of the SOL calculus is that it is possible to impose conditions on the literals that can be skipped, be it on their shape or their quantity. This allows the user to filter the kind of implicates that are computed.

Example i.12 Figure 8 presents two complete SOL tableaux for the formula $S' = \{p(X) \vee s(X), \neg s(Y), q(Z) \vee \neg p(Z)\}$ that is a subset of S from Example i.11, respectively deriving the prime implicates $p(X)$ and $q(X)$. ♣

Through a modification method [37] inspired from [12], SOLAR can also directly handle equational formulæ (without having to provide the axioms of equality in the input). This method modifies the rules of the calculus so as to simulate the

4. http://logos.ucd.ie/web/doku.php?id=primer

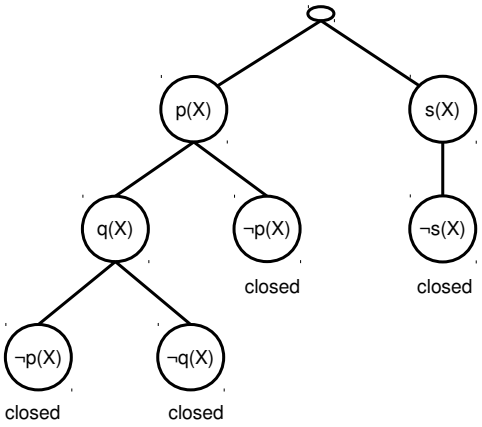


Figure 7 – A tableau refutation

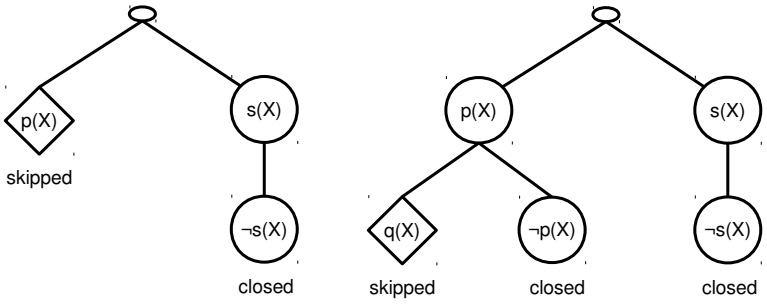


Figure 8 – Two SOL tableaux

effects of the axioms of equality on the equational literals present in the formula. It also associates to these literals ordering constraints that must be satisfied for a tableau to be closed [18].

The second method of practical use that generated prime implicates of first-order formulæ is a method incorporated into the `MISTRAL` SMT solver [5] that applies to any first-order theory admitting quantifier elimination and a decision procedure for testing the satisfiability of the formula (e.g. Presburger arithmetic)

5. `http://www.cs.utexas.edu/~tdillig/mistral/index.html`

[22]. This method computes some specific prime implicates of DNF formulæ [6], but it is not complete. These implicates are the negations of minimum partial variable assignments falsifying the formula [7].

Example i.13 We consider the arithmetic formula $S = x+y < 1 \wedge x+y+z > 3$. The assignment $x = 0 \wedge y = 1$ falsifies the formula, hence $x \neq 0 \vee y \neq 1$ is an implicate of S. This is also true of $z \neq 0$. ♣

To ensure that the generated implicates are prime the assignments must minimize a cost function.

Example i.14 Assume given the cost function c such that $\forall x, c(x) = 1$. Going back to the formula S introduced in the previous example, the implicate $x \neq 0 \vee y \neq 1$ has a cost of 2 while the cost of $z \neq 0$ is only one. Since there is no partial assignment with a cost of 0 (the formula S is satisfiable), the implicate $z \neq 0$ is prime, and so are $z \neq 1$, $z \neq 2$ and $z \neq 3$. ♣

In [22] a branch and bound algorithm is devised to compute such prime implicates and several heuristics are presented to prune the search space. Practical applications of this technique are introduced in [21, 19].

Other methods have been devised for generating prime implicates in first-order logic, based mostly either on first-order resolution [45] or the sequent calculus [49]. These methods can handle equational reasoning, the domain targeted in this paper, by adding equality axioms. Different techniques also exist, such as [42] which proposes a built-in method for handling equational reasoning based on an analysis of unification failures. These approaches are very general but not well-suited for real-world applications since termination is not ensured (even for standard decidable classes); furthermore they include no technique for removing equational redundancies, which is a major source of inefficiency. [77] uses the superposition calculus [3] to generate *positive* and *unit* prime implicates for specific theories. However, as shown in [25, 42], the superposition calculus is not complete for non-positive or non-unit prime implicates. [72, 73] propose an approach to synthesize prime implicate-like constraints ensuring that a system satisfies some invariant or safety properties. This approach relies on external provers to check satisfiability of first-order formulæ in some base theories. It is very generic and modular, however no automated method is presented to simplify the obtained constraints.

The generation of prime implicates has also been considered in the modal logic $\mathcal{K}$ [10]. This logic is the simplest modal logic there is. It differs from propositional logic only by its use of the modal operators $\Box$ and $\Diamond$, and their corresponding possible world interpretations. In [8], different definitions of the prime implicates of a modal formula are discussed and for the two best options, an algorithm is proposed. This algorithm uses a decomposition method inspired from propositional logic to compute all the prime implicates of the formula at the same time. This work is extended in [61] into an incremental algorithm.

6. This method is originally presented to compute the prime implicants of CNF formulæ.
7. In the original method, the computed implicant are minimum partial assignments satisfying the formula.

3 Summary

We have seen different algorithms computing the prime implicates of formulæ in propositional logic (PL), first-order logic (FOL) and modal logic (ML).

The methods in PL can be grouped in two families: the resolution-based methods and the decomposition-based methods. One of the strong points of the decomposition methods is that they can be applied to all kinds of formulæ (not only to CNFs as the resolution-based methods). On the other hand, a decomposition is possible only when the number of propositional variables is finite, which impairs greatly the possibilities of extending these methods to first order logic (FOL). However, extending resolution-based methods to FOL is not an easy task either. Indeed, some properties are lost when going from PL to FOL such as the finiteness of the set of all prime implicates and its equivalence with the original formula. This loss can jeopardize criteria that are critical to the smooth running of the algorithms (like termination). Thus, such an extension must be done very carefully and restrictive measures must be taken to ensure the termination of the algorithms. It is comparatively easier to extend the prime implicate generation algorithms to ML since the decomposition methods from PL can be adapted to this framework.

The work presented in this thesis builds on the propositional algorithm CLTMS [17] to create the first practical prime implicate generation method targeting specifically ground equational logic, a subset of first-order logic. As in De Kleer's algorithm, the input must be in CNF, and clauses are stored in trie-like data structures. Different variants of the paramodulation calculus are proposed as replacements for the resolution calculus used in CLTMS.

Chapter ii

Generic Context - Ground First-Order Logic with Equality

1 Ground equational logic, with and without uninterpreted function symbols

In this thesis, we work with two different logics. The first one, denoted by $\mathbb{E}_0$ represents ground flat first-order logic with equality, meaning that the formulas under consideration contain no variables (they are ground) or function symbols other than constants (they are flat), and admit a unique predicate, the equality predicate. The second logic is denoted by $\mathbb{E}_1$. It is the extension of $\mathbb{E}_0$ to uninterpreted functions. We quickly review standard definitions, more details can be found in [6]

For $n \geq 0$, Σ_n denotes a *signature* of function symbols of arity n, usually denoted by f, g. Σ_0 is the signature of constant symbols, usually denoted by a,b,c, and we let $\Sigma = \bigcup_{n=0}^{\infty} \Sigma_n$. The notation $\mathfrak{T}(\Sigma)$ stands for the set of well-formed ground *terms* over Σ, defined as usual, and most often denoted by s, t, u, v, w. The terms occurring in formulas of $\mathbb{E}_0$ are all elements of $\mathfrak{T}(\Sigma_0) = \Sigma_0$. We never use the former notation and when in $\mathbb{E}_0$, we assimilate the symbols of Σ_0 to the constant terms. For any term $s \in \mathfrak{T}(\Sigma)$, $\mathrm{Pos}(s)$ is the set of all *positions* in s, for example $\mathrm{Pos}(f(a, g(b))) = \{\epsilon, 1, 2, 2.1\}$. The expression $\mathrm{head}(s)$ represents the symbol of Σ appearing in s at position ϵ. Generally, the subterm occurring at position p in a term t is represented by $t_{|p}$. We assume that an ordering $\prec$ on terms is given.

An *atom* is an expression of the form $s \simeq t$, where $s, t \in \mathfrak{T}(\Sigma)$. Atoms are considered modulo commutativity of $\simeq$, i.e. $s \simeq t$ and $t \simeq s$ are viewed as identical. A *literal*, usually denoted by l or m, is either an atom $s \simeq t$

(*positive literal*) or the negation of an atom $s \not\simeq t$ (*negative literal*). A literal l will sometimes be written $s \bowtie t$, where the symbol $\bowtie$ stands for $\simeq$ or $\not\simeq$. The literal l^c denotes the complement of l, i.e., $s \not\simeq t$ (resp. $s \simeq t$) when l is $s \simeq t$ (resp. $s \not\simeq t$). In general literals are always represented with the greater term on the left-hand side, unless when the contrary is obvious, explicitly stated or when the literal is defined from another one using operations that do not guarantee the preservation of the ordering. A literal of the form $s \not\simeq s$ is called a *contradictory literal* (or a *contradiction*) and a literal of the form $s \simeq s$ is a *tautological literal* (or a *tautology*).

A *clause* is a finite multiset of literals, usually written as a disjunction. As usual $\Box$ denotes the empty clause. A clause is *unit* if the underlying set of literals is a singleton and *atomic* if it contains a single positive literal. For every clause C, we define $C^+ \stackrel{\text{def}}{=} \{l \in C \mid l \text{ is positive}\}$ and $C^- \stackrel{\text{def}}{=} \{l \in C \mid l \text{ is negative}\}$. A clause C such that $C = C^+$ is *positive* and it is *negative* when $C = C^-$. In addition, $C \backslash l \stackrel{\text{def}}{=} \{m \in C \mid m \neq l\}$ and $|C|$ is the length of the clause, i.e. the number of literals it contains. If $C = \bigvee_{i=1}^{n} l_i$ then $\neg C \stackrel{\text{def}}{=} \bigwedge_{i=1}^{n} l_i^c$. We often identify sets of unit clauses with conjunctions, e.g., considering a set of clauses S, we write $S \cup \bigwedge_{i=1}^{n} l_i$ for $S \cup \{l_i \mid i \in [1,n]\}$, and instead of $\{l_1, \ldots, l_n\} \subseteq \{l'_1, \ldots, l'_m\}$, we write $\bigwedge_{i=1}^{n} l_i \subseteq \bigwedge_{i=1}^{m} l'_i$. *Formulæ* are collections of clauses, built on the terms in $\mathfrak{T}(\Sigma)$ for $\mathbb{E}_1$ and on the terms in Σ_0 for $\mathbb{E}_0$.

In $\mathbb{E}_1$, an *equational interpretation* $\mathcal{I}$ is defined as a congruence relation on $\mathfrak{T}(\Sigma)$. In other words, $\mathcal{I}$ is an equivalence relation on $\mathfrak{T}(\Sigma)$ satisfying the following property: $\forall f \in \Sigma_n, \big(\forall i \in \{1..n\}, s_i =_{\mathcal{I}} t_i \Rightarrow f(s_1,..,s_n) =_{\mathcal{I}} f(t_1,..,t_n)\big)$, where $s =_{\mathcal{I}} t$ means that the terms s and t belong to the same class in $\mathcal{I}$. In $\mathbb{E}_0$, an equational interpretation $\mathcal{I}$ is an equivalence relation on Σ_0. In both logics, a positive literal $l = s \simeq t$ is evaluated to $\top$ (true) in $\mathcal{I}$, written $\mathcal{I} \models l$, if $s =_{\mathcal{I}} t$; otherwise l is evaluated to $\bot$ (false). A negative literal $l = s \not\simeq t$ is evaluated to $\top$ in $\mathcal{I}$ if $s \neq_{\mathcal{I}} t$, and to $\bot$ otherwise. A clause C is *true* in $\mathcal{I}$ if it contains a literal l that is true in $\mathcal{I}$. A clause set S is *true* in $\mathcal{I}$ if all clauses in S are true in $\mathcal{I}$. We write $\mathcal{I} \models E$ and we say that $\mathcal{I}$ is a *model* of E if the expression (literal, clause or clause set) E is true in $\mathcal{I}$. For all expressions E, E', we write $E \models E'$ if every model of E is a model of E'. A *tautology* is a clause of which all equational interpretations are models and a *contradiction* is a clause that has no model. For instance, $f(a) \not\simeq f(b) \vee b \simeq a$ is a tautology, whereas $\Box$ and $a \not\simeq a$ are contradictions.

We now introduce the central notion of a prime implicate.

Definition ii.1 In $\mathbb{E}_0$ and in $\mathbb{E}_1$, a clause C is an *implicate* of a clause set S if $S \models C$. C is a *prime implicate* of S if, moreover, C is not a tautology, and for every clause D such that $S \models D$, we have either $D \not\models C$ or $C \models D$. $\Diamond$

Example ii.2 In $\mathbb{E}_0$, consider the clause set S:

1	$a \simeq b \vee d \simeq a$	2	$a \simeq c$
3	$c \not\simeq b$	4	$c \not\simeq e \vee d \simeq e$

The clause $d \simeq a$ is an implicate of S, since Clauses 2 and 3 together entail $a \not\simeq b$ and thus $d \simeq a$ can be inferred from $a \not\simeq b$ and Clause 1. The clause

$a \not\simeq e \vee d \simeq e$ can be deduced from 4 and 2 and thus is also an implicate. But it is not prime, since $d \simeq a \models a \not\simeq e \vee d \simeq e$ (it is clear that $d \simeq a, a \simeq e \models d \simeq e$, by transitivity) but $a \not\simeq e \vee d \simeq e \not\models d \simeq a$. ♣

Rewriting in $\mathbb{E}_0$

Let E be a literal, a clause or a formula. The notation $E[a/b]$ represents this same expression where all occurrences of the constant b have been replaced by a. Similarly, the notation $E[A]$, where A is a set of atomic clauses, denotes the expression E where every constant a is replaced by $\min_{\prec}\{a' \mid A \models a \simeq a'\}$.

Proposition ii.3 *Let C, D be two clauses and a, b be two constants. If $D \models C$ then $D[a/b] \models C[a/b]$.*

Proof. Let $\mathcal{M}$ be a model of $D[a/b]$ and $\mathcal{I}$ be the interpretation that coincides with $\mathcal{M}$ on all constants except on b which is such that $b =_{\mathcal{I}} a$. Since b does not appear in $D[a/b]$, $\mathcal{I}$ is also a model of $D[a/b]$, thus $\mathcal{I} \models a \not\simeq b \vee D[a/b]$. The clause D is logicaly equivalent to $a \not\simeq b \vee D[a/b]$ hence $\mathcal{I} \models D$, and since $D \models C$, $\mathcal{I} \models C$. Thus since $\mathcal{I} \models a \simeq b$, $\mathcal{I} \models C[a/b]$ and finally $\mathcal{M} \models C[a/b]$ because b does not appear in $C[a/b]$. ■

Proposition ii.4 *Consider the clauses C, D and the constants a, b such that $a \prec b$. Assume that $a \not\simeq b \vee C$ is a clause where b does not appear. Then $D \models a \not\simeq b \vee C$ if and only if $D[a/b] \models C$.*

Proof. Assume $D[a/b] \models C$ and let $\mathcal{I}$ be an interpretation such that $\mathcal{I} \models D$. If $a \neq_{\mathcal{I}} b$ then $\mathcal{I} \models a \not\simeq b \vee C$. Otherwise, we have $\mathcal{I} \models D[a/b]$ and by hypothesis $\mathcal{I} \models C \models a \not\simeq b \vee C$.

Assume $D \models a \not\simeq b \vee C$, let $\mathcal{I} \models D[a/b]$ and consider the interpretation $\mathcal{J}$ identical to $\mathcal{I}$, except that $a =_{\mathcal{J}} b$. Then by construction $\mathcal{J} \models D[a/b]$, hence $\mathcal{J} \models D$ and $\mathcal{J} \models a \not\simeq b \vee C$. From $\mathcal{J} \models a \simeq b$ we deduce that $\mathcal{J} \models C$. Since the constant b does not occur in C, we conclude that $\mathcal{I} \models C$. ■

Orderings

An *order* $\preceq$ is a reflexive, transitive and anti-symmetric relation. An *strict* order $\prec$ is an irreflexive, transitive and anti-symmetric relation induced by a (non-strict) order in the following way: $x \prec y$ iff $x \preceq y$ and $x \neq y$. A strict order on terms is *total* when, given any two terms s and t, either $s \prec t$, or $s = t$, or $t \prec s$.

Definition ii.5 A strict order $\prec$ on terms is *well-founded* if every subset $T \subseteq \mathfrak{T}(\Sigma)$ (or, in $\mathbb{E}_0$, $T \subseteq \Sigma_0$) has a minimal element, i.e. $\exists t \in T$ such that $\forall t' \in T, t \not\succ t'$.

An order is well-founded if the corresponding strict order is well-founded. ◊

Definition ii.6 In $\mathbb{E}_1$ a strict order $\prec$ on $\mathfrak{T}(\Sigma)$ is a *rewrite order* iff:
$\forall s_1, s_2 \in \mathfrak{T}(\Sigma), \forall n \geq 0, \forall f \in \Sigma_n, \forall i \in 1 \ldots n, \forall t_1, \ldots t_{i-1}, t_{i+1}, \ldots, t_n \in \mathfrak{T}(\Sigma)$,

if $s_1 \prec s_2$ then $f(t_1, \ldots, t_{i-1}, s_1, t_{i+1}, \ldots, t_n) \prec f(t_1, \ldots, t_{i-1}, s_2, t_{i+1}, \ldots, t_n)$

$\Diamond$

Definition ii.7 A strict order is a *reduction order* when it is a *well-founded rewrite* order. $\Diamond$

Remark ii.8 *The usual definition of a rewrite order include the need to be closed under substitution, but since $\mathbb{E}_1$ does not include variables, this property is ommited here. Note also that in $\mathbb{E}_0$ any strict order is well-founded and a rewrite order since only Σ_0 is used. Thus any strict order in $\mathbb{E}_0$ is a reduction order.*

In $\mathbb{E}_1$, an example of reduction order is the *Knuth-Bendix order* also called *KBO*.

Example ii.9 Let $\prec_{sig}$ be a strict order on Σ and $w : \Sigma \to \mathbb{R}^*$ be a weight function. The weight function is extended to terms in the following way: $\forall t \in \mathfrak{T}(\Sigma), w(t) = \sum_{\sigma \in \Sigma} w(\sigma).|t|_\sigma$, where $|t|_\sigma$ is the number of occurences of the symbol σ in t. The *Knuth-Bendix order* $\prec_{kbo}$ on $\mathfrak{T}(\Sigma)$ induced by $\prec_{sig}$ and w is defined as follows: for $s, t \in \mathfrak{T}(\Sigma)$ we have $s \prec_{kbo} t$ iff

1. $w(s) < w(t)$ or
2. $w(s) = w(t)$ and one of the following properties holds:
   (a) There exist function symbols $f, g \in \Sigma_n \times \Sigma_m$ such that $f \prec_\Sigma g$ and $s = f(s_1, \ldots, s_m)$, $t = g(t_1, \ldots, t_m)$.
   (b) There exist a function symbol $f \in \Sigma_m$ and an index i, $1 \leq i \leq m$ such that $s = f(s_1, \ldots, s_m)$, $t = f(t_1, \ldots, t_m)$ and $s_1 = t_1, \ldots, s_{i-1} = t_{i-1}$ and $s_i \prec_{kbo} t_i$.

In [2], KBO is described in the following manner: *"Thus, the Knuth-Bendix order makes a lexicographic comparison, where first the weights of the terms are considered, second their root symbols, and third recursively the collections of the immediate subterms."* ♣

All the definitions above in this paragraph about order are extracted and, when necessary, adapted from [2]. In the continuation of this thesis, the given order $\prec$ on terms is assumed to be a *total strict reduction* order.

Definition ii.10 The two following strict orders on literals are used throughout this thesis, both in $\mathbb{E}_0$ and $\mathbb{E}_1$.

1. The order $\prec$ on terms is extended in the usual way to literals and then to clauses using the multiset extension, by associating a negative literal $t \not\simeq s$ to the set $\{\{t, s\}\}$ and a positive literal $t \simeq s$ to $\{\{t\}, \{s\}\}$.

2. The total order $<_\pi$ on literals is defined as follows:
   — equations are all greater than inequations;
   — if l_1 and l_2 are literals with the same polarity then $l_1 <_\pi l_2$ iff $l_1 \prec l_2$.

Both strict orders are relaxed (into the non-strict orders $\preceq$ and $\leq_\pi$ resp.) by also accepting equal literals or clauses. ◊

Example ii.11 Let $C = g(a) \not\simeq b \vee c \simeq d$ and $D = a \not\simeq b \vee f(c) \simeq d$, with $a \prec b \prec c \prec d \prec f(c) \prec g(a)$. We have $D \prec C$ and $C <_\pi D$, because on the one hand $a \not\simeq b \prec c \simeq d \prec f(c) \simeq d \prec g(a) \not\simeq b$, and on the other hand $a \not\simeq b <_\pi g(a) \not\simeq b <_\pi c \simeq d <_\pi f(c) \simeq d$. ♣

The order $\prec$ is used to determine which implicates are prime and which are redundant. The order $<_\pi$ is necessary for the clause storage and manipulation methods presented in Part II, but is not used outside of this scope.

Proposition ii.12 *In* $\mathbb{E}_0$*, any total ordering* $\prec$ *on constants is a* reduction ordering.

Example ii.13 Using the alphabetic order as the basis,
— $b \simeq a \prec b \not\simeq a$ because $\{\{a\}, \{b\}\} \prec \{\{a, b\}\}$,
— $c \simeq b \vee c \not\simeq a \prec c \simeq a \vee c \not\simeq b$ because $\{\{b\}, \{c\}, \{a, c\}\} \prec \{\{a\}, \{c\}, \{b, c\}\}$.
♣

2 Calculi and related notions

Definition ii.14 A *calculus* is a set of inference rules. There are two kinds of rules:

$$\text{Generation rules:} \qquad \frac{C_1 \ \ldots \ C_n}{C}$$

where the clauses $C_1, \ldots, C_n$ are the *premises* of the rule (or parent clauses) and the clause C is the consequent, i.e. the newly generated clause. The rule is *correct* iff C is entailed by $\{C_1, \ldots, C_n\}$.

$$\text{Deletion rules:} \qquad \frac{S \cup \{C\}}{S}$$

where the clause C is removed from the formula $S \cup \{C\}$. When the rule is correct, we have $S \cup \{C\} \equiv S$ (or at least satisfiability should be preserved).
Both kinds of rules can have additional constraints specifying under what conditions the rule can be applied. ◊

Notation ii.15 Let $\mathcal{A}$ be a calculus. We write $S \vdash_{\mathcal{A}} C$ if the clause C can be generated from S in one inference of $\mathcal{A}$. If there is no ambiguity, we may write $S \vdash C$ instead of $S \vdash_{\mathcal{A}} C$. ◊

Definition ii.16 A calculus is *correct* when all its rules are correct. ◊

The standard paramodulation calculus operates on the full first order logic with equality [3]. Its restriction to $\mathbb{E}_0$ is defined as follows.

Definition ii.17 In $\mathbb{E}_0$, the *paramodulation calculus* consists of the following rules:

$$\text{Positive Paramodulation:} \quad \frac{a \simeq b \vee C \qquad a \simeq c \vee D}{b \simeq c \vee C \vee D}$$

$$\text{Negative Paramodulation:} \quad \frac{a \not\simeq b \vee C \qquad a \simeq c \vee D}{c \not\simeq b \vee C \vee D}$$

$$\text{Factorization:} \quad \frac{a \simeq b \vee a \simeq c \vee C}{a \simeq b \vee b \not\simeq c \vee C}$$

$$\text{Reflexion:} \quad \frac{a \not\simeq a \vee C}{C}$$

This calculus is refined into the *superposition calculus* by adding the following constraints. Given the function *sel* that returns the greatest literal in a clause, the rules are constrained by the following conditions: $b, c \prec a$, $sel(a \simeq b \vee C) = a \simeq b$, $sel(a \simeq c \vee D) = a \simeq c$ and $sel(a \simeq b \vee a \simeq c \vee C) = a \simeq b$ ◊

It is easy to verify that the paramodulation calculus is correct.

Definition ii.18 Let $\mathcal{A}$ be a calculus. The formula S is *closed* by $\mathcal{A}$ if it contains all the clauses that can be generated using rules of $\mathcal{A}$ from premises in S. ◊

Example ii.19 With $a \prec b$, the formula $S = \{b \simeq a, a \simeq a, b \not\simeq a, a \not\simeq a, \square\}$ is closed by the superposition calculus. ♣

Saturation and redundancy

Often it is more practical to get rid of the clauses that are redundant with respect to other clauses in a formula rather than keeping them. It results in a simpler equivalent formula. There are different ways to define redundancy but they must all verify the following property.

Requirement ii.20 If, according to a given definition of redundancy, a clause C is redundant with respect to a formula S (or redundant in the formula $S \cup \{C\}$) then $S \cup \{C\} \equiv S$. ◊

The standard definition of redundancy associated to the superposition calculus is the following.

Definition ii.21 A clause C is *redundant* with respect to a formula S if there exist $C_1, \ldots, C_n \in S$ such that $C_1, \ldots, C_n \models C$ and the C_i are all smaller or equal to C for a given reduction order. ◊

This definition of redundancy preserves the completeness of the superposition calculus (see [3]).

The redundancy criterion that is used for implicate generation is slightly more restrictive. To distinguish the two notions, we empoy the expression *clausal redundancy*.

Definition ii.22 A clause C is *clause-redundant* with respect to a formula S if there exists a clause $D \in S$ such that $D \models C$. ◊

Other notions of redundancy will be introduced in the upcoming chapters when needed. It will always be clear from the context which notion of redundancy is being used, thus the specific denomination introduced here will not necessarilly be used.

The deletion of redundant clauses is integrated into a calculus through the so-called redundancy elimination rule.

Definition ii.23 Let S be a formula and C be a clause. The *redundancy elimination rule* is the deletion rule:

$$\frac{S \cup \{C\}}{S} \qquad \text{if } C \text{ is redundant in } S.$$

Definition ii.24 Let S be a formula. We say that S is *free of redundancy* if the redundancy elimination rule cannot be applied to S. ◊

By combining a calculus with the redundancy elimination rule, the notion of saturation can be defined.

Definition ii.25 Let $\mathcal{A}$ be a calculus to which a notion of redundancy is associated. The formula S is *saturated up to redundancy* (or simply *saturated*) if all inferences by the rules of $\mathcal{A}$ generate clauses that are redundant with respect to S. ◊

Example ii.26 The formula S of Example ii.19 is saturated up to redundancy. The formula $S' = \{a \simeq b, a \not\simeq b\}$, equivalent to S, is not saturated up to redundancy because $S \vdash \square$ by resolution, $\square \models a \simeq b$ and $\square \models a \not\simeq b$. The equivalent formula $\{\square\}$ is saturated up to redundancy and redundancy free. ♣

Notation ii.27 Let $\mathcal{A}$ be a calculus, S be a set of clauses, and C be a clause. The set $S_{\vdash^i}$ denotes all clauses obtained from S (up to redundancy) by exactly i inferences of $\mathcal{A}$ and $S_{\vdash^{i,C}}$ is the set of all clauses obtained from $S \cup \{C\}$ by exactly i inferences of $\mathcal{A}$ such that at least one parent of each inference is C (including unary inferences). The set $S_{\vdash C}$ is defined as $\bigcup_{i=0}^{\infty} S_{\vdash^{i,C}}$. It can be viewed as the saturation up to redundancy of $S \cup \{C\}$ by a restriction of $\mathcal{A}$ to inferences with C as a parent. The saturation up to redundancy of S using the full calculus is denoted by $S_{\vdash_{\mathcal{A}}}$ or $S_{\vdash}$. ◊

Definition ii.28 Given a calculus $\mathcal{A}$ and an associated notion of redundancy, $\mathcal{A}$ is *refutationaly complete* if any formula that is saturated up to redundancy by $\mathcal{A}$ and unsatisfiable necessarily contains $\square$. ◊

Theorem ii.29 *The superposition calculus with its corresponding notion of redundancy is refutationaly complete [54].*

When a calculus is correct and refutationaly complete, to know if a formula is satisfiable or not, it is sufficient to compute its saturation up to redundancy (provided it terminates). If a formula is unsatisfiable, the empty clause is guaranteed to be generated and the elimination of redundant clauses efficiently prunes the search space.

However, this property is not enough for the generation of all prime implicates, which is our aim. Indeed, only the empty clause is guaranteed to be generated (if the formula is unsatisfiable). Thus we have to consider a stronger notion of completeness.

Definition ii.30 A calculus $\mathcal{A}$ is *deductive-complete* if for all formulas S that are saturated up to redundancy and clauses C such that $S \models C$, there exists a clause $D \in S$ such that $D \models C$. ◊

Example ii.31 The superposition calculus is not deductive-complete. For example, it is not possible to derive $a \not\simeq c \vee b \simeq d$ or a clause that entails it from the formula $\{a \simeq b, c \simeq d\}$, even though $\{a \simeq b, c \simeq d\} \models a \not\simeq c \vee b \simeq d$. ♣

Note that a formula saturated up to redundancy by a deductive-complete calculus is exactly its own set of prime implicates.

Saturation procedure

The most common procedure used to saturate a set of clauses consists in storing all the input clauses in a *waiting set* W and pick each of them one at a time to perform all possible inferences between the chosen clause and the already *processed set* of clauses P. Once this is done, the clause is stored with the other processed clauses in P and the consequents are added to the waiting set W. We present two formalizations of this procedure, one in pseudo-code and one as a set of rules.

Definition ii.32 The saturation of a formula S can be performed by applying the following rules starting from the pair of sets $\langle \emptyset ; S \rangle$ until none can be applied.

$$\text{Redundancy elimination } (R): \quad \frac{P ; W \cup \{C\}}{P ; W}$$

if C is redundant with respect to P.

$$\text{Inference generation } (I): \quad \frac{P ; W \cup \{C\}}{P' \cup \{C\} ; W \cup P'_{\vdash 1, C} \setminus \{C\}}$$

if no redundancy elimination applies on C and where

$$P' = \{D \in P \mid D \text{ is not redundant in } P \cup \{C\}\}. \qquad \lozenge$$

Note that the conditions associated with the rules are mutually exclusive and that one of them always holds. Hence all the clauses in W can be processed and the procedure is completed when $W = \emptyset$.

Example ii.33 The saturation of $\{b \simeq a, b \not\simeq a\}$ by the superposition calculus can be done in the following way, where $a \prec b$ and the clause underlined in W is the one on which a rule applies:

$$\emptyset; \{\underline{b \simeq a}, b \not\simeq a\} \vdash_I \{b \simeq a\}; \{\underline{a \simeq a}, b \not\simeq a\} \vdash_R \{b \simeq a\}; \{\underline{b \not\simeq a}\}$$
$$\vdash_I \{b \simeq a, b \not\simeq a\}; \{\underline{a \not\simeq a}\} \vdash_I \{b \simeq a, b \not\simeq a, a \not\simeq a\}; \{\underline{\square}\} \vdash_I \{\square\}; \emptyset$$

The inside evolution of the sets P and W depends on the order in which the clauses are selected. If the initial formula is satisfiable, the final result also depends on this order (since the superposition calculus is not deductive-complete). ♣

The whole procedure is synthesized into Algorithm 1. In the literature it is

Algorithm 1 Saturation of S

```
P := ∅
W := S
while W ≠ ∅ do
    Choose a clause C ∈ W
    if C is not redundant with respect to P then
        P := P ∪ {C} \ {D ∈ P | D is redundant in P ∪ {C}}
        W := (W ∪ P_{⊢1,C}) \ {C}
    else
        W := W \ {C}
    end if
end while
return P
```

known as the Given-clause algorithm [68]. Note that the rules of Definition ii.32 correspond exactly to the two cases of the algorithm's while-loop. There exist several variants of this algorithm, that differ mainly on the order in which the clauses of W are picked and on the timing in which the redundancy elimination rules are applied [65].

Part I

Calculi for Implicate Generation

This part of the thesis regroups the different calculi for prime implicate generation that were considered during the thesis. In the first chapter, some preliminary results as well as the techniques for representing equivalent clauses and detecting redundancy are introduced. Their principles are common to all the calculi, although adaptations are necessary depending on the circumstances. The second chapter's main theme is the $\mathcal{K}$-paramodulation calculus, which is a form of "conditional paramodulation" where equalities are asserted rather than deduced. $\mathcal{K}$-paramodulation is defined only for $\mathbb{E}_0$. Two improvements of this calculus are also presented that have to do with the rewriting of atomic implicates. The third and last chapter introduces a different calculus, the constrained superposition calculus, where asserted equalities are "frozen" so as to prevent further inferences on them.

Chapter 1

Entailment and Representation of Clauses Modulo Equality

In propositional logic, detecting redundant clauses is an easy task, because a clause C is a logical consequence of D iff either it is a tautology or every literal in D also occurs in C. Furthermore, the only tautologies in propositional logic are the clauses containing complementary literals, which is straightforward to test. However, in $\mathbb{E}_0$ and $\mathbb{E}_1$, these properties do not hold anymore: for example in $\mathbb{E}_0$ the clause $a \not\simeq b \vee b \simeq c$ is a logical consequence of $a \simeq c$ but obviously $a \simeq c$ is not a sub-clause of $a \not\simeq b \vee b \simeq c$. In $\mathbb{E}_1$, the clause $e \not\simeq b \vee b \not\simeq c \vee f(a) \simeq f(b)$ is redundant w.r.t. the clause $e \not\simeq c \vee a \simeq c$. Thus testing clause inclusion is no longer sufficient. Testing whether two clauses are equivalent is not straightforward either: for instance $a \not\simeq b \vee b \simeq c$ and $a \not\simeq b \vee a \simeq c$ do not share the same literals although they are equivalent. In this chapter, we devise a new redundancy criterion that generalizes subsumption. To this purpose we show in Section 1 how to *normalize* clauses according to the total ordering $\prec$ on constant symbols, and in Section 2 we introduce new notions of *subsumption* for testing redundancy.

1 Representation of clauses modulo equality and entailment in $\mathbb{E}_0$ and $\mathbb{E}_1$

Definition 1.1 Let C be a clause in $\mathbb{E}_1$, we define for any term $s \in \mathfrak{T}(\Sigma)$ the *C-equivalence class* of s as:

$$[s]_C \quad = \quad \{t \mid \neg C \models s \simeq t\}\,.$$

The corresponding equivalence relation is written $\equiv_C$. By definition, the following equivalences hold: $(s \equiv_C t) \Leftrightarrow (s \not\simeq t \models C) \Leftrightarrow (\neg C \models s \simeq t)$.

The *C-representative* of a term s, a literal l and a clause D are respectively defined by:

$$\begin{array}{rcl} s_{\downarrow C} & \stackrel{\text{def}}{=} & \min_{\prec}([s]_C), \\ l_{\downarrow C} & \stackrel{\text{def}}{=} & s_{\downarrow C} \bowtie t_{\downarrow C}, \text{ for } l = s \bowtie t, \text{ and} \\ D_{\downarrow C} & \stackrel{\text{def}}{=} & \{l_{\downarrow C} \mid l \in D\} \end{array}$$

For any expression (constant, literal, clause) E, the expression $E_{\downarrow C}$ is called the *projection of E on C*.

In $\mathbb{E}_0$, these definitions are the same, but restricted to the terms of Σ_0. $\Diamond$

Note that every term s has a representative, since it is clear that $\neg C \models s \simeq s$.

Proposition 1.2 *Let C be a clause in $\mathbb{E}_0$. If C is a tautology, then $\equiv_C = \Sigma_0 \times \Sigma_0$, i.e. $\forall a, b \in \Sigma_0$, $a \equiv_C b$. Otherwise $\equiv_C$ is the reflexive and transitive closure of the set $\{(a, b) \mid a \not\simeq b \in C\}$.*

Let C be a clause in $\mathbb{E}_1$. If C is a tautology, then $\equiv_C = \mathfrak{T}(\Sigma) \times \mathfrak{T}(\Sigma)$. Otherwise $\equiv_C$ is the (reflexive and transitive) congruence closure of the set $\{(s, t) \mid s \not\simeq t \in C\}$.

Proposition 1.3 *Let s be a term, l be a literal and C and D be two clauses, then:*

$$\neg C \models s \simeq s_{\downarrow C}, \quad \neg C \models l \Leftrightarrow l_{\downarrow C}, \quad \text{and } \neg C \models D \Leftrightarrow D_{\downarrow C}.$$

where $\Leftrightarrow$ is the usual logical equivalence.

Example 1.4 Let $C = a \not\simeq b \vee b \not\simeq c \vee d \not\simeq e \vee a \simeq e \in \mathbb{E}_0$. We have $\neg C \models a \simeq b$ and $\neg C \models b \simeq c$ since both $a \not\simeq b$ and $b \not\simeq c$ occur in C. By transitivity, this implies that $\neg C \models a \simeq c$, and therefore we have $a_{\downarrow C} = b_{\downarrow C} = c_{\downarrow C} = a$ (recall that constants are ordered alphabetically). Similarly, $d_{\downarrow C} = e_{\downarrow C} = d$. If f is a constant distinct from a, b, c, e, d, then $f_{\downarrow C} = f$. We have $(b \simeq e \vee a \not\simeq b)_{\downarrow C} = a \simeq d \vee a \not\simeq a$. ♣

Example 1.5 In $\mathbb{E}_1$, let $C = a \not\simeq b \vee f(c) \not\simeq d \vee f(b) \simeq f(c)$. Since $\neg C \models a \simeq b$, we have also $\neg C \models f(a) \simeq f(b)$. Moreover $\neg C \models f(c) \simeq d$. Given the order $a \prec b \prec c \prec d \prec f(a) \prec f(b) \prec f(c)$, we obtain the relations $a_{\downarrow C} = b_{\downarrow C} = a$, $f(a)_{\downarrow C} = f(b)_{\downarrow C} = f(a)$ and $f(c)_{\downarrow C} = d_{\downarrow C} = d$. Thus $(f(b) \simeq f(c))_{\downarrow C} = f(a) \simeq d$. ♣

Proposition 1.6 *The following basic properties hold:*

1. *If C and D are two non-tautological clauses containing the same negative literals then for every constant a we have $a_{\downarrow C} = a_{\downarrow D}$.*
2. *If a non-tautological clause C contains no negative literal then for any constant a, $a_{\downarrow C} = a$.*

1.1 Results specific to $\mathbb{E}_0$

The following proposition analyzes the effect on the relation $\equiv_C$ of adding a disequation $a \not\simeq b$ to C. It is clear that this addition can only affect the equivalence classes of a and b:

Proposition 1.7 *Let C be a clause and let a, b, c, d be constant symbols. If $c \equiv_{a \not\simeq b \vee C} d$ and $c \not\equiv_C d$, then $a \not\equiv_C b$ and $\{a_{\downarrow C}, b_{\downarrow C}\} = \{c_{\downarrow C}, d_{\downarrow C}\}$. Conversely, if $\{a_{\downarrow C}, b_{\downarrow C}\} = \{c_{\downarrow C}, d_{\downarrow C}\}$, then $c \equiv_{a \not\simeq b \vee C} d$.*

Proof. We begin by proving the first implication. Obviously we cannot have $a \equiv_C b$, otherwise $\equiv_C$ would be identical to $\equiv_{a \not\simeq b \vee C}$. By definition of $\equiv_{a \not\simeq b \vee C}$ and Proposition 1.2 there exist a sequence of constant symbols $e_1, \ldots, e_n$ such that $e_1 = c$, $e_n = d$ and for all $i \in [1, n-1]$, $e_i \not\simeq e_{i+1}$ occurs in $a \not\simeq b \vee C$. W.l.o.g. we assume that this sequence is minimal, which implies that the e_i's are pairwise distinct. Then there exists at most one $i \in [1, n-1]$ such that $e_i \not\simeq e_{i+1}$ is identical to $a \not\simeq b$ (up to commutativity). Notice that such an i necessarily exists otherwise we would have $c \equiv_C d$. We have $c = e_1 \equiv_C e_i$ and $e_{i+1} \equiv_C e_n = d$. If $e_i \not\simeq e_{i+1}$ is $a \not\simeq b$, then we have $c \equiv_C a$ and $d \equiv_C b$, otherwise $e_i \not\simeq e_{i+1}$ must be $b \not\simeq a$, and we have $c \equiv_C b$ and $d \equiv_C a$.

For the converse implication, a similar reasoning is used. Let us assume w.l.o.g. that $a_{\downarrow C} = c_{\downarrow C}$ and $b_{\downarrow C} = d_{\downarrow C}$ (the other possibility being completely symmetric). There exists two sequences of constant symbols $a_1, \ldots, a_n$ and $b_1, \ldots, b_m$ such that $a_1 = c$, $a_n = a$, $b_1 = b$, $b_n = d$ and for all $i \in \{1, \ldots, n-1\}$ and $j \in \{1, \ldots, m-1\}$, the literals $a_i \not\simeq a_{i+1}$ and $b_j \not\simeq b_{j+1}$ all occur in C. By concatenating these two sequences, we create the sequence $e_1(= a_1), \ldots, e_n(= a_n), e_{n+1}(= b_1), \ldots, e_{n+m}(= b_m)$ such that for all $i \in \{1, \ldots, n+m-1\}$, the literals $e_i \not\simeq e_{i+1}$ occur in $a \not\simeq b \vee C$, thus ensuring that $c \equiv_{a \not\simeq b \vee C} d$. ∎

The next proposition introduces the notion of a *normal form* for clauses in $\mathbb{E}_0$, which in particular permits to test efficiently whether a clause is tautological. The intuition behind this proposition is that the relation $\equiv_C$ can be defined in a canonical way by stating that each constant a is mapped to its normal form $a_{\downarrow C}$, which if $a \neq a_{\downarrow C}$ is expressed by the negative literal $a \not\simeq a_{\downarrow C}$. Then each constant a can be replaced by its normal form in the positive part of the clause.

Proposition 1.8 *Every clause C is equivalent to the clause:*

$$C_{\downarrow} \stackrel{def}{=} \bigvee_{a \in \Sigma_0, a \neq a_{\downarrow C}} a \not\simeq a_{\downarrow C} \vee \bigvee_{a \simeq b \in C} a_{\downarrow C} \simeq b_{\downarrow C}.$$

Furthermore, C is a tautology iff $C_{\downarrow}$ contains a literal $a \simeq a$.

Proof. By definition of $a_{\downarrow C}$, we have $\neg C \models a \simeq a_{\downarrow C}$, for every constant a. Furthermore, for every literal $a \simeq b \in C$, we have $a \simeq a_{\downarrow C}, b \simeq b_{\downarrow C}, a_{\downarrow C} \simeq b_{\downarrow C} \models a \simeq b \models C$ and therefore $C_{\downarrow} \models C$, i.e. $\neg C \models \neg C_{\downarrow}$. Conversely, for every constant a, the relation $\neg C_{\downarrow} \models a \simeq a_{\downarrow C}$ holds by definition of $C_{\downarrow}$. Let l be a

literal in C. If l is a negative literal $b \not\simeq a$ then we have $\neg C \models b \simeq a$, hence $b_{\downarrow C} = a_{\downarrow C}$, thus $\neg C_\downarrow \models l^c$. If l is a positive literal $b \simeq a$ then $C_\downarrow$ contains a literal $a_{\downarrow C} \simeq b_{\downarrow C}$ and $a_{\downarrow C} \not\simeq b_{\downarrow C}, a \simeq a_{\downarrow C}, b \simeq b_{\downarrow C} \models l^c$, therefore we must have $\neg C_\downarrow \models l^c$. Consequently, $\neg C \models l^c$.

By definition, if $C_\downarrow$ contains $a \simeq a$ then $C_\downarrow$ is equivalent to $\top$, and thus C is a tautology. Conversely, if $C_\downarrow$ contains no such literal, then we have $a \not\equiv_C b$, for every literal $a \simeq b \in C_\downarrow$ (since $a = a_{\downarrow C}$ and $b = b_{\downarrow C}$ by definition of $C_\downarrow$) thus the interpretation $\equiv_C$ falsifies every literal in $C_\downarrow$ (since every negative literal in $C_\downarrow$ is of the form $a \not\simeq a_{\downarrow C}$ and thus must be false in $\equiv_C$). Thus $C_\downarrow$ cannot be equivalent to $\top$ and C is not a tautology. ∎

Definition 1.9 A non-tautological clause C is in *normal form* if $C = C_\downarrow$ and if, moreover, all literals occur at most once in C. ◊

Unless stated otherwise[1], we assume that all literals occur at most once in all non-projected clauses, allowing us to denote the normal form of a clause C simply by $C_\downarrow$ without additional specifications.

Example 1.10 The clause $C = a \not\simeq b \vee b \not\simeq c \vee d \not\simeq e \vee a \simeq e$ of Example 1.4 is equivalent to the clause in normal form $C_\downarrow = b \not\simeq a \vee c \not\simeq a \vee e \not\simeq d \vee a \simeq d$. Let $D = a \not\simeq b \vee b \not\simeq c \vee a \simeq c$, then $D_\downarrow$ is $b \not\simeq a \vee c \not\simeq a \vee a \simeq a$, and therefore D is a tautology. ♣

1.2 Results specific to $\mathbb{E}_1$

Some of the results of this subsection extend notions presented in the previous subsection from $\mathbb{E}_0$ to $\mathbb{E}_1$. We begin by extending to $\mathbb{E}_1$ the normal form of clauses.

Definition 1.11 A non-tautological clause C is in *normal form* if:

1. every negative literal l in C is such that $l_{\downarrow C \setminus l} = l$;
2. every literal $t \simeq s \in C$ is such that $t = t_{\downarrow C}$ and $s = s_{\downarrow C}$;
3. there are no two distinct positive literals l, m in C such that $m_{\downarrow l^c \vee C^-}$ is a tautology;
4. C contains no literal of the form $t \not\simeq t$ or $t \simeq t$;
5. the literals in C occur exactly once in C.

The normal form equivalent to C is denoted by $C_\downarrow$. ◊

Remark 1.12 *The differences between this normal form and the one for clauses in $\mathbb{E}_0$ lie with points 1 and 3 of Definition 1.11. They strengthen the requirements on negative and positive literals resp. to cover the non-flat ones.*

Example 1.13 Using the ordering $a \prec b \prec c \prec e \prec f(a) \prec f(b)$ on terms, the clause $c \not\simeq b \vee e \not\simeq b \vee f(b) \simeq f(a)$ is the normal form of the clauses $c \not\simeq b \vee e \not\simeq b \vee f(c) \simeq f(a)$, $c \not\simeq b \vee e \not\simeq b \vee f(e) \simeq f(a)$, $c \not\simeq e \vee e \not\simeq b \vee f(b) \simeq f(a)$, etc... ♣

1. I.e. everywhere except in Part II for the clauses in relaxed normal form.

To prove the uniqueness of the clausal normal form, a rewriting system [2] is associated with each clause in normal form.

Definition 1.14 Let C be a clause in normal form. The rewriting system R_C associated to C is such that:

$$t \to s \in R_C \text{ iff } t \not\succeq s \in C \text{ with } t \succ s \qquad \Diamond$$

Due to the definition of a clausal normal form, the associated rewriting systems are always convergent. We denote by $t_{\downarrow R_C}$ the term $t \in \mathfrak{T}(\Sigma)$ on which all possible rewriting rules from a rewrite system R_C have been applied. Note that $t_{\downarrow R_C} = t_{\downarrow C}$.

Proposition 1.15 *Let C be a clause in normal form and t be a term in $\mathfrak{T}(\Sigma)$. The term $t_{\downarrow R_C}$ is obtained from t by using only the rules of R_C that have a left-hand side smaller or equal to t.*

Proof. To compute $t_{\downarrow R_C}$, rewriting rules can be applied only on t or its sub-terms. All the sub-terms of t are smaller than t and by definition of R_C a rewriting always replaces a term by a smaller one. ∎

Theorem 1.16 *The normal form of a non-tautological clause C is the smallest clause equivalent to C.*

Proof. We first prove the uniqueness of the normal form by considering the negative literals in a clause and then the positive ones. Consider two equivalent clauses C_1 and C_2 that are in normal form, so that $\equiv_{C_1} = \equiv_{C_2}$.

By Definition 1.14, C_1 and C_2 are associated to rewriting systems respectively denoted by R_1 and R_2. If $C_1^- \neq C_2^-$ then $R_1 \neq R_2$. Let us consider the smallest rule $t \to s$ with $t \succ s$ that does not appear in both rewriting systems, w.l.o.g. we assume $t \to s \in R_1$ and $t \to s \notin R_2$. Since C_1 and C_2 are equivalent and $t_{\downarrow R_1} = s$, we also have $t_{\downarrow R_2} = s$. By Proposition 1.15, the left-hand side of the rules used to rewrite t are smaller or equal to t. Note that since t is at least rewritten into s in R_1, there must be at least one rule in R_2 that can be applied to t.

- If in R_2, the rules applicable to t are all smaller than $t \to s$, then these rules also appear in R_1, making $t \to s$ redundant in R_1 and $t \not\succeq s$ redundant in C_1, thus contradicting the definition of the normal form (Definition 1.11, point 6).
- Otherwise there is a rule $t \to v \in R_2$, with $t \succ v \succ s$, and $t \to v \notin R_1$ by definition of the normal form. This same definition ensures that $v = s$, which is impossible under the current hypotheses.

We now consider the case of the positive literals. The equivalence of C_1 and C_2 allows us to invoke Theorem 1.27 in both directions. Let l_1 be a positive literal in C_1. There exists a literal l_2 in C_2 such that ${l_2}_{\lfloor {l_1}^c \vee C_2^-}$ is a tautology, hence $l_1 \models C_2^- \vee l_2$. Similarly for l_2, there is a positive literal m_1 in C_1 such that ${m_1}_{\lfloor {l_2}^c \vee C_1^-}$, so that $l_2 \models C_1^- \vee m_1$. By combining the two entailment relations,

given that $C_1^- \equiv C_2^-$, we deduce that $l_1 \models C_1^- \vee m_1$, i.e. ${m_1}^c \models C_1^- \vee {l_1}^c$, thus $m_{1\restriction C_1^- \vee {l_1}^c}$ is a tautology. This contradicts point 5 of the normal form definition, unless $m_1 = l_1$. Since this property is symmetric for C_1 and C_2, it follows that $C_1^- \vee l_1 \equiv C_2^- \vee l_2$, hence $l_{1\restriction C_1} \equiv l_{2\restriction C_2}$ and so $l_{1\restriction C_1} = l_{2\restriction C_2}$. In addition C_1 and C_2 are in normal form, thus $l_1 = l_2$ and by symmetry this result can be generalized to $C_1^+ = C_2^+$.

As for the proof of minimality, it stems directly from the definition of the normal form. Consider a clause C that is not in normal form. One of the points of the definition must be contradicted.

— If C contains a negative literal $t \not\simeq s$ with $s \prec t$ and such that $s \neq t_{\restriction C}$, it is greater than the equivalent $s \not\simeq t_{\restriction C} \vee t \not\simeq t_{\restriction C}$ (which may not even appear in the normal form of C if it is implied by other negative literals of this clause).

— If C contains a positive literal $u \simeq v$ such that $u \neq u_{\restriction C}$ or $v \neq v_{\restriction C}$ (or both) then replacing this literal with $u_{\restriction C} \simeq v_{\restriction C}$ yields a smaller equivalent clause.

— No literal $t \simeq t$ can be present in a non-tautological clause and removing literals of the form $t \not\simeq t$ yields a smaller equivalent clause.

— The two last criteria guaranty the absence of redundant literals without which a smaller equivalent clause is also generated. ∎

2 Entailment and redundancy detection

We now introduce conditions that will permit us to design efficient methods to test if a given clause is redundant w.r.t. another clause.

2.1 Results specific to $\mathbb{E}_0$

Definition 1.17 Let C, D be two clauses. The clause D E-*subsumes* C, written $D \leq_{\text{E}} C$, iff the two following conditions hold:

— $\equiv_D \subseteq \equiv_C$,

— for every positive literal $l \in D$, there exists a literal $l' \in C$ such that $l \equiv_C l'$.

If S, S' are sets of clauses, we write $S \leq_{\text{E}} C$ if $\exists D \in S$, such that $D \leq_{\text{E}} C$ and we write $S \leq_{\text{E}} S'$ if $\forall C \in S', S \leq_{\text{E}} C$. ◊

Proposition 1.18 *Let C and D be two clauses.*

$$\equiv_D \subseteq \equiv_C \quad \textit{iff every negative literal in } D_{\restriction C} \textit{ is a contradiction.}$$

Intuitively, testing $D \models C$ is performed by verifying that $\neg C \models \neg D$. To this purpose, we first check that all equations in $\neg D$ are logical consequences of those in $\neg C$, which can be easily done by checking that the relation $\equiv_D \subseteq \equiv_C$ holds using Proposition 1.18. Then, we consider the negative literals in $\neg D$. Such a literal l^c can only be entailed by $\neg C$ iff $\neg C$ contains a literal ${l'}^c$ that can be reduced to l^c by the relation $\equiv_C$.

Example 1.19 Let $C = a \not\simeq b \vee b \not\simeq c \vee d \not\simeq e \vee a \simeq e$ be the clause of Example 1.4. C is E-subsumed by the clauses $a \not\simeq b \vee a \not\simeq c$, $a \not\simeq b \vee c \simeq e$ and $c \simeq d$. However, it is not E-subsumed by the clause $a \not\simeq d$, because $a_{\downarrow C} \neq d_{\downarrow C}$, or by the clause $a \simeq b$, because there is no literal $l \in C$ such that $(a \simeq b)_{\downarrow C} = l_{\downarrow C}$. ♣

Example 1.20 The clause $d \simeq a$ E-subsumes the clause $D = a \not\simeq e \vee d \simeq e$, because the clause $d \simeq a$ contains no negative literal (and thus $\equiv_{d \simeq a}$ is the identity) and the D-representatives of the literals $d \simeq e$ and $d \simeq a$ are identical.♣

Theorem 1.21 *Let C and D be two clauses and assume that C is not a tautology. Then $D \models C$ iff $D \leq_{\mathrm{E}} C$.*

Proof. Assume that $D \models C$, that C is not a tautology and that $D \not\leq_{\mathrm{E}} C$. If there is a negative literal $b \not\simeq a$ in D such that $b_{\downarrow C} \neq a_{\downarrow C}$, then $a \not\simeq b \not\models C$, hence we cannot have $D \models C$, since $b \not\simeq a \in D$. Now, consider a positive literal $b \simeq a \in D$ and assume that $b_{\downarrow C} \simeq a_{\downarrow C}$ does not occur in $C_{\downarrow C}$. We consider the interpretation $\mathcal{I}$ such that $=_{\mathcal{I}}$ is the smallest reflexive, symmetric and transitive relation satisfying $b =_{\mathcal{I}} a$ and $d =_{\mathcal{I}} c$ for every $c, d \in \Sigma_0$ such that $d \not\simeq c \in C$. It is clear that $\mathcal{I} \models D$, thus we must have $\mathcal{I} \models C$. Furthermore, $\mathcal{I}$ falsifies all the negative literals in C, by definition. Therefore, $\mathcal{I}$ must satisfy a positive literal $d \simeq c$ in C. By definition of $=_{\mathcal{I}}$ this means that there exists a sequence of constant symbols $c_1, \ldots, c_n$ such that $c_1 = d$, $c_n = c$ and for every $i \in [1, n-1]$, one of the following holds:

— $c_i \not\simeq c_{i+1} \in C$,
— $c_i = a$ and $c_{i+1} = b$,
— $c_{i+1} = a$ and $c_i = b$.

If for every $i \in [1, n-1]$ the first condition holds, then we have $d \not\simeq c \models C$, and therefore C must be a tautology (since $d \simeq c$ occurs in C), which contradicts our hypothesis. Otherwise, we can assume, without loss of generality, that the sequence $c_1, \ldots, c_n$ is minimal, so that there is *exactly* one index i satisfying the second condition. In this case we must have $d \not\simeq a \models C$ and $b \not\simeq c \models C$ (or $d \not\simeq b \models C$ and $a \not\simeq c \models C$), and thus $d_{\downarrow C} = a_{\downarrow C}$ and $b_{\downarrow C} = c_{\downarrow C}$ (or $d_{\downarrow C} = b_{\downarrow C}$ and $b_{\downarrow C} = a_{\downarrow C}$). But then, since $d \simeq c \in C$, this entails that $a_{\downarrow C} \simeq b_{\downarrow C}$ occurs in C which again contradicts our hypothesis.

Conversely, assume that $D \leq_{\mathrm{E}} C$. Let $\mathcal{I}$ be a model of D. By definition $\mathcal{I}$ satisfies some literal $l \in D$. If l is a negative literal $b \not\simeq a$ then we have $b_{\downarrow C} = a_{\downarrow C}$, and thus $b \not\simeq a \models C$. Consequently, $\mathcal{I} \models C$. If l is a positive literal $b \simeq a$ then there exists a literal $b' \simeq a' \in C$ such that ${b'}_{\downarrow C} = b_{\downarrow C}$ and ${a'}_{\downarrow C} = a_{\downarrow C}$. If $\mathcal{I} \models c \not\simeq c_{\downarrow C}$ for some constant symbol $c \in \Sigma_0$ we have $\mathcal{I} \models C$, since by definition $c \not\simeq c_{\downarrow C} \models C$. Thus we can assume that $\mathcal{I} \models a \simeq a', b \simeq b'$, and since $\mathcal{I} \models b \simeq a$ we deduce that $\mathcal{I} \models b' \simeq a'$, hence that $\mathcal{I} \models C$. ∎

Remark 1.22 *Thanks to this theorem, it is possible to test entailment in $\mathbb{E}_0$ in a syntactic way, using* E*-subsumption to verify that:*

— *every negative literal $l \in D^-$ is such that $l_{\downarrow C}$ is a contradiction,*
— *every positive literal $l \in D^+$ is such that $l_{\downarrow C} \in C$.*

We call this test the projection method since it relies heavily on the projection of literals.

In the following, we will actually use a slightly more restrictive version of this criterion for redundancy elimination: we impose that the positive literals in $D_{\restriction C}$ are mapped to *pairwise distinct* literals in $C_{\restriction C}$.

Definition 1.23 Let C and D be two clauses. The clause D I-*subsumes* C, written $D \leq_{\mathrm{I}} C$, iff the following conditions hold:

— $\equiv_D \subseteq \equiv_C$,

— there exists an *injective* function γ from D^+ to C^+ such that for any literal $l \in D^+$, $l \equiv_C \gamma(l)$.

This notion is extended to formulæ in the same fashion as for E-subsumption.$\Diamond$

This additional restriction is necessary to prevent the factors of a clause from being redundant w.r.t. the initial clause.

Example 1.24 Let $C = a \simeq b \vee c \simeq b$ and $D = a \simeq b \vee a \not\simeq c$. Then $C \leq_{\mathrm{E}} D$ but $C \not\leq_{\mathrm{I}} D$. ♣

The following proposition details the relationship between clausal normal form, E-subsumtion and I-subsumption.

Proposition 1.25 *Let C and D be two clauses.*

1. *If $C \leq_{\mathrm{I}} D$ then $C \leq_{\mathrm{E}} D$.*
2. *If $C \leq_{\mathrm{I}} D$ then $C \models D$.*
3. *If $C \leq_{\mathrm{I}} D$ then $C \leq_{\mathrm{E}} D_{\downarrow}$.*
4. *If $C \leq_{\mathrm{I}} D$ then $C_{\downarrow} \leq_{\mathrm{I}} D$.*

Proof. Each point is proved separately.

1. This result is trivial by definition of E-subsumption and I-subsumption.
2. This point is a consequence of the previous point and Theorem 1.21.
3. This point is a consequence of the first point, and of the fact that $D \equiv D_{\downarrow}$
4. This point holds because $\equiv_{C_{\downarrow}} = \equiv_C$ and there exists an injective map from the positive literals of $C_{\downarrow}$ to those of C by Proposition 1.8. ■

2.2 Results specific to $\mathbb{E}_1$

The following proposition and theorem extend the projection method introduced in the previous subsection to $\mathbb{E}_1$. It is used for testing entailment in a syntactic way.

Proposition 1.26 *Let C be a clause, and s, t be two terms. If there is no positive literal l in C such that $l_{\restriction C \vee s \not\simeq t}$ is tautological, then $C \vee s \not\simeq t$ is not a tautology.*

PROOF. Let $D = C \vee t \not\simeq s$. Let $\mathcal{I} = \equiv_C$ and $\mathcal{J} = \equiv_D$. We show that $\mathcal{J}$ is a counter-model of D. Note that in particular, if $s =_{\mathcal{I}} t$, then $\mathcal{I} = \mathcal{J}$. $\mathcal{J} \not\models s \not\simeq t$ since $s =_{\mathcal{J}} t$ and for any negative literal $u \not\simeq v \in C$, by definition $u \not\simeq v \in D$, thus $u =_{\mathcal{J}} v$ and finally $\mathcal{J} \not\models u \not\simeq v$. For the positive literals of C, let $u \simeq v \in C$, and assume $\mathcal{J} \models u \simeq v$. Under this assumption, $u \not\simeq v \models D$, hence $u_{\downarrow D} = v_{\downarrow D}$, contradicting the hypothesis about the positive literals of C. Thus $\mathcal{J} \not\models u \simeq v$.■

Theorem 1.27 *Let C and D be two non-tautological clauses. The relation $D \models C$ holds iff for every negative literal l in D, the literal $l_{\downarrow C}$ is a contradiction and for every positive literal l in D, there exists a positive literal m in C such that $m_{\downarrow C \vee l^c}$ is tautological.*

PROOF. First assume that $D \models C$. Consider a negative literal $s \not\simeq t \in D$. Then we must have $s \not\simeq t \models C$, thus $t_{\downarrow C} = s_{\downarrow C}$, and the corresponding literal in $D_{\downarrow C}$ is a contradiction. Now consider a positive literal $s \simeq t \in D$. If there is no positive literal m in C such that $m_{\downarrow C \vee s \not\simeq t}$ is a tautology then $C \vee s \not\simeq t$ is not a tautology by Proposition 1.26. But $D \vee s \not\simeq t \models C \vee s \not\simeq t$, and $D \vee s \not\simeq t$ is a tautology, which yields a contradiction.

For the converse implication, we prove that every literal in D entails C. Let $s \not\simeq t \in D$, then by hypothesis we have $s_{\downarrow C} = t_{\downarrow C}$, thus $[s]_C = [t]_C$, and by definition $s \not\simeq t \models C$. Let $s \simeq t \in D$, then by hypothesis there is a literal $u \simeq v \in C$ such that $u_{\downarrow C \vee s \not\simeq t} = v_{\downarrow C \vee s \not\simeq t}$. It follows that $u \not\simeq v \models C \vee s \not\simeq t$, which is equivalent to $s \simeq t \models C \vee u \simeq v$. Since $u \simeq v \in C$, we conclude that $s \simeq t \models C$. ■

Example 1.28 Given the order $a \prec b \prec c \prec e \prec f(a) \prec f(b)$ on terms, let $C = e \not\simeq c \vee a \simeq c$ and $D = e \not\simeq b \vee b \not\simeq c \vee f(a) \simeq f(b)$ and let $l = e \not\simeq c$ and $m = a \simeq c$ be the literals of C. We have, $l_{\downarrow D} = b \not\simeq b$ because $[b]_D = \{b, c, e\}$ and $\min_{\prec}([b]_D) = b$. Moreover the literal $f(a) \simeq f(b) \in D$ is such that $(f(a) \simeq f(b))_{\downarrow D \vee m^c} = f(a) \simeq f(a)$, which, by Theorem 1.27, proves that D is redundant w.r.t. C. ♣

Remark 1.29 *Compared to the projection method for $\mathbb{E}_0$, the test on positive literals is different. In fact, due to the presence of function symbols and the substitutivity axiom of equality, the previous test is not enougth in $\mathbb{E}_1$. Going back to the previous example, let us consider once again the literal $l = a \simeq c \in C$. It is clear that $l_{\downarrow D} \notin D$. The literal appearing in D is $f(a) \simeq f(b)$, that is implied by $l_{\downarrow D}$. This is detected by the extended projection method defined in this subsection.*

The notion of E-subsumption is extended to $\mathbb{E}_1$ to correspond to the newly defined projection method, thus preserving the result of Theorem 1.21, i.e. $D \models C$ iff $D \leq_{\mathrm{E}} C$ for any two non-tautological clauses C and D.

Chapter 2

Variations on the $\mathcal{K}$-paramodulation Calculus

In this chapter, we define a variant of the paramodulation calculus that is deductive-complete. The deductive-completeness result is proved in the first section. The second and third sections introduce two variants of this calculus that use the atomic implicates of the input formula to simplify the considered problem. In Section 2, this simplification is done as a preprocessing of the input formula and in Section 3 we introduce a method to perform similar simplifications during the saturation process.

1 Main calculus

Our objective is to create a calculus operating on clauses in $\mathbb{E}_0$ that computes all the prime implicates of a formula. To achieve this, we define the $\mathcal{K}$-paramodulation calculus in $\mathbb{E}_0$.

The principle underlying this calculus is to assert equations rather than proving them, in contrast to the superposition calculus. For example, let us consider the formula $S = \{a \simeq b, c \simeq d\}$. By superposition, nothing can be generated from S since no constant occurs in both clauses. In contrast, with $\mathcal{K}$-paramodulation, we want to be able to generate clauses like $a \not\simeq c \vee b \simeq d$, which is a prime implicate of S and can also be seen as the result of the paramodulation of $a \simeq b$ into $c \simeq d$ under the assumption that a and c are equal. To do so, we must allow the unification of constants that are not known to be equal, like a and c in S, by adding to the generated clause the negation of a hypothesis justifying this superposition, here, the literal $a \not\simeq c$.

1.1 Definition of the $\mathcal{K}$-paramodulation calculus

The principle presented in the introductory paragraph is formalized in the following calculus.

Definition 2.1 The *simple $\mathcal{K}$-paramodulation calculus* is defined in $\mathbb{E}_0$ by three inference rules:

$$\text{Positive Paramodulation } (\mathtt{P}^+): \quad \frac{a \simeq b \vee C \qquad a' \simeq c \vee D}{a \not\simeq a' \vee b \simeq c \vee C \vee D}$$

$$\text{Negative Paramodulation } (\mathtt{P}^-): \quad \frac{a \not\simeq b \vee C \qquad a' \simeq c \vee D}{a \not\simeq a' \vee c \not\simeq b \vee C \vee D}$$

$$\text{Factorization } (\mathtt{F}): \quad \frac{a \simeq b \vee a' \simeq b' \vee C}{a \simeq b \vee a \not\simeq a' \vee b \not\simeq b' \vee C}$$

where the consequents are systematically normalized. ◊

Compared to the superposition calculus (see Definition ii.17),
- all the ordering constraints have been removed so that all prime implicates can be generated,
- there is no use for a Reflection rule because the generated clauses are systematically normalized.

The rules $\mathtt{P}^+$ and $\mathtt{P}^-$ are similar to the standard paramodulation rule, except that the unification of the terms a and a' is omitted and replaced by the addition of the literal $a \not\simeq a'$ ensuring that these terms are semantically equivalent. Similarly, $\mathtt{F}$ factorizes the literals $a \simeq b$ and $a' \simeq b'$ under the assumption that $a \simeq a'$ and $b \simeq b'$.

Remark 2.2 *The simple $\mathcal{K}$-paramodulation calculus simulates the superposition calculus. For example, from $a \simeq b \vee C$ and $a \simeq c \vee D$, the clause $b \simeq c \vee C \vee D$ is generated by standard paramodulation if the ordering constraints are respected. This clause is equivalent to $(a \not\simeq a \vee b \simeq c \vee C \vee D)_{\downarrow}$, that is generated by $\mathcal{K}$-paramodulation.*

We shall prove (see Lemma 2.11) that this calculus is deductive-complete, in the sense that it can generate all the prime implicates of a set of flat clauses. However, this property does not hold in the presence of redundancy elimination rules, as the following example shows:

Example 2.3 Consider the set: $S = \{c \not\simeq a \vee d \not\simeq a,\ a \simeq b \vee c \not\simeq b \vee d \not\simeq b\}$ and let $C = c \not\simeq b \vee d \not\simeq b$. It is clear that $S \models C$ (e.g. by paramodulating twice the second clause into the first one on literal $a \simeq b$) and that C is a prime implicate of S. However, it can be verified that S is saturated up to redundancy, i.e. all the clauses generated from S in one iteration are redundant in S. For example, paramodulating the literal $a \simeq b$ of the second clause into the literal $c \not\simeq a$ of the first clause yields $(a \not\simeq a \vee c \not\simeq b \vee d \not\simeq a \vee c \not\simeq b \vee d \not\simeq b)_{\downarrow} = c \not\simeq a \vee d \not\simeq a$, which is I-subsumed by the first clause in S. Therefore, with the simple $\mathcal{K}$-paramodulation calculus augmented with eager redundancy elimination rules, C cannot be generated. ♣

From a practical point of view, redundancy elimination is of course essential. Thus we have to extend the calculus so that completeness still holds when redundant clauses are deleted. A slight modification of the $\mathtt{P}^-$ rule suffices to correct

this defect: the modification permits to perform simultaneous paramodulations into the negative literals of a clause. The result is the following calculus.

Definition 2.4 The *$\mathcal{K}$-paramodulation calculus* is defined in $\mathbb{E}_0$ by three inference rules:

$$\text{Paramodulation } (\mathtt{P}^+): \quad \frac{a \simeq b \vee C \qquad a' \simeq c \vee D}{a \not\simeq a' \vee b \simeq c \vee C \vee D}$$

$$\text{Factorization } (\mathtt{F}): \quad \frac{a \simeq b \vee a' \simeq b' \vee C}{a \simeq b \vee a \not\simeq a' \vee b \not\simeq b' \vee C}$$

$$\text{Negative Multi-Paramodulation } (\mathtt{M}): \quad \frac{\bigvee_{i=1}^{n}(a_i \not\simeq b_i) \vee C \qquad c \simeq d \vee D}{\bigvee_{i=1}^{n}(a_i \not\simeq c \vee d \not\simeq b_i) \vee C \vee D}$$

We write $S \vdash_{\mathcal{K}} C$ if C is generated from premises in S by one application of the rules $\mathtt{P}^+$, $\mathtt{F}$ or $\mathtt{M}$. The premises are assumed to be in normal form and the consequent is normalized before being stored. ◊

The rules $\mathtt{P}^+$ and $\mathtt{F}$ are the same as in the simple $\mathcal{K}$-paramodulation calculus and the rule $\mathtt{M}$ that replaces $\mathtt{P}^-$ corresponds to the simultaneous unification of c and the a_i to paramodulate the second clause into the first one on each of the $a_i \not\simeq b_i$ at the same time. Note that $\mathtt{M}$ is strictly more general than $\mathtt{P}^-$ in the sense that it allows one to infer strictly more clauses (the branching factor is thus increased). In fact, for $n = 1$ this rule is the same as $\mathtt{P}^-$.

Remark 2.5 *Note that the* $\mathtt{M}$ *rule is more general than what is known as simultaneous paramodulation in the literature [7] where simultaneous unifications are only allowed with the different occurrences of the same term, like when all the* a_i *of the* $\mathtt{M}$ *rule are equal.*

Example 2.6 The application of the rule $\mathtt{P}^+$ on $d \simeq c \vee d \simeq b$ and $d \simeq a$ yields the following clauses (among others):

$$\begin{array}{ll} d \not\simeq d \vee a \simeq c \vee d \simeq b & (\text{terms } d \text{ and } d) \\ c \not\simeq a \vee d \simeq d \vee d \simeq b & (\text{terms } c \text{ and } a) \\ b \not\simeq d \vee d \simeq c \vee d \simeq a & (\text{terms } b \text{ and } d) \end{array}$$

The clauses are normalized afterwards: the first clause is replaced by $a \simeq c \vee d \simeq b$, the second stays the same and the third one is replaced by $d \not\simeq b \vee c \simeq b \vee b \simeq a$ (since $b \prec d$). Note that the redundancy elimination rule deletes the second clause since it is a tautology. ♣

Example 2.7 The rule $\mathtt{F}$ applies on the clause $a \simeq b \vee a \simeq c \vee c \simeq d$, yielding, e.g., $a \simeq b \vee a \not\simeq a \vee b \not\simeq c \vee c \simeq d$. The normal form of the latter is $c \not\simeq b \vee a \simeq b \vee b \simeq d$. ♣

Example 2.8 Consider the clauses $a \not\simeq b \vee a \not\simeq c$ and $a \simeq b$. With $n = 1$, the rule $\mathtt{M}$ applies on the couples of literals $(a \not\simeq b, a \simeq b)$ or $(a \not\simeq c, a \simeq c)$, the first application yielding (among other, trivial results) $a \not\simeq a \vee b \not\simeq b \vee a \not\simeq c$, i.e., after normalization $a \not\simeq c$. It also applies with $n = 2$, yielding $a \not\simeq a \vee b \not\simeq b \vee a \not\simeq a \vee b \not\simeq c$, or, in normalized form, $b \not\simeq c$. ♣

Example 2.9 Going back to the formula $S = \{c \not\simeq a \vee d \not\simeq a,\ a \simeq b \vee c \not\simeq b \vee d \not\simeq b\}$ introduced in Example 2.3, the problem previously illustrated disappears by using the newly defined $\mathcal{K}$-paramodulation since $C = c \not\simeq b \vee d \not\simeq b$ can be generated by a single inference of the rule `M` in S. ♣

The deletion of redundant clauses is done using I-subsumption instead of entailment in the redundancy elimination rule. Note that I-subsumption is incomparable with the standard redundancy criterion (even if restricted to one-to-one redundancy test) because on the one hand the existence of an injective mapping between positive literals is assumed and on the other hand no ordering condition is imposed. The standard redundancy criterion based on the use of a reduction ordering (Definition ii.21) to compare clauses cannot be used here because it does not ensure the completeness of the $\mathcal{K}$-paramodulation calculus (although no simple example can be given to illustrate this fact).

Definition 2.10 A set of clauses S is *$\mathcal{K}$-saturated* iff for every non-tautological clause C that can be derived from S using these rules, there exists a clause $C' \in S$ such that $C' \leq_{\mathrm{I}} C$. ◊

1.2 Completeness of the $\mathcal{K}$-paramodulation calculus for implicate generation

Since the $\mathcal{K}$-paramodulation calculus simulates the superposition calculus [1], it is refutationaly complete: if a set of clauses S is closed under $\mathcal{K}$-paramodulation and does not contain the empty clause, then S is satisfiable. But as mentioned previously, refutational completeness is not enough to ensure that this calculus can generate all the prime implicates of a set of clauses. We need to prove the deductive completeness of the calculus: if S is $\mathcal{K}$-saturated and normalized, C is normalized and not a tautology and if $S \models C$ then $S \leq_{\mathrm{I}} C$. The proof proceeds in two steps. First, in Lemma 2.11, we prove the result without considering the deletion of redundant clauses, i.e. if S is closed under simple $\mathcal{K}$-paramodulation and $S \models C$ then there exists a clause $D \in S$ such that $D \models C$. Second, in Theorem 2.15, we build on the result from Lemma 2.11 to reach the desired conclusion.

Simple deductive completeness

We prove that a $\mathcal{K}$-saturated set S subsumes all its implicates. Thus, if moreover S is free of redundancy then it contains exactly its set of prime implicates, up to equivalence. We first state the following result, that is similar in essence, but much weaker since it concerns the simple $\mathcal{K}$-paramodulation calculus and does not cope with redundancy elimination.

1. The normal form of a clause C is the smallest clause equivalent to C (see Theorem 1.16 page 41) and thus it subsumes C. For this reason, the normalization of consequents generated by $\mathcal{K}$-paramodulation is compatible with the usual superposition calculus.

Lemma 2.11 *Let S be a set of clauses such that S contains every clause that can be derived from premises in S by simple $\mathcal{K}$-paramodulation. For every implicate C of S, there exists a clause $D \in S$ such that $D \models C$.*

PROOF SKETCH. The proof proceeds as follows. Under the hypotheses of the lemma, we assume that there exists an implicate C of S such that no $D \in S$ verifies $D \models C$ and we construct an equational interpretation satisfying $S \cup \neg C$ – thus contradicting the fact that $S \models C$. This construction is done in two steps. First, we associate to the pair (C, S) a propositional interpretation $\mathcal{I}(C, S)$, constructed by induction on a suitably chosen ordering $<_C$. Then, we show that $\mathcal{I}(C, S)$ is actually an equational model of $S \cup \neg C$ if S is closed and contains no implicant of C.

PROOF. Let C be a non-tautological clause such that for all $D \in S$, $D \not\models C$. We begin by defining an ordering $<_C$ on literals that will permit to distinguish the literals entailing C (by ensuring that these literals are smaller than the other ones), and to order all the literals according to their projection on C. For all literals l_1, l_2, $l_1 <_C l_2$ iff one of the conditions below holds:

- $l_1 \models C$ and $l_2 \not\models C$;
- $l_{1\downarrow C} = a_1 \bowtie b_1$, with $a_1 \succeq b_1$; $l_{2\downarrow C} = a_2 \bowtie b_2$, with $a_2 \succeq b_2$ and
  - $a_1 \prec a_2$, or
  - $a_1 = a_2$ and $b_1 \prec b_2$.

The relation $<_C$ is only a pre-ordering on atoms, but a total ordering on the projection of the atoms on C. This order is extended into an ordering $\ll_C$ for clauses using the standard multiset extension. We then define an interpretation $\mathcal{I}$ that will satisfy S but not C, by induction on the ordering $<_C$. Note that $\mathcal{I}$ is constructed as a *propositional* interpretation, i.e., it maps atoms to truth values. We shall prove later that $\mathcal{I}$ actually defines an equational interpretation, i.e. an equivalence relation on Σ_0.

Let $p_1 <_C \ldots <_C p_n$ be the set of all atoms projected on C. For all $i \in \{1 \ldots n\}$, we define the propositional interpretation $\mathcal{I}_i$ as follows.

1. For any atom l such that $l_{\downarrow C} = p_j$ with $j < i$, $\mathcal{I}_i \models l$ iff $\mathcal{I}_{i-1} \models l$.
2. For any atom l such that $l_{\downarrow C} = p_j$ with $j > i$, $\mathcal{I}_i \not\models l$.
3. For any atom l such that $l_{\downarrow C} = p_i$, $\mathcal{I}_i \models l$ iff
   (a) either p_i is of the form $a \simeq a$,
   (b) or $p_i \notin C_\downarrow$ and there exists a clause $D \vee l' \in S$ such that:
   - $l'_{\downarrow C} = p_i$,
   - $\forall l'' \in D,\ l'' <_C l'$,
   - $\mathcal{I}_{i-1} \not\models D$.

We denote by $\mathcal{I}$ the interpretation $\mathcal{I}_n$. For all $i \in \{1 \ldots n\}$, $\mathcal{I}$ coincides with $\mathcal{I}_i$ for all literals l such that $l_{\downarrow C} = p_j$ or $l_{\downarrow C} = {p_j}^c$, where $j \leq i$. In particular, given two atoms l and l' such that $l_{\downarrow C} = l'_{\downarrow C}$, necessarily, either $\mathcal{I} \models l$ and $\mathcal{I} \models l'$, or $\mathcal{I} \not\models l$ and $\mathcal{I} \not\models l'$.

We now show that $\mathcal{I}$ is actually an equational interpretation (i.e. that it satisfies the equality axioms), and that it satisfies the formula $S \cup \neg C$. We prove

first that $\mathcal{I} \not\models C$, assuming that $\mathcal{I}$ is an equational interpretation such that $\mathcal{I} \models S$, which will be proven later.

Assume that $\mathcal{I} \models C$. In this case there is a literal $l \in C$ such that $\mathcal{I} \models l$.

— If l is a positive literal, then there exists an $i \in \{1 \dots n\}$ such that $l_{\restriction C} = p_i$. By definition of $\mathcal{I}$, either p_i is of the form $a \simeq a$, in which case $C_{\downarrow}$ and C are both tautologies by Proposition 1.8, a contradiction; or $p_i \notin C_{\downarrow}$, hence $l \notin C$, again a contradiction.

— Otherwise l is a negative literal and there exists an $i \in \{1 \dots n\}$ such that $l_{\restriction C} = {p_i}^c$. In this case, p_i cannot be a tautology by Condition 3a of the definition of $\mathcal{I}$. But since l is of the form $a \not\simeq b$ and $l \models C$, necessarily $a_{\restriction C} = b_{\restriction C}$ and $p_i = a_{\restriction C} \simeq b_{\restriction C}$ is a tautology, a contradiction.

Now that $\mathcal{I} \not\models C$ is established, we prove that $\mathcal{I} \models S$, under the assumption that $\mathcal{I}$ is an equational interpretation. For $i = 1, \dots, n$, we consider the set $\mathcal{L}_{i,C}$ of literals l such that $p_i \not<_C l$, and define $S_{i,C} = \{D \in S \mid \forall l \in D,\, l \in \mathcal{L}_{i,C}\}$. We prove by induction that $\mathcal{I}_i \models S_{i,C}$. Let $D \in S_{i,C}$. If $D \in S_{i-1,C}$ then by the induction hypothesis $\mathcal{I}_{i-1} \models D$, and since $\mathcal{I}_i$ coincides with $\mathcal{I}_{i-1}$ on $\mathcal{L}_{i-1,C}$, $\mathcal{I}_i \models D$. We now assume that there exists a literal $l \in D$ such that $l_{\restriction C} \in \{p_i, {p_i}^c\}$, i.e. $D \in S_{i,C} \setminus S_{i-1,C}$.

1. If there are two literals l and l' in D such that $l_{\restriction C} = p_i$ and ${l'}_{\restriction C} = {p_i}^c$, then by construction, if $\mathcal{I}_i \not\models l$ then $\mathcal{I}_i \models l'$ and if $\mathcal{I}_i \not\models l'$ then $\mathcal{I}_i \models l$. Thus in both cases $\mathcal{I}_i \models D$.

2. If there is no literal $l \in D$ such that $l_{\restriction C} = p_i$, then there is at least one literal $l \in D$ such that $l_{\restriction C} = {p_i}^c$, because $D \in S_{i,C} \setminus S_{i-1,C}$. The clause D is of the form $l \vee D'$. If $\mathcal{I}_i \models {p_i}^c$, then the result is immediate. Otherwise, we distinguish the cases where p_i is and is not a tautology.
   — If p_i is a tautology then $l_{\restriction C}$ is a contradiction. By Theorem 1.21, $l \models C$, which, by definition of $<_C$, entails that $l' \models C$ for every $l' \in D'$. Thus $D \models C$, a contradiction;
   — If p_i is not a tautology then we use an inductive reasoning by assuming that all clauses $E \in S_{i,C}$ such that $E \ll_C D$ are such that $\mathcal{I}_i \models E$. Note that the base case, i.e. the clauses in $S_{i-1,C}$, is already covered. In the general case, by definition of $\mathcal{I}_i$, $p_i \notin C_{\downarrow}$ and there exists a clause $D'' \vee l' \in S$ such that ${l'}_{\restriction C} = p_i$, $l'' <_C l'$ for all $l'' \in D''$ and $\mathcal{I}_{i-1} \not\models D''$. Assume $l = a \not\simeq b$ and $l' = a' \simeq b'$ where $a_{\restriction C} = {a'}_{\restriction C}$ and $b_{\restriction C} = {b'}_{\restriction C}$, and let $E = a \not\simeq a' \vee b \not\simeq b' \vee D' \vee D''$. The clause $E_{\downarrow}$ is generated from of $l \vee D'$ and $l' \vee D''$ by inference rule $\mathtt{P}^-$, thus $E_{\downarrow} \in S_{i,C}$. Note that $E_{\downarrow} \ll_C E$ by definition of the normal form and $E \ll_C D$ because $a \not\simeq a' \models C$ and $b \not\simeq b' \models C$ thus $a \not\simeq a', b \not\simeq b' <_C p_i$ by definition of $<_C$, for all $l'' \in D''$, $l'' <_C p_i$ and $D' \ll_C D$ because D' contains one less maximal literal, namely l. Thus by induction $\mathcal{I}_i \models E_{\downarrow}$, but $\mathcal{I}_i \not\models a \not\simeq a', b \not\simeq b', D', D''$ by construction, so that $\mathcal{I}_i \not\models E$, a contradiction.

3. If there is no literal $l \in D$ such that $l_{\restriction C} = {p_i}^c$, then there is at least one literal $l \in D$ such that $l_{\restriction C} = p_i$ and D is of the form $D' \vee l$. If $\mathcal{I}_{i-1} \models D'$, then by definition $\mathcal{I}_i \models D$. Otherwise, if $p_i \in C_{\downarrow}$, then $l \models C$ by Theorem

1.21 and by definition of $<_C$, for all literals $l' \in D'$, necessarily $l' \models C$, so that $D \models C$, which contradicts our hypothesis. Thus $p_i \notin C_{\downarrow}$ and we reason by induction on the order $\ll_C$. If l is the only literal in D that is projected onto p_i in C, then l verifies Condition 3b of the definition of $\mathcal{I}_i$, hence $\mathcal{I}_i \models l$ and $\mathcal{I}_i \models D$. Otherwise, there is at least a second literal $l' \in D$ such that ${l'}_{\downarrow C} = p_i$ and D is of the form $l \vee l' \vee D'$. By applying the the rule F on l and l' in D, the clause $E_{\downarrow} = (a \not\simeq a' \vee b \not\simeq b' \vee l \vee D')_{\downarrow}$ is generated. This clause is such that $E_{\downarrow} \in S_{i,C}$ and $E_{\downarrow} \ll_C E \ll_C D$. By induction $\mathcal{I}_i \models E_{\downarrow}$ but $\mathcal{I}_i \not\models a \not\simeq a', b \not\simeq b', D'$, thus $\mathcal{I}_i \models l$ so $\mathcal{I}_i \models D$.

This proves that $\mathcal{I}_i \models S_{i,C}$, there remains to prove that the restriction of $\mathcal{I}_i$ to $\mathcal{L}_{i,C}$ is an equational interpretation. Let $l = a \simeq b$ be a literal such that $l_{\downarrow C} = p_i$.

Reflexivity: if l is a tautology, then so is p_i and by construction $\mathcal{I}_i \models l$.

Commutativity: since $a \simeq b$ and $b \simeq a$ are assumed to be identical, commutativity is naturally respected by $\mathcal{I}_i$.

Transitivity: assume $\mathcal{I}_i \models a \simeq b$ and $\mathcal{I}_i \models a \simeq c$, where $a \simeq c \in \mathcal{L}_{i,C}$, we prove that $\mathcal{I}_i \models b \simeq c$, provided that $b \simeq c \in \mathcal{L}_{i,C}$. There are several cases to consider, depending on whether one of $(a \simeq b)_{\downarrow C}$ or $(a \simeq c)_{\downarrow C}$ is a tautology.

1. If $b_{\downarrow C} = a_{\downarrow C} = c_{\downarrow C}$, then $(b \simeq c)_{\downarrow C} = p_i$ is a tautology and by construction, $\mathcal{I}_i \models b \simeq c$.
2. Assume $b_{\downarrow C} = a_{\downarrow C}$ and $(a \simeq c)_{\downarrow C}$ is not a tautology. Then there is a $j \leq i$ such that $(a \simeq c)_{\downarrow C} = p_j$, and there exists a clause $D \vee a' \simeq c' \in S$ such that $(a \simeq c)_{\downarrow C} = (a' \simeq c')_{\downarrow C}$, $D <_C (a' \simeq c')$ and $\mathcal{I}_{j-1} \not\models D$. By construction $\mathcal{I}_j \models a \simeq c$, and since $\mathcal{I}_i$ and $\mathcal{I}_j$ coincide on $\mathcal{L}_{j,C}$, we deduce that $\mathcal{I}_i \models a \simeq c$. But $(b \simeq c)_{\downarrow C} = (a \simeq c)_{\downarrow C}$, therefore $\mathcal{I}_i \models b \simeq c$.
3. The same reasoning proves the result if $c_{\downarrow C} = a_{\downarrow C}$ and $(a \simeq b)_{\downarrow C}$ is not a tautology.
4. Assume neither $(a \simeq b)_{\downarrow C}$ nor $(a \simeq c)_{\downarrow C}$ is a tautology. Then there exists a clause $D \vee d \simeq e \in S_{i,C}$ such that $(a \simeq b)_{\downarrow C} = (d \simeq e)_{\downarrow C}$, for all $l \in D$, $l <_C (d \simeq e)$ and $\mathcal{I}_{i-1} \not\models D$. W.l.o.g., we assume that $a_{\downarrow C} = d_{\downarrow C}$ and $b_{\downarrow C} = e_{\downarrow C}$. Similarly, for some $j \leq i$, there exists a clause $D' \vee d' \simeq f$ in $S_{j,C}$ such that $(a \simeq c)_{\downarrow C} = (d' \simeq f)_{\downarrow C}$, $D' <_C (d' \simeq f)$ and $\mathcal{I}_{j-1} \not\models D'$. W.l.o.g., we assume that $a_{\downarrow C} = {d'}_{\downarrow C}$ and $c_{\downarrow C} = f_{\downarrow C}$.

   Let $E = d \not\simeq d' \vee e \simeq f \vee D \vee D'$. The clause $E_{\downarrow} \in S$ by application of the rule $\mathtt{P}^+$on $D \vee d \simeq e$ and $D' \vee d' \simeq f$. If $E_{\downarrow} \notin S_{i,C}$, then since $d \not\simeq d' \models C$ and for all $l \in D \vee D'$, $l <_C p_i$, necessarily $p_i <_C (e \simeq f)_{\downarrow C}$ and there is nothing to prove. Otherwise, since $\mathcal{I}_i \models S_{i,C}$, we deduce that $\mathcal{I}_i \models E_{\downarrow}$; and since $\mathcal{I}_i \not\models D, D', d \not\simeq d'$, necessarily $\mathcal{I}_i \models e \simeq f$, and thus $\mathcal{I}_i \models b \simeq c$. ∎

Deductive completeness with redundancy elimination

Theorem 2.15 states a more powerful result than Lemma 2.11, namely that any implicate of a $\mathcal{K}$-saturated set S is I-subsumed by a clause in S. The proof of Theorem 2.15 requires the handling of redundancy elimination. We start by proving some preliminary results. The first one links E-subsumption and I-subsumption through factorization.

Proposition 2.12 *Let C, D be two clauses. Assume that $C \leq_{\mathrm{E}} D$. Then there exists a clause C' derivable by factorization from C such that $C' \leq_{\mathrm{I}} D$.*

Proof. We reason by induction on the number of positive literals appearing in the clause C. By definition, for all negative literals $a \not\simeq b \in D$, we have $a \equiv_D b$. Moreover, there exists a function γ mapping the positive literals in C to positive literals in D such that $\forall l \in C, l \equiv_D \gamma(l)$. Note that if γ is injective, then $C \leq_{\mathrm{I}} D$ and the proof is complete. If γ is not injective, C is of the form $a \simeq b \vee c \simeq d \vee C''$, where $(a \simeq b) \equiv_D (c \simeq d)$. W.l.o.g., we may assume that $a \equiv_D c$ and $b \equiv_D d$. Then inference rule F applied to C generates $(a \not\simeq c \vee b \not\simeq d \vee a \simeq b \vee C'')_{\downarrow}$. This clause satisfies the same requirements as C and contains one less positive literal. By induction, we deduce that there exists a clause C' derivable by factorization from C such that $C' \leq_{\mathrm{I}} D$. ∎

We also look into the details of a specific I-subsumption relation that appears in the proof of Lemma 2.14

Proposition 2.13 *Let $C = a' \simeq d' \vee b' \not\simeq c' \vee P_1' \vee P_2'$ and $D = a \simeq d \vee b \not\simeq c \vee P_1 \vee P_2$, where $P_1' \leq_{\mathrm{I}} P_1$, $P_2' \leq_{\mathrm{I}} P_2$, $(a \simeq b) \equiv_{P_1} (a' \simeq b')$ and $(c \simeq d) \equiv_{P_2} (c' \simeq d')$. Then $C \leq_{\mathrm{I}} D$.*

Proof. We verify each point of the definition of I-subsumption.

— Since $\equiv_{P_1'} \subseteq \equiv_{P_1}$ and $\equiv_{P_2'} \subseteq \equiv_{P_2}$, we only need to prove that $b' \equiv_D c'$. This is the case because $b \equiv_{P_1} b'$ thus $b \equiv_D b'$; $c \equiv_{P_2} c'$ thus $c \equiv_D c'$ and $b \not\simeq c \in D$ thus $b \equiv_D c$.

— Since $P_1' \leq_{\mathrm{I}} P_1$, there exists an injective function γ_1 from ${P_1'}^+$ to ${P_1}^+$ such that for all $l \in {P_1'}^+$, $l \equiv_D \gamma_1(l)$. A similar injective map γ_2 is associated with P_2 and P_2'. We define the function γ from C^+ to D^+ by:

$$\forall l \in C^+,\ \gamma(l) = \begin{cases} \gamma_1(l) & \text{if } l \in {P_1'}^+ \\ \gamma_2(l) & \text{if } l \in {P_2'}^+ \\ a \simeq d & \text{if } l = a' \simeq d' \end{cases}$$

Since γ_1 and γ_2 are both injective, γ is also injective. In addition, for the last point of the definition, $\gamma(a' \simeq d') \equiv_D a' \simeq d'$ because $a \equiv_D a'$ and $d \equiv_D d'$. Thus γ fits the requirements for the second point of the I-subsumption definition. ∎

The next lemma deals with inferences applied to positive literals only. It shows that any sequence of such inferences on a set of clauses S' can be simulated by applying inference rules on any set $S \leq_{\mathrm{I}} S'$.

Lemma 2.14 *Let S be a set of clauses that is $\mathcal{K}$-saturated. If $S \leq_{\text{I}} S'$ and if C is a clause deduced from clauses in S' by a sequence of applications of the rules* `F` *or* $\mathtt{P}^+$*, then $S \leq_{\text{I}} C$.*

Proof. The proof is done by induction on the number of inference steps. If C occurs in S' then the proof is immediate since $S \leq_{\text{I}} S'$. Otherwise, let $D_{\downarrow}$ be the first clause generated in the derivation leading to C. The clause $D_{\downarrow}$ must be deduced from clauses in S' by applying either the rule $\mathtt{P}^+$ or the rule `F`. We consider only the case of the rule $\mathtt{P}^+$, the case of the rule `F` is similar. Assume that $D_{\downarrow} = (a \simeq d \vee b \not\simeq c \vee P_1 \vee P_2)_{\downarrow}$ is generated by $\mathtt{P}^+$, from clauses $a \simeq b \vee P_1$ and $c \simeq d \vee P_2$. We prove that $S \leq_{\text{I}} D_{\downarrow}$. By definition, S contains a clause E that I-subsumes $a \simeq b \vee P_1$. If $E \leq_{\text{I}} P_1$, then $E \leq_{\text{I}} D$ thus $E \leq_{\text{E}} D_{\downarrow}$ and by Proposition 2.12 there exists a clause $E' \in S$ such that $E' \leq_{\text{I}} D_{\downarrow}$. The case where S contains a clause I-subsuming P_2 is symmetrical. Otherwise, S contains a clause of the form $a' \simeq b' \vee P'_1$, where $(a \simeq b) \equiv_{P_1} (a' \simeq b')$ and $P'_1 \leq_{\text{I}} P_1$ (since $\equiv_{P'_1}$ and $\equiv_{a' \simeq b' \vee P'_1}$ are identical and since no positive literal in P'_1 can be mapped to $a \simeq b$); and S contains also a clause $c' \simeq d' \vee P'_2$, with $(c' \simeq d') \equiv_{P_2} (c \simeq d)$ and $P'_2 \leq_{\text{I}} P_2$. The rule $\mathtt{P}^+$ applied to both clauses yields $E_{\downarrow} = (a' \simeq d' \vee b' \not\simeq c' \vee P'_1 \vee P'_2)_{\downarrow}$, and $E \leq_{\text{I}} a \simeq d \vee b \not\simeq c \vee P_1 \vee P_2$ by Proposition 2.13. Since S is $\mathcal{K}$-saturated, we have $S \leq_{\text{I}} E_{\downarrow}$, hence $S \leq_{\text{I}} a \simeq d \vee b \not\simeq c \vee P_1 \vee P_2$. Thus, $S \leq_{\text{I}} S' \cup \{a \simeq d \vee b \not\simeq c \vee P_1 \vee P_2\}$ and by the induction hypothesis, $S \leq_{\text{I}} C$. ∎

Theorem 2.15 *Let S be a normalized clause set that is $\mathcal{K}$-saturated. If $S \models C$ then $S \leq_{\text{I}} C$.*

Proof. Assume that $S \models C$ and let $S' = \{D' \mid \exists D \in S, D \leq_{\text{I}} D'\}$. Note that $S \subseteq S'$ and $S \leq_{\text{I}} S'$; thus $S \equiv S'$ and $S \models C$ if and only if $S' \models C$. We prove that all the clauses that can be derived from S' by simple $\mathcal{K}$-paramodulation are I-subsumed by S. If this is the case then S also I-subsumes the simple $\mathcal{K}$-paramodulation closure S'' of S', and since $S'' \leq_{\text{E}} C$ by Lemma 2.11, we will have the result by Proposition 2.12.

Since $S \leq_{\text{I}} S'$, by Lemma 2.14, we already know that S I-subsumes all clauses that can be obtained from clauses in S' by applying the rules `F` or $\mathtt{P}^+$. Thus we only consider the case of the application of the rule `M`. Let $a \not\simeq b \vee P_1$, $c \simeq d \vee P_2$ be two clauses in S', for which the rule `M` generates $Q_{\downarrow} = (a \not\simeq c \vee b \not\simeq d \vee P_1 \vee P_2)_{\downarrow}$. We prove that $S \leq_{\text{I}} Q_{\downarrow}$. The same reasoning as in the proof of Lemma 2.14, can be used to show that either $S \leq_{\text{I}} P_2$, in which case $S \leq_{\text{I}} Q_{\downarrow}$, or S contains a clause $c' \simeq d' \vee P'_2$ such that $c \equiv_{P_2} c'$, $d \equiv_{P_2} d'$ and $P'_2 \leq_{\text{I}} P_2$. We distinguish two cases involving $a \not\simeq b \vee P_1$.

— Assume that S contains a clause P'_1 such that $P'_1 \leq_{\text{I}} a \not\simeq b \vee Q$ and $\equiv_{P'_1} \subseteq \equiv_Q$. By definition, there exists an injective function γ mapping the positive literals in P'_1 to the positive literals in $a \not\simeq b \vee Q$ such that for every positive literal $l \in P'_1$, $l \equiv_{a \not\simeq b \vee Q} \gamma(l)$. Let D be the disjunction of positive literals $l \in P'_1$ such that $l \not\equiv_Q \gamma(l)$, and let D' be the disjunction of the literals $\gamma(l)$ for $l \in D$. Note that $D' \subseteq Q$, since $a \not\simeq b$ is negative. By definition, every negative literal in P'_1 I-subsumes $Q_{\downarrow}$, since $\equiv_{P'_1} \subseteq \equiv_Q$

by hypothesis, and for all positive literals l in $P'_1 \setminus D$, we have $l \equiv_Q \gamma(l)$. Thus P'_1 is of the form $Q'_1 \vee D$, where $Q'_1 \leq_{\mathrm{I}} Q_{\downarrow}$.
By construction $D \equiv_{a \not\simeq b \vee Q} D'$, thus for every constant symbol g occurring in D, there exists a constant g' occurring at the same position in D' such that $g \equiv_{a \not\simeq b \vee Q} g'$. We consider the clause D'' obtained from D by replacing every constant symbol g by a constant g'' chosen as follows:
— if $g \not\equiv_Q g'$ and $g \equiv_Q a$ then $g'' = d'$,
— if $g \not\equiv_Q g'$ and $g \equiv_Q b$ then $g'' = c'$,
— otherwise $g'' = g$.
We show that $g'' \equiv_Q g'$ for every constant g in D. Assume that this property is falsified for some constant g. If $g \equiv_Q g'$ then $g'' = g$ by definition of g'', a contradiction. If $g \equiv_Q a$ and $g \not\equiv_Q g'$, then by Proposition 1.7, $g' \equiv_Q b$, since $g \equiv_{a \not\simeq b \vee Q} g'$ by hypothesis. But then $g'' = d'$, and since $d' \equiv_{P_2} d$, we deduce that $d' \equiv_Q d \equiv_Q b \equiv_Q g'$, which contradicts our initial assumption. The proof is similar if $g \equiv_Q b$ and $g \not\equiv_Q g'$. Otherwise, we must have $g \not\equiv_Q a$, $g \not\equiv_Q b$ and $g = g''$, which is impossible by Proposition 1.7, since otherwise we would have $g \not\equiv_{a \not\simeq b \vee Q} g'$. Therefore $g'' \equiv_Q g'$ for every constant g in D, and $D'' \equiv_Q D'$. Since $D' \subseteq Q$, this entails that $D'' \leq_{\mathrm{I}} Q$.
The paramodulation of $c' \simeq d' \vee P'_2$ into $Q'_1 \vee D$ generates a clause $E_{\downarrow}$ of the form $(Q'_1 \vee D'' \vee F \vee P'_2 \vee \cdots \vee P'_2)_{\downarrow}$, where F is a disjunction of disequations of one of the following forms:
— $g \not\simeq c'$ with $g \equiv_Q a$ and $g \not\equiv_Q g'$,
— $g \not\simeq d'$ with $g \equiv_Q b$ and $g \not\equiv_Q g'$.
Since $c' \equiv_Q c \equiv_Q a$ and $d' \equiv_Q d \equiv_Q b$, it is clear that $F \leq_{\mathrm{I}} Q$. Since $P'_2 \leq_{\mathrm{I}} Q$, $Q'_1 \leq_{\mathrm{I}} Q$ and $D'' \leq_{\mathrm{I}} Q$, by Proposition 2.12 the clause $E_{\downarrow}$ can be factorized into a clause $F' \leq_{\mathrm{I}} Q_{\downarrow}$, because $E_{\downarrow} \leq_{\mathrm{E}} Q_{\downarrow}$. By Lemma 2.14 $S \leq_{\mathrm{I}} F'$, hence $S \leq_{\mathrm{I}} Q_{\downarrow}$.

— Assume that S contains no clause P'_1 such that $P'_1 \leq_{\mathrm{I}} a \not\simeq b \vee Q$ and $\equiv_{P'_1} \subseteq \equiv_Q$. Since $S \leq_{\mathrm{I}} S'$, the set S must contain a clause $P'_1 \leq_{\mathrm{I}} a \not\simeq b \vee P_1$, and we may assume that $\equiv_{P'_1} \not\subseteq \equiv_{P_1}$ (otherwise we would have $\equiv_{P'_1} \subseteq \equiv_Q$ and $P'_1 \leq_{\mathrm{I}} a \not\simeq b \vee Q$, hence we would be back in the previous case). P'_1 is of the form $\bigvee_{i=1}^{n}(e_i \not\simeq f_i) \vee P''_1$, where $\equiv_{P''_1} \subseteq \equiv_{P_1}$, and for every $i \in [1, n]$, we have $e_i \equiv_{a \not\simeq b \vee P_1} f_i$ but $e_i \not\equiv_{P_1} f_i$. By Proposition 1.7, we know that for every $i \in [1, n]$, we have either $e_i \equiv_{P_1} a$ and $f_i \equiv_{P_1} b$ or $e_i \equiv_{P_1} b$ and $f_i \equiv_{P_1} a$. By commutativity, we may assume that we always have $e_i \equiv_{P_1} a$ and $f_i \equiv_{P_1} b$. The rule `M` applied to P'_1 and $c' \simeq d' \vee P'_2$ generates $R_{\downarrow} = (\bigvee_{i=1}^{n}(e_i \not\simeq c' \vee d' \not\simeq f_i) \vee P''_1 \vee P'_2)_{\downarrow}$. We have $c' \equiv_{P_2} c$ and $c \equiv_{a \not\simeq c} a$ hence, by Proposition 1.7, for all $i \in [1, n]$, $c' \equiv_{a \not\simeq c \vee P_1 \vee P_2} e_i$ and $c' \equiv_Q e_i$. Similarly, for every $i \in [1, n]$, $d' \equiv_{b \not\simeq d \vee P_1 \vee P_2} f_i$ hence $d' \equiv_Q f_i$. Since $P''_1 \leq_{\mathrm{I}} a \not\simeq b \vee P_1$ and $P'_2 \leq_{\mathrm{I}} P_2$ by definition, we deduce that $R \leq_{\mathrm{I}} a \not\simeq b \vee Q$. Furthermore, since $\equiv_{P''_1} \subseteq \equiv_{P_1}$, we deduce that $\equiv_R \subseteq \equiv_Q$. Since S is $\mathcal{K}$-saturated, S contains a clause $R' \leq_{\mathrm{I}} R_{\downarrow}$. We have $\equiv_{R'} \subseteq \equiv_Q$ and $R' \leq_{\mathrm{I}} a \not\simeq b \vee Q$ as in the previous case, a contradiction. ∎

The completeness of the $\mathcal{K}$-paramodulation calculus is thus proved. It can be used in a saturation algorithm to generate all the prime implicates of an equational formula.

2 Rewriting beforehand with unordered paramodulation

As shown by the experiments (see Chapter 10), an analysis of the sets of prime implicates generated by $\mathcal{K}$-paramodulation suggests an opportunity of improving this calculus through a better handling of atomic implicates. For example the $\mathcal{K}$-saturation of the formula $S = \{a \simeq b, a \not\simeq c \vee b \simeq d\}$ results in the set $S' = \{a \simeq b, a \not\simeq c \vee a \simeq d, a \not\simeq c \vee b \simeq d, b \not\simeq c \vee a \simeq d, b \not\simeq c \vee b \simeq d\}$. All the added implicates are variants of the original $a \not\simeq c \vee b \simeq d$ where a and b are replaced by each other. In particular the prime implicate $b \not\simeq c \vee a \simeq d$ is generated at least twice, a first time through the derivation $S \vdash_{\mathcal{K}} a \not\simeq c \vee a \simeq d \vdash_{\mathcal{K}} b \not\simeq c \vee a \simeq d$ and a second time with $S \vdash_{\mathcal{K}} b \not\simeq c \vee b \simeq d \vdash_{\mathcal{K}} b \not\simeq c \vee a \simeq d$. These redundant computations are a major drawback of the $\mathcal{K}$-paramodulation calculus [2]. Exploiting this observation, we devised an efficient way of handling atomic implicates based on the rewriting of atomic implicates. Obviously, a most natural and efficient way of handling an equation $a \simeq b$ is to uniformly replace one of the terms, say b, by the other, a, thus yielding a simpler problem. It is clear that this operation preserves satisfiability, because a formula $S \wedge a \simeq b$ is satisfiable iff $S[a/b]$ is. Moreover, since $S \wedge a \simeq b \equiv S[a/b] \wedge a \simeq b$, the prime implicates of both sets are the same, and can be computed from the prime implicates of $S[a/b]$. The proposed method extend this principle to all the atomic prime implicates of S. It proceeds in three steps:

1. finding all the atomic prime implicates (set A) of the formula S and use them to rewrite S into the simpler S' such that $S \equiv A \cup S'$ and no constant b appears in S' where there exists an atom $b \simeq a \in A$ and $b \succ a$;
2. using the $\mathcal{K}$-paramodulation calculus to generate all the prime implicates of S';
3. reconstructing the set of prime implicates of S from the $\mathcal{K}$-saturation of S' (and A).

Section 2.1 presents the calculus that we devised to perform the first point of the method and Section 2.2 exposes the principles underlying the third point. The second point of this method is the subject of the previous chapter and we do not go back to it in the current one.

2.1 Atomic implicate generation

Although the $\mathcal{K}$-paramodulation calculus generates all atomic prime implicates of a formula in $\mathbb{E}_0$, it cannot be used to perform the first step of this new

2. This is verified experimentally, as exposed in Part III.

method, because there is no guarantee that the atomic prime implicates are generated in priority. Hence with $\mathcal{K}$-paramodulation, to ensure that all atomic prime implicates are generated, it would be necessary to wait for the whole saturation to be complete, rendering the whole approach irrelevant. Instead, we chose to consider unordered paramodulation with the selection of negative literals. This calculus is not fully deductive-complete, for example it is not possible to deduce $b \not\simeq c \vee a \simeq d$ from $a \simeq b$ and $c \simeq d$ using it, but we show in this section that it does generate all atomic prime implicates of a formula, hence fitting our requirements nicely.

Definition 2.16 The *unordered paramodulation* calculus is defined by the following rules, where the selection function *sel* is such that an arbitrary negative literal is selected if there is one in the considered clause and otherwise selects an arbitrary literal (necessarily positive), and where the parent clauses are in normal form and the consequents are systematically normalized:

$$\text{Resolution } (\mathtt{R}): \quad \frac{u \simeq v \vee C \qquad u \not\simeq v \vee D}{C \vee D},$$

where C is positive and $u \not\simeq v = sel(u \not\simeq v \vee D)$;

$$\text{Positive Paramodulation } (\mathtt{P}): \quad \frac{u \simeq v \vee C \qquad u \simeq v' \vee D}{v \simeq v' \vee C \vee D},$$

where C and D are positive. ◊

Definition 2.17 Let S be a formula. S is $\mathcal{U}$-saturated if it is saturated by unordered paramodulation up to redundancy ◊

Compared to the superposition calculus, described Section 2 of Chapter ii, nearly all the constraints are lifted or relaxed to adapt to the specificities of the selection function of unordered paramodulation. The two missing rules, namely Factorization and Reflexion, are subsumed by the normalization of consequents. This calculus is slightly more restrictive than what is commonly called unordered paramodulation [66]. The Negative Paramodulation rule is replaced by the Resolution rule. This restriction is not possible outside of $\mathbb{E}_0$ because replacements in subterms must be allowed in negative literals, e.g. in $\mathbb{E}_1$, to generate $c \simeq d$ from $f(a) \not\simeq f(b) \vee c \simeq d$ and $a \simeq b$. However, in $\mathbb{E}_0$, the Resolution rule is enough, given that the ordering constraints are indeed lifted.

Example 2.18 Let $C = c \not\simeq a \vee d \simeq a$ and $D = c \simeq a \vee d \simeq a$ be two clauses, where $a \prec b \prec c \prec d$. From C and D, the rule R generates $d \simeq a$, one of the atomic prime implicates of the formula $S = \{C, D\}$. Let us consider the saturation of S by three different calculi:

1. superposition,
2. unordered paramodulation as defined here,
3. the usual definition of unordered paramodulation with Factorization and Reflexion, and Negative Paramodulation instead of resolution.

Let S_i where $i \in \{1, 2, 3\}$ denote the saturation of S by the ith calculus. Then $S_1 = \{C, D\}$, $S_2 = \{C, D, d \simeq a\}$ and $S_3 = \{C, D, c \not\simeq d \vee d \simeq a \vee c \simeq a, c \not\simeq d \vee c \simeq a, d \simeq a, a \not\simeq a \vee d \simeq a, \ldots\}$. The set S_1 does not contain $a \simeq d$, illustrating that superposition is not deductive complete for atomic implicates. In contrast, S_3 contains this atom, but it also contains numerous unneeded clauses like $c \not\simeq d \vee c \simeq a$. This third calculus, although deductive-complete for atomic prime implicates, is not as well fitted to the task of generating them as the second calculus. ♣

Remark 2.19 *Note that in [77] calculi are introduced with the equivalent purpose of generating a set of atomic implicates that entail all atomic implicates of a satisfiable formula. In particular, in Theorem 5 of [77], the superposition calculus is shown to have this property. The major difference with the situation described here is that this theorem focuses solely on Horn clauses (clauses containing at most one positive literal). In fact, Superposition also does not fit our needs as illustrated by the previous example (where D is non-Horn).*

Deductive completeness restricted to atomic prime implicates

To show that all atomic prime implicates of a formula S are indeed generated by unordered paramodulation (Theorem 2.20), we focus the attention on a nondescript atom $a \simeq b$.

Theorem 2.20 *Let S be a satisfiable $\mathcal{U}$-saturated formula. If $b \neq a$ and $S \models b \simeq a$ then $b \simeq a \in S$.*

To prove this theorem, several notions and preliminary results are introduced. All of them are related to the atomic implicate $a \simeq b$ considered in the theorem, even though they do not always appear as parameters so as to lighten the notations. The proof is done by contraposition. Thus until then we assume given a formula S that is $\mathcal{U}$-saturated and such that $a \simeq b \notin S$. We show that $S \not\models a \simeq b$ by exhibiting an interpretation $\mathcal{I}$ such that $\mathcal{I} \models S$ and $\mathcal{I} \not\models a \simeq b$. We use I-subsumption as the redundancy criterion, along with the associated notion of projection.

We assume given a total ordering $\prec_{ab}$ on terms such that for all $t \in \Sigma_0 \backslash \{a, b\}$, $a \prec_{ab} b \prec_{ab} t$. The order $\prec_{ab}$ is extended to literals so that every negative literal is greater than every atom and atoms are compared using the lexicographic order. It is also extended to clauses as usual. For literals and clauses, it is denoted by $<_{ab}$. A partial order on clauses $\lhd_{ab}$ is also defined:

Definition 2.21 Let C and D be two clauses. $C \lhd_{ab} D$ iff
- $|C^-| < |D^-|$
- or $|C^-| = |D^-|$ and $C^+ <_{ab} D^+$ ◊

Proposition 2.22 *Let C and D be two clauses in normal form such that $C \neq D$. If $C \leq_{\mathrm{I}} D$ then $C \lhd_{ab} D$.*

PROOF. If $C \leq_{\mathrm{I}} D$ then $\equiv_C \subseteq \equiv_D$ thus $|C^-| \leq |D^-|$. In the case when $|C^-| = |D^-|$, $\equiv_C = \equiv_D$, thus for all $l \in C^+$, $l = l_{\restriction D} \in D$ and since $C \neq D$, $C^+ <_{ab} D^+$ ■

Let $p_0 <_{ab} \cdots <_{ab} p_n$ be an enumeration of all the atoms (thus $p_0 = a \simeq a$, $p_1 = b \simeq a$, etc). We define a set of ground rewrite rules and the equational interpretation that we use in the proof of Theorem 2.20 by induction on p_k.

Definition 2.23 Let $R_1(S) = \emptyset$ and $\mathcal{I}_{R_1} = \emptyset$. For all $k \in \{2 \dots n\}$, $R_k(S)$ is such that:

— $R_k(S) = R_{k-1}(S) \cup \{u \to v\}$ with $p_k = u \simeq v$ iff
  — $\exists D \vee p_k \in S$ with $D <_{ab} p_k$ such that $\mathcal{I}_{R_{k-1}}(S) \not\models D$,
  — and p_k is irreducible by $R_{k-1}(S)$,
— $R_k(S) = R_{k-1}(S)$ otherwise,

and $\mathcal{I}_{R_k}$ is the reflexive, commutative and transitive closure of $R_k(S)$. We denote by $R(S)$ the final set $R_n(S)$ and by $\mathcal{I}_R(S)$ the reflexive, commutative and transitive closure of $R(S)$. $\mathcal{I}_R(S) (= \mathcal{I}_{R_n}(S))$ is an equational interpretation. ◊

Definition 2.24 Let R be a set of ground rewrite rules and $u_1, \dots, u_n$ with $n \geq 2$ be constants. We call the tuple $(u_1, \dots, u_m)$ a *chain* in R iff for all $i \in \{1 \dots m-1\}$, either $\{u_i \to u_{i+1}\} \in R$ or $\{u_{i+1} \to u_i\} \in R$ and the u_i are pairwise distinct. A *link* is a triplet of constants (u_i, u_{i+1}, u_{i+2}) that belongs in a chain of size $m \geq 3$. We distinguish three types of link:

— local maximum, where $\{u_{i+1} \to u_i\} \in R$ and $\{u_{i+1} \to u_{i+2}\} \in R$,
— local minimum, where $\{u_i \to u_{i+1}\} \in R$ and $\{u_{i+2} \to u_{i+1}\} \in R$,
— and monotone link for the two other configurations possible. ◊

Proposition 2.25 *All the chains in $R(S)$ have no links that are monotone or local maxima.*

PROOF. If there is a local maximum in a chain of $R(S)$, then there exists $w \prec_{ab} v \prec_{ab} u$ such that $\{u \to v\} \in R$ and $\{u \to w\} \in R$. There exist i,j such that $p_i = u \simeq w$ and $p_j = u \simeq v$. Since $p_i <_{ab} p_j$, $\{u \to w\} \in R_{j-1}(S)$ thus $u \simeq v$ is reducible by $R_{j-1}(S)$ hence $\{u \to v\} \notin R$, a contradiction. The same reasoning applies for a monotone link because it implies that $\{u \to v\} \in R$ and $\{v \to w\} \in R$ with $w \prec_{ab} v \prec_{ab} u$. ■

Corollary 2.26 *For all constants u, v such that $v \prec_{ab} u$ and there exists a chain $(u = u_1, \dots, v = u_m)$ in $R(S)$, one of the following propositions holds:*

— *$m = 2$,*
— *$m = 3$ and (u_1, u_2, u_3) is a local minimum.*

Proposition 2.27 $\forall k \in \{1 \dots n\}$, $\forall C <_{ab} p_k$, $\mathcal{I}_{R_{k-1}}(S) \models C \Leftrightarrow \mathcal{I}_R(S) \models C$

PROOF. The direct implication is trivial because $R_{k-1} \subseteq R$. The reverse implication is proved by contradiction. If there is a clause $C <_{ab} p_k$ such that $\mathcal{I}_{R_{k-1}}(S) \not\models C$ and $\mathcal{I}_R(S) \models C$ then at least one of the literals of C has the

same property. Let us consider one such literal $l = u \simeq v$ with $v \prec_{ab} u$ (l cannot be negative since $l <_{ab} p_k$). Since $\mathcal{I}_R(S) \models l$, there exists a chain in $R(S)$ from u to v. By Corollary 2.26, there are only two kinds of chains to consider (u, v) or (u, x, v) with $x \prec_{ab} v$. In the first case $\{u \to v\} \in R(S)$, and since $u \simeq v <_{ab} p_k$, $\{u \to v\} \in R_{k-1}(S)$ thus $\mathcal{I}_{R_{k-1}}(S) \not\models l$, a contradiction. In the second case $\{u \to x\}, \{v \to x\} \in R(S)$ and again $u \simeq x, v \simeq x <_{ab} p_k$ thus the same contradiction is reached. ∎

We now have all the necessary elements to prove Theorem 2.20.

Proof. (of Theorem 2.20) Recall that we assume $b \simeq a \notin S$ to show that $S \not\models b \simeq a$. To do so, we use the equational interpretation $\mathcal{I}_R(S)$, from Definition 2.23, that we rename $\mathcal{I}$ and show that $\mathcal{I} \models S$ and $\mathcal{I} \not\models b \simeq a$.

— **$\mathcal{I} \not\models b \simeq a$.**
If $\mathcal{I} \models b \simeq a$ then by Corollary 2.26:
  — either $\{b \to a\} \in R$, which is impossible since $b \simeq a = p_1$ and $R_1(S) = \emptyset$ by construction,
  — or there exists a constant $x \prec_{ab} a, b$ such that (a, x, b) is a chain in R, which is also impossible since a is the minimal constant by $\prec_{ab}$.

— **$\mathcal{I} \models S$.**
Assume $\mathcal{I} \not\models S$ and let D be the smallest clause in S by $\lhd_{ab}$ such that $\mathcal{I} \not\models D$. The satisfiability of S ensures that $D \neq \Box$, so let $D = l \vee D'$ with $l = sel(D)$.
  — If $l = u \not\simeq v$, then $\mathcal{I} \models u \simeq v$ since $\mathcal{I} \not\models D$ (and for the same reason $u \simeq v \neq b \simeq a$). By Corollary 2.26, two cases must be considered.
    — Either $\{u \to v\} \in R$, thus $\exists C \vee u \simeq v \in S$ such that $C <_{ab} u \simeq v$ and by Proposition 2.27, $\mathcal{I} \not\models C$. Then R applied on S generates $(D' \vee C)_{\downarrow}$ and since S is saturated up to redundancy, there exists a clause $E \in S$ such that $E \leq_{\mathrm{I}} D' \vee C$. By Proposition 2.22, $E \lhd_{ab} D' \vee C$ and $D' \vee C \lhd_{ab} D$ because C is a positive clause and $|D'^{-}| = |D^{-}| - 1$. Thus $\mathcal{I} \models E$ leading to $\mathcal{I} \models D' \vee C$, a contradiction.
    — Or there exists a constant $x \prec_{ab} v$ such that $\{u \to x\}, \{v \to x\} \in R$. Thus $\exists C_u \vee u \simeq x, C_v \vee v \simeq x \in S$ such that $C_u <_{ab} u \simeq x$, $C_v <_{ab} v \simeq x$ and $\mathcal{I} \not\models C_u, C_v$. The rule P applied on S generates $(u \simeq v \vee C_u \vee C_v)_{\downarrow}$ and as a consequence, there exists a clause $E \in S$ such that $E \leq_{\mathrm{I}} u \simeq v \vee C_u \vee C_v$. Since C_u and C_v are positive, the clause E also verifies $E \subseteq u \simeq v \vee C_u \vee C_v$ and since $E \lhd_{ab} D$, $\mathcal{I} \models E$. If $E \subseteq C_u \vee C_v$ then $\mathcal{I} \models C_u \vee C_v$ which contradicts their definition, thus $E = u \simeq v \vee E'$ with $\mathcal{I} \not\models E'$ for the same reason. Applied on E and D in S, R generates $(E' \vee D')_{\downarrow}$, thus there exists a clause $F \in S$ such that $F \leq_{\mathrm{I}} D' \vee E'$ and a contradiction is reached by the same means as in the previous point (since $D' \vee E' \lhd_{ab} D$).
  — If $l = u \simeq v$ with $v \prec_{ab} u$, then $l > D'$ and there exists $k \in 2 \dots n$ such that $u \simeq v = p_k$ (note that $k \neq 1$ because otherwise $a \simeq b \in S$). Since $\mathcal{I} \not\models D'$, necessarily $u \simeq v$ is reducible by $R_{k-1}(S)$ and thus

by R. Hence $\{u \to w\} \in R$ or $\{v \to x\} \in R$ with $w, x \prec_{ab} v$. We assume $\{u \to w\} \in R$ but the reasoning is similar in the other case. By definition of R, there exists a clause $u \simeq w \vee C \in S$ such that $C <_{ab} u \simeq w$ and by Proposition 2.27, $\mathcal{I} \not\models C$. Applied on S, P generates $(v \simeq w \vee C \vee D')_{\downarrow}$, thus there exists a clause $E \in S$ such that $E \leq_{\mathrm{I}} v \simeq w \vee C \vee D'$. Since $v \simeq w \vee C \vee D' \lhd_{ab} u \simeq v \vee D'$, we have $E \lhd_{ab} D$, hence by minimality of D, $\mathcal{I} \models E$ and since $\mathcal{I} \not\models C \vee D'$, necessarily $\mathcal{I} \models v \simeq w$. Because $\mathcal{I} \models u \simeq w$, by transitivity $\mathcal{I} \models u \simeq v$, thus $\mathcal{I} \models D$, a contradiction. ∎

2.2 Recovery of the original solution

By applying the first two steps of this method, we have:

1. extracted the set of atomic prime implicates A of a formula S by unordered paramodulation and rewritten S into the formula S' such that for all atoms $a \simeq b \in A$ with $b \succ a$, b does not appear into S' and $S \equiv A \cup S'$, and
2. generated the set T of prime implicates of S' by $\mathcal{K}$-paramodulation.

In the third step, we recover the set of prime implicates of the original formula S from T and A. From a practical point of view, this step is arguably needed. An application that requires the prime implicates of S could very well be satisfied by simply recovering A and T, not being interested in all the 'variants' added by this step. Nevertheless, for the sake of completing the method, an efficient technique of prime implicate recovery was also investigated and we found out that it is possible to recover the prime implicates of S by applying $\mathcal{K}$-paramodulation inferences on T and A. The key point, which ensures the efficiency of this approach, is that it is not necessary to apply any inference between the newly generated clauses: only the inferences involving A need to be considered. Formally, what renders this method efficient is that all the implicates of a set of clauses $S \cup \{b \simeq a\}$ (with $a \prec b$) are I-subsumed by clauses recursively obtained by $\mathcal{K}$-paramodulation between $a \simeq b$ and the prime implicates of $S[a/b]$.

Definition 2.28 Given two constants a and b, we denote by $\mathcal{K}^{a \simeq b}$-paramodulation the following restriction of the $\mathcal{K}$-paramodulation calculus.

$$\text{Paramodulation } (\mathtt{P}^{+}): \quad \frac{a' \simeq c \vee C \qquad a \simeq b}{a \not\simeq a' \vee b \simeq c \vee D}$$

$$\text{Negative Multi-Paramodulation } (\mathtt{M}): \quad \frac{\bigvee_{i=1}^{n}(a_i \not\simeq b_i) \vee C \qquad a \simeq b}{\bigvee_{i=1}^{n}(a_i \not\simeq a \vee b \not\simeq b_i) \vee C}$$

As in $\mathcal{K}$-paramodulation, the premises are assumed to be in normal form and the consequent is systematically normalized. The rule F does not appear because it cannot be applied on $a \simeq b$. The saturation of a formula S by $\mathcal{K}^{a \simeq b}$-paramodulation is denoted $S_{\vdash a \simeq b}$. ◊

Proposition 2.29 *Let S be a set of clauses. The set $S_{\vdash a \simeq b}$ has the following properties:*

- $S \cup \{a \simeq b\} \models S_{\vdash a \simeq b}$,
- $S_{\vdash a \simeq b} \leq_{\mathrm{I}} S \cup \{a \simeq b\}$,
- $\forall D \in S_{\vdash a \simeq b}$, *if D' is deducible by one application of* $\mathtt{P}^+$ *or* $\mathtt{M}$ *on D and $a \simeq b$, then $S_{\vdash a \simeq b} \leq_{\mathrm{I}} D'$.*

To prove the consistency of this method (Lemma 2.33), some new definitions and results need to be introduced. First we define a notion of distance to a clause.

Definition 2.30 We define $\Delta_D(C)$, the *distance* of the clause C to the clause D, both non-tautological, by $\Delta_D(C) = (x, y)$ such that:

- $x = \begin{cases} 0 & \text{if } C^- \leq_{\mathrm{I}} D, \\ 1 & \text{otherwise.} \end{cases}$
- $y = |\{u \simeq v \in C \mid u \simeq v \not\leq_{\mathrm{I}} D\}|$

Distances are compared using the lexicographic order. ◊

The following results about the distance are used in the proof of Lemma 2.33 and emphasize that this distance measures how far from being in an entailment relation two clauses are.

Proposition 2.31 *Let C, C' and D be three clauses.*

1. *If $\Delta_D(C) = (0, 0)$ then $C \models D$.*
2. *If $C' \leq_{\mathrm{I}} C$ and $\Delta_D(C) = (0, y)$ then $\Delta_D(C') \leq \Delta_D(C)$.*

Proof. We prove the two properties separately.

1. The first point is a direct consequence of the definition of the distance.
2. To prove the second point, we write $\Delta_D(C') = (x', y')$. If $C' \leq_{\mathrm{I}} C$ then $C'^- \leq_{\mathrm{I}} C^-$. Since $\Delta_D(C) = (0, y)$, by definition we have $C^- \leq_{\mathrm{I}} D$, which implies $C'^- \leq_{\mathrm{I}} D$, thus $x' = 0$. To estimate the value of y', let us consider the literals $u' \simeq v' \in C'$ such that $u' \simeq v' \not\leq_{\mathrm{I}} D$. Since $C' \leq_{\mathrm{I}} C$, there exists an injective mapping from all literals $u' \simeq v'$ to literals $u \simeq v \in C$ such that $(u' \simeq v')_{\downharpoonright C} = u \simeq v$. Moreover, since $C^- \leq_{\mathrm{I}} D$, we also have $(u' \simeq v')_{\downharpoonright D} = (u \simeq v)_{\downharpoonright D}$, thus $u \simeq v \not\leq_{\mathrm{I}} D$. Therefore all literals $l' \not\leq_{\mathrm{I}} D$ of C' correspond to pairwise distinct literals $l \not\leq_{\mathrm{I}} D$ of C, and so $y' \leq y$. Hence the relation $\Delta_D(C') \leq \Delta_D(C)$. ∎

Proposition 2.32 *Let D, D' be two clauses and a, b be two constants with $a \prec b$ such that $a \simeq b \not\models D$ and $D' \leq_{\mathrm{I}} D_0 = D \vee a \not\simeq b$. If there exists a literal $u' \simeq v' \in D'$ such that $u' \simeq v' \not\leq_{\mathrm{I}} D$, then there exists $u \simeq v \in D$ such that:*

$$\begin{cases} u \equiv_{D_0} u' \\ v \equiv_{D_0} v' \end{cases} \quad \textit{and necessarily one of the following holds:}$$

— $\begin{cases} u \not\equiv_D u' \\ v \equiv_D v' \end{cases}$ *and* $\{u_{\lfloor D}, u'_{\lfloor D}\} = \{a_{\lfloor D}, b_{\lfloor D}\}$ *or,*

— $\begin{cases} u \equiv_D u' \\ v \not\equiv_D v' \end{cases}$ *and* $\{v_{\lfloor D}, v'_{\lfloor D}\} = \{a_{\lfloor D}, b_{\lfloor D}\}$.

Proof. Since $D' \leq_{\text{I}} D_0$ and $u' \simeq v' \in D'$, there exists a literal $u \simeq v \in D_0$ such that $u \equiv_{D_0} u'$ and $v \equiv_{D_0} v'$. Furthermore, by definition of D_0, $u \simeq v \in D$. By contradiction:

— If $u \equiv_D u'$ and $v \equiv_D v'$ then $u' \simeq v' \leq_{\text{I}} D$ which raises a contradiction.

— If $u \not\equiv_D u'$ and $v \not\equiv_D v'$ then, since $u \equiv_{D_0} u'$ and $v \equiv_{D_0} v'$, either $\begin{cases} u' \equiv_D a \\ v' \equiv_D b \end{cases}$ or $\begin{cases} u' \equiv_D b \\ v' \equiv_D a \end{cases}$ and in both cases $a \simeq b \models D$ (since $u' \simeq v' \models D \vee a \neq b$), or $u' \equiv_D v' \equiv_D x$ with $x \in \{a, b\}$ and then D_0 is a tautology, meaning again that $a \simeq b \models D$. As in the previous case, a contradiction is reached.

In addition, if any two constants x and x' are such that $x \equiv_{D_0} x'$ and $x \not\equiv_D x'$ then it can be deduced that $\{x_{\lfloor D}, x'_{\lfloor D}\} = \{a_{\lfloor D}, b_{\lfloor D}\}$. ∎

Lemma 2.33 *Let S be a set of clauses, $a \simeq b$ be a literal such that $a \prec b$ and $S' = (PI(S[a/b])_{\vdash a \simeq b}$ where $PI(S[a/b])$ is the set of prime implicates of $S[a/b]$. Let D be a clause such that $S \cup \{a \simeq b\} \models D$, then $S' \leq_{\text{I}} D$.*

Proof. Let us consider $D' \in S'$, a clause such that $D' \leq_{\text{I}} D_0 = D \vee a \not\simeq b$ and $\Delta_D(D') = \min_{\{X \in S' \mid X \leq_{\text{I}} D_0\}} \Delta_D(X)$. Note that $D[b/a]$ is an implicate of $S[b/a]$, thus $PI(S[b/a])$ necessarily contains a clause X I-subsuming $D[b/a]$, hence I-subsuming $D \vee a \not\simeq b$. Consequently, $\{X \in S' \mid X \leq_{\text{I}} D \vee a \not\simeq b\}$ is not empty by definition of S', which ensures the existence of D'.

— If $\Delta_D(D') = (0, 0)$ then, by Proposition 2.31(1), $D' \models D$ and since $D' \leq_{\text{I}} D_0$ we have $D' \leq_{\text{I}} D$.

— If $\Delta_D(D') = (0, y)$, with $y \geq 1$, then there exists a positive literal $u \simeq v \in D'$ such that $u \simeq v \not\leq_{\text{I}} D$, and since $u \simeq v \leq_{\text{I}} D_0$, there exists a positive literal $u' \simeq v' \in D$ such that $u \equiv_{D_0} u'$ and $v \equiv_{D_0} v'$. Based on Proposition 2.32, we assume w.l.o.g. that $u \not\equiv_D u'$ and $v \equiv_D v'$, which entails $\{u_{\lfloor D}, u'_{\lfloor D}\} = \{a_{\lfloor D}, b_{\lfloor D}\}$. We assume $(u_{\lfloor D}, u'_{\lfloor D}) = (a_{\lfloor D}, b_{\lfloor D})$, the other possibility being completely symmetrical (indeed the only difference between a and b, the fact that $a \prec b$, plays no role in the proof). By rule $\mathtt{P}^+$ with parents D' and $a \simeq b$, we generate $E_{\downarrow} = (u \not\simeq a \vee b \simeq v \vee D'')_{\downarrow}$. We know that $E \leq_{\text{I}} D_0$, since

  — $E^- \leq_{\text{I}} D^- \leq_{\text{I}} D' \vee a \not\simeq b$ because $u \not\simeq a \leq_{\text{I}} D^- (= D''^-)$,

  — and $E^+ \leq_{\text{I}} D_0$, because for every literal $l' \in E^+ \setminus \{b \simeq v\}$ we have $l' \in D'$ hence the existence of distinct positive literals in D_0 (and thus in D) associated to these literals l', and because $b \simeq v$ can be associated to $u' \simeq v'$ since $u' \equiv_D b$ and $v' \equiv_D v$.

  Note that $\{o \simeq p \in E \mid o \simeq p \not\leq_{\text{I}} D\} = \{o \simeq p \in D' \mid o \simeq p \not\leq_{\text{I}} D\} \setminus \{u \simeq v\}$, thus $\Delta_D(E) < \Delta_D(D')$. Moreover, by definition of S', there

exists a clause $E' \in S'$ such that $E' \leq_{\text{I}} E_{\downarrow}$. Since $\Delta_D(E) = (0, y'')$, by Proposition 2.31(2), $\Delta_D(E') \leq \Delta_D(E_{\downarrow}) \leq \Delta_D(E)$. We have $E' \in \{X \in S' \mid X \leq_{\text{I}} D \vee a \not\simeq b\}$ and $\Delta_D(E') < \Delta_D(D')$. Consequently $\Delta_D(D')$ is not minimal which contradicts the definition of D', so this case cannot happen.

— If $\Delta_D(D') = (1, y)$, we can write D' as $D' = \bigvee_{i=1}^{n} u_i \not\simeq v_i \vee D''$, with for all $i \in \{1..n\}$, $u_i \not\equiv_D v_i$ and for all $u \not\simeq v \in D''$, $u \equiv_D v$. Since for every $i \in \{1..n\}$, $u_i \equiv_{D \vee a \not\simeq b} v_i$, we assume w.l.o.g that $u_i \equiv_D a$ and $v_i \equiv_D b$ (otherwise, a symmetrical result holds and we can simply swap u_i and v_i to obtain the desired result). We generate using `M` the clause $E_{\downarrow} = (\bigvee_{i=1}^{n} u_i \not\simeq a \vee v_i \not\simeq b \vee D'')_{\downarrow}$. The clause E is built such that $E^- \leq_{\text{I}} D^-$ and, since $D'' \leq_{\text{I}} D$, we have $E \leq_{\text{I}} D \vee a \not\simeq b$, thus $E_{\downarrow} \leq_{\text{I}} D \vee a \not\simeq b$. By definition, there exists $E' \in S'$ such that $E' \leq_{\text{I}} E_{\downarrow}$, hence $E'^- \leq_{\text{I}} D$ and $\Delta_D(E') = (0, y'')$. Since $E' \leq_{\text{I}} D \vee a \not\simeq b$ and $\Delta_D(E') < \Delta_D(D')$, there is also a contradiction in this case.

We have proven that the clause D' necessarily I-subsumes D under our hypotheses. Since $D' \in S'$, Lemma 2.33 is proven. ∎

Corollary 2.34 *For S, S' and $a \simeq b$ as in Lemma 2.33, $S' \equiv S \cup \{a \simeq b\}$ and S' is saturated up to redundancy.*

Proof. Direct consequence of Lemma 2.33 ∎

3 Rewriting on the fly

The idea that led to the development of the rewriting method described in the previous section — using atomic implicates to simplify the input problem through rewriting — can be exploited in a different way, which is the subject of this section. Instead of trying to first find all atomic implicates, then use them to preprocess the input formula before saturating it with $\mathcal{K}$-paramodulation, it is possible to perform the rewriting on the fly during the $\mathcal{K}$-saturation, as soon as an atomic implicate is generated. This amounts to merging the first two steps of the previous method:

1. using $\mathcal{K}$-paramodulation and rewriting combined to generate $A \cup S'$ from the input formula S, where A is a set of atomic implicates of S entailing all its atomic implicates, S' is $\mathcal{K}$-saturated and $S' \equiv S[A]$,
2. reconstructing the set of prime implicates of S from A and S'

Since the second step of this method is strictly identical to the third one of the previous method, we do not reexplain it. See Section 2.2 for more details.

3.1 High-level view

The first step of this method can be formalized simply by adding to the $\mathcal{K}$-paramodulation calculus the following rule:

$$\text{Rewriting } (\mathtt{Rw_{HL}}): \quad \frac{S \cup \{a \simeq b\}}{S[a/b]}$$

where $a \prec b$ and $a \simeq b$ is stored in a set A of atoms never used in further inferences.

This seems simple enough, but considered as is, the rule $\mathtt{Rw_{HL}}$ interferes with the usual saturation procedure presented in Definition ii.32. Indeed, clauses that are in the proceeded set are affected by the rewriting of clauses, which can enable inferences that could not be applied previously.

Example 2.35 Let $v \prec u \prec s \prec a \prec b \prec c \prec e \prec f$ be a set of ordered constants. Let $S = \{C, D, a \simeq b\}$ where $C = u \simeq v \vee e \simeq f$ and $D = a \not\simeq s \vee b \simeq c$. Starting from $\langle \emptyset; S \rangle$, we apply the inference generation rule of the saturation procedure on C and then on D. We obtain $\langle \{C, D\}; W \rangle$, where $\{a \simeq b, E\} \subseteq W$ and $E = u \not\simeq a \vee v \not\simeq s \vee b \simeq c \vee f \simeq e$. The clause E belong to W because it is generated by $\mathtt{M}$ with parents C and D (on $u \simeq v$ and $a \not\simeq s$ respectively). Assume the next chosen clause from W is $a \simeq b$. Then we apply the new rule $\mathtt{Rw_{HL}}$ to the whole set:

- C is unchanged,
- D becomes $D' = a \not\simeq s \vee s \simeq c$, (indeed, D is replaced by $D[a/b] = a \not\simeq s \vee a \simeq c$, which normalizes into D' — a is replaced by s in the positive part since $a_{\downarrow D[a/b]} = s$),
- W becomes $W' = W[a/b]_{\downarrow}$ and in particular E becomes $E' = u \not\simeq a \vee v \not\simeq s \vee u \simeq c \vee e \simeq f$ (after normalization).

The clauses C and D' stay in the processed set.

The problem here is that the clause $E'' = u \not\simeq a \vee v \not\simeq s \vee v \simeq c \vee e \simeq f$ that is the consequent of the inference rule $\mathtt{M}$ with parents C and D' on the literals $u \simeq v$ and $a \not\simeq s$ will never be generated because C and D' remain in P and thus will not be processed again. Note in particular the difference between E'' and E' in the positive literals, even though the "same" inference and rewriting steps were used to generate them, simply not in the same order. Because of this phenomenon, the procedure cannot be applied as is. ♣

3.2 Adaptation of the saturation procedure

Intuitively, the incompleteness of the saturation procedure can be understood in the following way. Due to the systematic normalization of clauses, the replacement of a constant b by another constant a can enable new inferences that were not applicable on the original clauses. Thus if this happens between clauses that are both located in the processed set, the new inferences are never performed. A straightforward way to recover completeness would be to transfer all the (modified) processed clauses back to the waiting set after each rewriting.

However, this would heavily impact the efficiency of the procedure. Nevertheless, some clauses need to be reprocessed and the problem is to determine which ones.

The following definition introduces a criterion that identifies clauses that do not need to be reprocessed.

Definition 2.36 A clause D is $\langle a, b\rangle$-*neutral* if $D^+[a/b] = (D[a/b]_\downarrow)^+$. $\Diamond$

This property means that the replacement of b by a does not affect the representatives of the equivalence classes occurring in the positive part of a clause, even if it contains a and b. It is called the *collision* criterion.

Using this collision criterion, the saturation procedure can be adapted to incorporate rewriting.

Definition 2.37 The saturation procedure with rewriting (denoted $\mathrm{Sat}_{R-\mathcal{K}}$ for the $\mathcal{K}$-paramodulation calculus) is formalized by the following rules, where $\langle P; W; A\rangle$ is a triplet composed of the processed set, the waiting set and a set of atoms.

$$\text{Redundancy elimination } (\mathtt{R}): \quad \frac{P; W \cup \{C\}; A}{P; W; A} \text{ (1.)},$$

$$\text{Inference generation } (\mathtt{I}): \quad \frac{P; W \cup \{C\}; A}{P' \cup \{C\}; W \cup P'_{\vdash^{1},C} \setminus \{C\}; A} \text{ (2.)},$$

$$\text{Atomic rewriting in } W \ (\mathtt{Rw_W}): \quad \frac{P; W \cup \{b \simeq a\}; A}{P'; (W[a/b])_\downarrow \cup P''; A \cup \{b \simeq a\}} \text{ (3.)},$$

$$\text{Atomic rewriting in } P \ (\mathtt{Rw_P}): \quad \frac{P \cup \{b \simeq a\}; W; A}{P'; (W[a/b])_\downarrow \cup P''; A \cup \{b \simeq a\}} \text{ (3.)},$$

with the following conditions:

1. if C is redundant with respect to P,
2. if P and $W \cup \{C\}$ contain no atomic clauses and if C is not redundant with respect to P; where $P' = \{D \in P \mid D \text{ is not redundant to } C\}$,
3. where $a \prec b$ and $P' = \left\{ D[a/b]_\downarrow \mid D \in P \wedge D \text{ is } \langle a, b\rangle\text{-neutral} \right\}$ and $P'' = P[a/b]_\downarrow \setminus P'$. $\Diamond$

The conditions of application of this procedure's rules ensure that as soon as an atom is generated (by inference generation or rewriting), it must be extracted and used in a rewriting step before any further inference generation step. The rule $\mathtt{Rw_P}$ is necessary because new atoms can appear in P after an inference by the rule $\mathtt{Rw_W}$ (e.g., $b \not\simeq a \vee c \simeq d \in P$ is rewritten in $c \simeq d$ if $b \simeq a$ appears in W)

The key property ensuring the completeness of the whole procedure is its *rewrite-stability*.

Definition 2.38 A rule is *rewrite-stable* if its parent $\langle P; W; A\rangle$ and consequent $\langle P'; W'; A'\rangle$ verify the following property:

$$\text{if } (P \cup W_{\mathtt{F}}) \leq_{\mathrm{I}} P_{\vdash 1} \text{ then } (P' \cup {W'}_{\mathtt{F}}) \leq_{\mathrm{I}} {P'}_{\vdash 1},$$

where $W_{\mathtt{F}}$ is the saturation of the set W up to redundancy by the rule `F` only (i.e. $W_{\mathtt{F}}$ contains all clauses deducible from W by a finite number of applications of the rule `F`, up to redundancy). A procedure is rewrite-stable if all its rules are rewrite-stable. ◊

Intuitively, this property means that all clauses that can be inferred from P (in one step) must be redundant w.r.t. P or W (or from a factor of W).

Theorem 2.39 *The saturation procedure with rewriting is rewrite-stable.*

Proof. The rewrite-stability of the rules `I` and `R` is trivial. That of the rules $\mathtt{Rw_P}$ and $\mathtt{Rw_W}$ is stated in a separate theorem, namely Theorem 2.54. Its proof is the subject of the following subsection. ∎

Remark 2.40 *The standard saturation procedure where P is fully emptied into W after each rewriting step is trivially rewrite-stable.*

Theorem 2.41 *To saturate and rewrite a formula S with $\mathrm{Sat}_{R-\mathcal{K}}$, the triplet $\langle \emptyset; S; \emptyset\rangle$ is input to the procedure. It outputs $\langle S'; \emptyset; A\rangle$ where S' contains no atomic clause, $S' = S'[A]$, $S \equiv S' \cup A$ and S' is saturated up to redundancy.*

Proof. Let $\langle P, W, A\rangle$ be the result of a $\mathrm{Sat}_{R-\mathcal{K}}$ rule. It is straightforward to verify that if $W \neq \emptyset$ or if P contains an atomic clause, then the procedure is not complete.

We show by induction on the derivation of $\mathrm{Sat}_{R-\mathcal{K}}$ that $S' = S'[A]$ and $S \equiv S' \cup A$. Let $\langle P, W, A\rangle$ be a triplet in the derivation considered. Our induction hypotheses are $S \equiv P \cup W \cup A$, $P[A] = P$ and $W[A] = W$. These properties are trivially true for $\langle \emptyset; S; \emptyset\rangle$. Now let us consider the triplet $\langle P'; W'; A'\rangle$ that comes in the derivation just after $\langle P; W; A\rangle$. Since the $\mathcal{K}$-paramodulation calculus is correct, all the rules of $\mathrm{Sat}_{R-\mathcal{K}}$ are such that $P \cup W \cup A \equiv P' \cup W' \cup A'$. In addition, since no rule of $\mathrm{Sat}_{R-\mathcal{K}}$ introduces new constants to the sets and since for atoms $b \simeq a$ where $a \prec b$ added to A, the constant b is immediately deleted from $P \cup W$, the results $P'[A'] = P$ and $W'[A'] = W'$ are clear.

Finally, to prove that S' is saturated, it suffices to invoke the rewrite-stability of $\mathrm{Sat}_{R-\mathcal{K}}$. Since all the rules of this procedure are rewrite-stable and since $\emptyset \cup S_{\mathtt{F}} \leq_{\mathrm{I}} \emptyset_{\vdash 1}$, we know that $S' \leq_{\mathrm{I}} {S'}_{\vdash 1}$. ∎

3.3 Rewrite-stability of $\mathrm{Sat}_{R-\mathcal{K}}$

The remainder of this section presents a proof of the rewrite-stability of the rules $\mathtt{Rw_P}$ and $\mathtt{Rw_W}$ of $\mathrm{Sat}_{R-\mathcal{K}}$. In this proof, we construct for each clause in ${P'}_{\vdash 1}$ the corresponding clause of $P' \cup {W'}_{\mathtt{F}}$ that I-subsumes it. We show how this clause is built from a clause in $P \cup W_{\mathtt{F}}$. To do so, a precise decomposition of

the concerned clauses is needed, different in each case. We start by defining and proving these decompositions, then we consider the clauses in ${P'}_{\vdash 1}$ which are consequents of an application of the rule F, then of the rule $\mathtt{P}^+$ and finally of the rule M. In each case, we first exhibit a clause $D \in P' \cup {W'}_{\mathtt{F}}$ such that its negative literals are equivalent after a rewriting step to the considered clause C in ${P'}_{\vdash 1}$. Then we use an $\langle a, b\rangle$-neutrality assumption to prove that D, after a rewriting step and normalization, I-subsumes the clause C.

To lighten the notations, we assume from this point on in all this subsection that a and b are constants such that $a \prec b$, and we use the convention that for i in $\{1, 2\}$, D_i and C_i are clauses in normal form such that $D_i[a/b] \equiv C_i$. We consider a fresh constant λ, and extend $\prec$ in such a way that for all other constants c, we have $c \prec \lambda$. Let us define, for $i \in \{1, 2\}$:

$$r_b(i) = b_{\downarrow D_i}, \qquad r_a(i) = a_{\downarrow D_i},$$

$$r(i) = \min\{r_a(i), r_b(i)\}, \quad \overline{r}(i) = \max\{r_a(i), r_b(i)\},$$

$$r'_b(i) = \begin{cases} \min([b]_{D_i} \setminus \{b\}), \text{ when } [b]_{D_i} \neq \{b\} \\ \lambda \quad \text{otherwise} \end{cases}$$

$$r'(i) = \min\{r'_b(i), r_a(i)\}, \quad \overline{r}'(i) = \max\{r'_b(i), r_a(i)\}.$$

Although these constants all depend on D_i, it will always be clear from the context which of D_1 and D_2 they refer to, thus the " (i) " will always be omitted. It can also be noted that the equality $r = r'$ is always true because either $r = r_a$ and $r' = r_a$, or $r = r_b$ thus $r_b \prec r_a \preceq a \prec b$, hence $r_b = r'_b \prec r_a$. Nevertheless, the two notations will be kept to distinguish the clauses before rewriting (containing r) and after rewriting (containing r'). Informally, r_a and r_b are the representatives of the normal form of a and b. After b is replaced by a, these normal forms are merged, thus the new representative of the normal class is the minimum of r_a and r_b (b excluded), e.g., r', while $\overline{r'}$ denotes the other representative.

In what follows, given a clause $B = \bigvee_{i=1}^{n} c_i \not\approx d_i$, we will use the following shorthand:

$$B_{u,v} \stackrel{\text{def}}{=} \bigvee_{i=1}^{n} c_i \not\approx u \vee d_i \not\approx v$$

For instance if $B = a \not\approx c \vee b \not\approx c$ then $B_{u,v} = a \not\approx u \vee b \not\approx u \vee c \not\approx v$ and if $B = \square$ then we let $B_{u,v} = \square$. We also use the following equivalence:

Proposition 2.42 *Given constants a, b, c, d, with $a \prec b$, and a clause D, we have $c[a/b] \equiv_{D[a/b]} d[a/b]$ if and only if $c \equiv_{D \vee a \not\approx b} d$.*

These many notations are introduced to build a generic decomposition of the clauses C_i and D_i where their relation $(D_i[a/b]_{\downarrow} = C_i)$ is explicitly apparent. This is done by isolating in C_i the negative literals belonging to the equivalence class of a, which correspond in D_i to the negative literals in the equivalence

class of either a or b (or both if they already belong to the same class in D_i). An optional literal is also introduced. It is a negative literal containing b. No such literal exists if the equivalent class of b is $\{b\}$ (the case where $r'_b = \lambda$). If such a literal exists then its form depends on whether b is identical to r_b. If not, then the only literal containing b is of the form $b \simeq r_b$ by definition of the normal form, otherwise it can be set to $r'_b \simeq b$, where r'_b is some element distinct from b in the equivalence class of b. The definition of r'_b has been tuned to cover both cases simultaneously (indeed, if $b \neq b_{\downarrow D_i}$ then necessarily $r'_b = r_b$). These decompositions are exposed in Proposition 2.43 and used in the subsequent propositions.

Proposition 2.43 *For $i = 1$ and $i = 2$, C_i and D_i can be decomposed in the following way. There is an $n \geq 0$, a negative clause C'_i and a positive clause C''_i such that C_i is of the form*

$$C_i = \bigvee_{j=1}^{n} (c_j \not\simeq r') \vee C'_i \vee C''_i \vee \begin{cases} \Box & \text{if } r'_b = \lambda \text{ or } r'_b = r_a, \\ r'_b \not\simeq r_a & \text{otherwise,} \end{cases}$$

and there exists a positive clause D''_i such that D_i is of the form

$$D_i = \bigvee_{j=1}^{n_1} (c_j \not\simeq r_b) \vee \bigvee_{j=n_1+1}^{n} (c_j \not\simeq r_a) \vee C'_i \vee D''_i \vee \begin{cases} \Box & \text{if } r'_b = \lambda, \\ b \not\simeq r'_b & \text{otherwise,} \end{cases}$$

where $0 \leq n_1 \leq n$. Furthermore, for all $c' \simeq d' \in C''_i$, there exists a literal $c \simeq d \in D''_i$ such that $c \simeq d \equiv_{C_i \vee a \not\simeq b} c' \simeq d'$, and for all $c \simeq d \in D''_i$, there exists a literal $c' \simeq d' \in C''_i$ such that $c' \simeq d' \equiv_{D_i \vee a \not\simeq b} c \simeq d$.

Proof. There are four cases to verify: If $b = r_b$ then $r_a (= r') \prec r_b \prec r'_b$ and if $b \neq r_b$, the remaining cases are $r_b (= r'_b = r') \prec r_a$, $r_a (= r') \prec r_b$ and $r_a = r_b (= r'_b = r')$. In all these cases, $c_1, \ldots, c_{n_1}$ are the constants that belong to $[b]_{D_i}$ (r'_b excepted) and $c_{n_1+1}, \ldots, c_n$ are those belonging to $[a]_{D_i}$. In the cases where $r'_b \neq \lambda$, we have $r_b \in \{b, r'_b\}$, thus $b \not\simeq r'_b$ belongs to D_i by definition of the normal form. Since $[a]_{C_i} = ([b]_{D_i} \cup [a]_{D_i}) \setminus \{b\}$, the constants $c_1, \ldots, c_n$ also belong to $[a]_{C_i}$. If $r'_b \neq \lambda$ and $r'_b \neq r_a$ then, since C_i is in normal form, depending on the case either $r' = r_a$ or $r' = r'_b$ which both mean that $r'_b \not\simeq r_a$ belongs in C_i since one of these constant is the representative of the other. Furthermore, if there exists a literal $c' \simeq d' \in C''_i$ such that for all $c \simeq d \in D''_i$, $c \simeq d \not\equiv_{C \vee a \not\simeq b} c' \simeq d'$, then $D_i[a/b]$ and C_i cannot be equivalent, a contradiction. The same reasoning holds $c \simeq d \in D''_i$. ∎

Proposition 2.44 *Let C be a clause generated from C_1 using the rule* F*, then there exists a clause D generated by* F *from D_1 such that $D[a/b]^- \equiv C^-$.*

Proof. Since C is generated from C_1 using F, we can write $C_1 = u' \simeq v' \vee s' \simeq t' \vee C'_1$ and $C = (u' \not\simeq s' \vee v' \not\simeq t' \vee u' \simeq v' \vee C'_1)_{\downarrow}$. Moreover, $D_1[a/b]_{\downarrow} = C_1$, thus by Proposition 2.43, D_1 can be written as $u \simeq v \vee s \simeq t \vee D'_1$, with

$x \equiv_{C'_1 \vee a \not\simeq b} x' \equiv_{D_1 \vee a \not\simeq b} x$, with $x \in \{u, v, s, t\}$. By applying $\mathtt{F}$ on D_1, we generate the clause $D = (u \not\simeq s \vee v \not\simeq t \vee u \simeq v \vee D'_1)_{\downarrow}$. Now that D is defined, we only have to ensure that $D[a/b]^- \equiv C^-$. To simplify the reasoning, we prove instead the equivalent result $D'[a/b]^- \equiv C'^-$ where the clauses $D' = u \not\simeq s \vee v \not\simeq t \vee u \simeq v \vee D'_1$ and $C' = u \not\simeq s \vee v \not\simeq t \vee u \simeq v \vee C'_1$ (not in normal form) are respectively equivalent to D and C. Let us consider the negative literals of D'_1. By definition, we have $D'_1[a/b]^- \equiv C'^-_1$. Since $C'^-_1 \leq_{\text{I}} C^-$, we have $u \not\simeq s \equiv_{C \vee a \not\simeq b} u' \not\simeq s'$ and $v \not\simeq t \equiv_{C \vee a \not\simeq b} v' \not\simeq t'$. Similarly $u \not\simeq s \equiv_{D \vee a \not\simeq b} u' \not\simeq s'$ and $v \not\simeq t \equiv_{D \vee a \not\simeq b} v' \not\simeq t'$, because $D'^-_1 \leq_{\text{I}} D'^-$. These arguments are enough to conclude that $D'[a/b]^- \equiv C'^-$, hence $D[a/b]^- \equiv C^-$. ∎

Corollary 2.45 *If in addition D_1 is $\langle a, b\rangle$-neutral, then $D[a/b]_{\downarrow} \leq_{\text{I}} C$.*

Proof. By Proposition 2.44, we already know that $D[a/b]^- \equiv C^-$, thus we only need to consider the positive literals:

— By definition, $u \simeq v \equiv_{C \vee a \not\simeq b} u' \simeq v'$, thus $(u \simeq v)[a/b] \leq_{\text{I}} C$.

— Let l be a positive literal in $D'_1[a/b]$. By definition of the normal form of clauses, there exists a positive literal $l' \in C_1$ such that $l \equiv_{C_1} l'$. If $l \equiv_{C_1} u \simeq v$ or $l \equiv_{C_1} s \simeq t$, then $|{C_1}^+| < |D_1[a/b]^+|$, because equivalent literals are factorized in normal form. In this case D_1 is not $\langle a, b\rangle$-neutral, which raises a contradiction. The only remaining possibility is that $l' \in C'_1$, thus $l \leq_{\text{I}} C$.

We now know that $D[a/b] \models C$ and if $D[a/b] \not\leq_{\text{I}} C$ then any two positive literals l_1, l_2 in D such that $l_1 \equiv_{C \vee a \not\simeq b} l_2$ verify also $l_1 \equiv_{D \vee a \not\simeq b} l_2$, because $D[a/b]^- \equiv C^-$. Hence l_1 and l_2 are projected on a single literal in $D[a/b]_{\downarrow}$, ensuring that $D[a/b]_{\downarrow} \leq_{\text{I}} C$. ∎

Proposition 2.46 *Let C be a clause generated from C_1 and C_2 using the rule $\mathtt{P}^+$, then there exists a clause D generated by $\mathtt{P}^+$ from D_1 and D_2 such that $D[a/b]^- \equiv C^-$.*

Proof. Since C is generated using $\mathtt{P}^+$, we can assume that the clauses are decomposed in the following way: $C_1 = u' \simeq v' \vee C'_1$ and $C_2 = s' \simeq t' \vee C'_2$ generating $C = (u' \not\simeq s' \vee v' \simeq t' \vee C'_1 \vee C'_2)_{\downarrow}$ and by Proposition 2.43:

— $D_1 = u \simeq v \vee D'_1$ where u and v verify $u \equiv_{C'_1 \vee a \not\simeq b} u' \equiv_{D'_1 \vee a \not\simeq b} u$ and $v \equiv_{C'_1 \vee a \not\simeq b} v' \equiv_{D'_1 \vee a \not\simeq b} v$;

— and $D_2 = s \simeq t \vee D'_2$ where s and t verify $s \equiv_{C'_2 \vee a \not\simeq b} s' \equiv_{D'_2 \vee a \not\simeq b} s$ and $t \equiv_{C'_2 \vee a \not\simeq b} t' \equiv_{D'_2 \vee a \not\simeq b} t$.

By applying $\mathtt{P}^+$ on D_1 and D_2, we generate $D = (u \not\simeq s \vee v \simeq t \vee D'_1 \vee D'_2)_{\downarrow}$. We write $D' = u \not\simeq s \vee v \simeq t \vee D'_1 \vee D'_2$ and $C' = u' \not\simeq s' \vee v' \simeq t' \vee C'_1 \vee C'_2$. Firstly, we observe that $u \not\simeq s \equiv_{C' \vee a \not\simeq b} u' \not\simeq s' \equiv_{D' \vee a \not\simeq b} u \not\simeq s$, because $C'_1 \vee C'_2 \leq_{\text{I}} C'$ and $D'_1 \vee D'_2 \leq_{\text{I}} D'$. In addition, we have immediately $(D'_1 \vee D'_2)[a/b] \equiv C'_1 \vee C'_2$, thus the conclusion. ∎

Corollary 2.47 *If in addition D_1 and D_2 are $\langle a, b\rangle$-neutral, then $D[a/b]_{\downarrow} \leq_{\text{I}} C$.*

PROOF. First, we prove that $D'[a/b] \models C'$ (with C' and D' defined as in the proof of Proposition 2.46 just above). We know that $(v \simeq t)[a/b] \leq_{\text{I}} C'$, because $v \simeq t \equiv_{C' \vee a \not\simeq b} v' \simeq t'$. Moreover, for any positive literal l in $D'_1 \vee D'_2$, there exists $l' \in C'_1 \vee C'_2$ such that $l \equiv_{C'_1 \vee C'_2 \vee a \not\simeq b} l'$, because D_1 and D_2 are $\langle a, b \rangle$-neutral (as seen in the proof of Corollary 2.45). Then we know that $D[a/b] \models C$ and since, by Proposition 2.46, $D[a/b]^- \equiv C^-$, we reach the desired conclusion by the same reasoning as in the proof of Corollary 2.45. ∎

Proposition 2.48 *Let C be a clause generated by* M *on C_1 and C_2. Then $C_1 = u' \simeq v' \vee C'_1$ with $u' \simeq v'$ the positive literal involved in* M*. Let c and d be constants.*

$$\text{If } \begin{cases} c \equiv_{C_2 \vee a \not\simeq b} d \\ c \not\equiv_{C \vee a \not\simeq b} d \end{cases} \text{, then either } \begin{cases} c \equiv_{C \vee a \not\simeq b} u' \\ d \equiv_{C \vee a \not\simeq b} v' \end{cases} \text{ or } \begin{cases} c \equiv_{C \vee a \not\simeq b} v' \\ d \equiv_{C \vee a \not\simeq b} u' \end{cases} .$$

PROOF. Assume that $c \equiv_{C_2 \vee a \not\simeq b} d$ and first suppose that $b \notin \{c, d\}$. Then since b does not occur in C_2, necessarily $c \equiv_{C_2} d$. Similarly, since $c \not\equiv_{C \vee a \not\simeq b} d$ and b does not occur in C, we have $c \not\equiv_C d$. This is only possible[3] if $c \equiv_C u'$ and $d \equiv_C v'$, or $c \equiv_C v'$ and $d \equiv_C u'$, hence the result by Proposition 1.7.

Now suppose that $b = c$, the case where $b = d$ is symmetrical. Then from $b \equiv_{C_2 \vee a \not\simeq b} d$, we deduce that $a \equiv_{C_2 \vee a \not\simeq b} d$, and finally that $a \equiv_{C_2} d$. Similarly, from $a \not\equiv_{C \vee a \not\simeq b} d$ we deduce that $b \not\equiv_C d$; hence, w.l.o.g., $b \equiv_C u'$ and $d \equiv_C v'$. Therefore, $c \equiv_{C \vee a \not\simeq b} u'$ and $d \equiv_{C \vee a \not\simeq b} v'$. ∎

Proposition 2.49 *Let C be a clause generated from C_1 and C_2 using the rule* M*, then there exists a clause D generated by* M *from D_1 and D_2 such that $D[a/b]^- \equiv C^-$.*

PROOF. Since C is generated by M from C_1 and C_2, a natural decomposition of these clauses is $C_1 = u' \simeq v' \vee C'_1$ and $C_2 = \bigvee_{i=1}^m s'_i \not\simeq t'_i \vee C''_2$ ($m \geq 1$), with $C = (\bigvee_{i=1}^m (s'_i \not\simeq u' \vee v' \not\simeq t'_i) \vee C'_1 \vee C''_2)_{\downarrow}$. A different (but equivalent) decomposition of C_2 is more interesting because it enables us to exhibit the desired clause D with a minimum of effort. This decomposition is the following: $C_2 = R' \vee L' \vee I \vee C'_2 \vee g' \vee h'$, with the clauses:

$$R' = \bigvee_{i=1}^j s'_i \not\simeq r' \ , \quad L' = \bigvee_{i=j+1}^k r' \not\simeq t'_i \ , \quad I = \bigvee_{i=k+1}^n s'_i \not\simeq t'_i \ ,$$

$$g' = \begin{cases} \square & \text{if } \overline{r}' = \lambda \text{ or } r'_b = r_a \\ r' \not\simeq \overline{r}' & \text{otherwise} \end{cases} \ , \qquad h' = \begin{cases} \square & \text{if } a = r_a \\ a \not\simeq r' & \text{otherwise} \end{cases} .$$

Note that here and in all this proof, the constants r_a, r_b, r, $\overline{r}$, r' and $\overline{r}'$ are defined based solely on D_2 and r' is always the C_2-representative of a. The link between this decomposition and the previous one is that $L' \cup R' \cup I \subseteq \bigvee_{i=1}^m s'_i \not\simeq t'_i$ and $C'_2 \subseteq C''_2$ with g' and h' belonging to either one depending on

3. Indeed, by definition of M, the only negative literals occurring in C_2 but not in C are of the form $e \not\simeq f$, with $e \equiv_C u'$, and $f \equiv_C v'$

C. Indeed, since C_2 is obtained from D_2 by replacing b by a and normalizing, $r_{\downarrow C_2}$ must be equal to $\min([a]_{D_2} \cup [b]_{D_2}) = \min(r_a, r'_b)$. If $r_{\downarrow C_2} = r_a$, then $r_a \preceq r_b$, and since we have $r_b \preceq r'_b$ we deduce that $r' = r_a$. Otherwise, we have $r_b \prec r_a$ then (since $b \succ a$), $[b]_{D_2} \neq \{b\}$ thus $r'_b = r_b$ and $r' = r_b$. The sets R' and L' represent the literals $s'_i \not\simeq t'_i$ in C_2 containing r' with the exception of $a \not\simeq r'$ and $\overline{r}' \not\simeq r'$ which if they exist are treated separately (in h' and g' respectively). More precisely L' contains those where $r'(= a_{\downarrow C_2})$ is associated to u' and R' those where r' is associated to v'. Then the clause C is of the form $C = (R'_{u',v'} \vee L'_{u',v'} \vee I_{u',v'} \vee C'_1 \vee C'_2 \vee G' \vee H')_{\downarrow}$, with $G' \in \{g', g'_{u',v'}, g'_{v',u'}\}$ and $H' \in \{h', h'_{u',v'}, h'_{v',u'}\}$. According to Proposition 2.43, we write $D_1 = u \simeq v \vee D'_1$ where $u \equiv_{C'_1 \vee a \not\simeq b} u' \equiv_{D_1 \vee a \not\simeq b} u$ and $v \equiv_{C'_1 \vee a \not\simeq b} v' \equiv_{D_1 \vee a \not\simeq b} v$, and $D_2 = R \vee \overline{R} \vee L \vee \overline{L} \vee I \vee N \vee \overline{N} \vee D'_2 \vee g \vee h$ where:

$$R = \bigvee_{i=1}^{j_1} s'_i \not\simeq r, \qquad \overline{R} = \bigvee_{i=j_1+1}^{j} s'_i \not\simeq \overline{r}, \qquad L = \bigvee_{i=j+1}^{k_1} r \not\simeq t'_i,$$

$$\overline{L} = \bigvee_{i=k_1+1}^{k} \overline{r} \not\simeq t'_i, \qquad N = \bigvee_{i=n+1}^{m_1} d_i \not\simeq r, \qquad \overline{N} = \bigvee_{i=m_1+1}^{m_2} d_i \not\simeq \overline{r},$$

$$g = \begin{cases} \Box & \text{if } r'_b = \lambda \\ b \not\simeq r'_b & \text{otherwise} \end{cases} \qquad h = \begin{cases} \Box & \text{if } a = r_a \\ a \not\simeq r_a & \text{otherwise} \end{cases}$$

For each literal $s'_i \not\simeq r'$ from R' (resp. $r' \not\simeq t'_i$ from L') there exists a corresponding literal $s_i \not\simeq r_a$ or $s_i \not\simeq r_b$ (resp. $r_a \not\simeq t_i$ or $r_b \not\simeq t_i$) in D_2. In R (resp. in L) are such literals where $r(= \min(r_a, r_b))$ appears and the others are in $\overline{R}$ (resp. $\overline{L}$). The same principle applies to the literals of N and $\overline{N}$, which correspond to the literals with constants in $[a]_{C_2}$ appearing in C'_2 (i.e. those not involved in the application of `M`) [4].

Let $C_0 = C \vee a \not\simeq b$. We distinguish several cases, depending on the relationships between a, r', and $\overline{r}'$. In each case, we define the clause D by specifying the literals of D_2 upon which the rule `M` is applied. Each such literal or set of literals l is replaced by $l_{u,v}$ in the conclusion of the rule.

1. Assume that $a \equiv_{C_0} r' \equiv_{C_0} \overline{r}'$. Then we define D as $(R_{u,v} \vee \overline{R}_{u,v} \vee L_{u,v} \vee \overline{L}_{u,v} \vee I_{u,v} \vee N \vee \overline{N} \vee D'_1 \vee D'_2 \vee G)_{\downarrow}$ where $G = \begin{cases} g \vee h & \text{if } u' \not\equiv_{C_0} v' \\ g_{u,v} \vee h_{v,u} & \text{otherwise} \end{cases}$. The value of G is explained as follows. If $u' \not\equiv_{C_0} v'$ then $G' = g'$ and $H' = h'$ (in C), otherwise the assumption $a \equiv_{C_0} r' \equiv_{C_0} \overline{r}'$ is contradicted, hence $G = g \vee h$ must appear in D. If $u' \equiv_{C_0} v'$ then we need $\{u, v\} \equiv_{C_2 \vee b \not\simeq a} \{a, b\}$ to ensure $u' \equiv_{D[a/b]} v'$. This is guaranteed by $G = g_{u,v} \vee h_{v,u}$, no matter the values of g and h. By definition, D is generated by `M` from D_1 and D_2. By looking at the decomposition given for C and D (before normalization) it is straightforward to verify that every negative literal in D entails $C^- \vee a \not\simeq b$, thus $D[a/b]^- \models C^-$. Let

4. Note that since these literals are not involved in `M`, we do not need another pair of subclauses for literals of the form $r \not\simeq d_i$ and $\overline{r} \not\simeq d_i$. We can assume w.l.o.g. that there are none.

$D_0 = D \vee a \not\simeq b$. By construction, $a \equiv_{D_0} b \equiv_{D_0} r'_b \equiv_{D_0} r_a$, and $u \not\equiv_{D_0} v$ exactly when $u' \not\equiv_{C_0} v'$, hence it can be verified that every negative literal in C entails D_0. We conclude that $D[a/b]^- \equiv C^-$.

2. Assume that $a \equiv_{C_0} r' \not\equiv_{C_0} \overline{r}'$. Then either $\overline{r}' = \lambda$ or, by Proposition 2.48, since $r' \equiv_{C_2 \vee b \not\simeq a} \overline{r}'$, necessarily $u' \equiv_{C_0} r'$ and $v' \equiv_{C_0} \overline{r}'$ (in this case and the followings, we discard the symmetrical cases where u' and v' are swapped). We can first note that $R = \overline{R} = \Box$ (because $R' = \Box$ otherwise $v' \not\simeq r' \models C$, thus $v' \equiv_{C_0} r'$, hence $u' \equiv_{C_0} v'$, a contradiction) [5]. Two cases must be distinguished:
   — if $b \neq r_b$, then $r = r'$ and $r'_b = r_b$ thus $\overline{r} = \overline{r}'$. We define D as $(L_{u,v} \vee \overline{L} \vee I_{u,v} \vee N \vee \overline{N}_{u,v} \vee D'_1 \vee D'_2 \vee G)_{\downarrow}$ where $G = \begin{cases} g \vee h_{u,v} & \text{if } r = r_b \\ g_{u,v} \vee h & \text{otherwise} \end{cases}$;
   — if $b = r_b$, then since $a \prec b$, $r = r' = r_a$ and $\overline{r}' = r'_b$, in which case we define D as $(L_{u,v} \vee \overline{L}_{u,v} \vee I_{u,v} \vee N \vee \overline{N} \vee D'_1 \vee D'_2 \vee g_{u,v} \vee h)_{\downarrow}$.
   In both cases, the clause D is generated by $\mathtt{M}$ from D_1 and D_2. By construction, $a \equiv_{D_0} b \equiv_{D_0} r \equiv_{D_0} u \not\equiv_{D_0} v \equiv_{D_0} \overline{r}$ and $D[a/b]^- \equiv C^-$.
3. Assume that $u' \equiv_{C_0} r' \not\equiv_{C_0} \overline{r}' \equiv_{C_0} a \equiv_{C_0} v'$. Then $R = \overline{R} = \Box$ as in case 2, and $D = (L_{u,v} \vee \overline{L} \vee I_{u,v} \vee N \vee \overline{N}_{u,v} \vee D'_1 \vee D'_2 \vee G)_{\downarrow}$ where:
   — if $b \neq r_b$, then $G = g_{v,u} \vee \begin{cases} h & \text{if } \overline{r}' = r_a \\ h_{v,u} & \text{otherwise} \end{cases}$;
   — if $b = r_b$, then $G = g \vee h_{v,u}$.
   Then we have $D[b/a]^- \equiv C^-$.
4. Assume that $u' \equiv_{C_0} r' \equiv_{C_0} \overline{r}' \not\equiv_{C_0} b \equiv_{C_0} v'$. Again we note that $R = \overline{R} = \Box$ and:
   — if $b \neq r_b$ then $D = (L_{u,v} \vee \overline{L}_{u,v} \vee I_{u,v} \vee N \vee \overline{N} \vee D'_1 \vee D'_2 \vee g_{v,u} \vee h_{v,u})_{\downarrow}$;
   — otherwise $D = (L_{u,v} \vee \overline{L} \vee I_{u,v} \vee N \vee \overline{N}_{u,v} \vee D'_1 \vee D'_2 \vee g_{v,u} \vee h_{v,u})_{\downarrow}$.
   Once again it is straightforward to verify that $D[a/b] \equiv C$.

Since by Proposition 2.48, the case where $r' \not\equiv_{C_0} \overline{r}'$, $r' \not\equiv_{C_0} a$ and $a \not\equiv_{C_0} \overline{r}'$ cannot happen, all the possible cases have been examined. ∎

Corollary 2.50 *If in addition D_1 and D_2 are $\langle a, b \rangle$-neutral then $(D[a/b])_{\downarrow} \leq_{\mathrm{I}} C$.*

Proof. We denote by C' the result of applying $\mathtt{M}$ between C_1 and C_2 without normalization, and by D' the same with D_1 and D_2. We prove first that $D'[a/b] \models C'$. By Proposition 2.49, $D'[a/b]^- \equiv C'^-$, thus we only need to consider the positive literals in D'. If D_1 and D_2 are $\langle a, b \rangle$-neutral then for every literal l in $D'^+ = (D'_1 \vee D'_2)^+$, we have $l[a/b] \in (C'_1 \vee C'_2)^+ = C'^+$. Hence $D'[a/b] \models C'$ and $D[a/b] \models C$. Then, as seen before, $D[a/b]^- \equiv C^-$ guarantees that $D[a/b]_{\downarrow} \leq_{\mathrm{I}} C$. ∎

Before proving the rewrite-stability of the rules $\mathtt{Rw_P}$ and $\mathtt{Rw_W}$, we still need more results. We define a notion of measure and present some results associated

5. And in the discarded symmetrical cases, we always have $L = \overline{L} = \Box$.

to this notion. We consider a clause D such that $b \simeq a \not\models D$. This condition, appearing in all the following propositions, ensures that $D \vee a \not\simeq b$ is not a tautology, which is (among other things) necessary to use projections and normal form on this clause.

Definition 2.51 Let D be a clause such that $a \simeq b \not\models D$. For any clause X such that $X \leq_{\mathrm{I}} D \vee a \not\simeq b$, the *measure* $\Gamma_{[a/b],D}(X)$ is the multiset containing the projections on $D \vee a \not\simeq b$ of the positive literals of X. More formally, $\Gamma_{[a/b],D}(X) = {X^+}_{\downarrow D \vee a \not\simeq b}$. An equivalent definition is $\Gamma_{[a/b],D}(X) = (X[a/b]^+)_{\downarrow D[a/b]}$, the multiset containing the projections on $D[a/b]$ of the positive literals of $X[a/b]$. This measure is associated to the multiset order based on the one used for literals in the usual way. ◊

Given this definition, the following facts can be inferred.

Proposition 2.52 *Let X, X_1 and X_2 be clauses such that $X_k \leq_{\mathrm{I}} D \vee a \not\simeq b$ for $k \in \{\epsilon, 1, 2\}$.*

1. *$\forall l \in \Gamma_{[a/b],D}(X)$, $l \in D[a/b]_{\downarrow}$.*
2. *If every literal of $\Gamma_{[a/b],D}(X)$ occurs at most once then $X[a/b] \leq_{\mathrm{I}} D[a/b]_{\downarrow}$.*
3. *If $X_1 \leq_{\mathrm{I}} X_2$ then $\Gamma_{[a/b],D}(X_1) \leq \Gamma_{[a/b],D}(X_2)$.*

Proof. Let us be in the conditions described in Proposition 2.52. Each of the following point is the demonstration of the corresponding property.

1. Direct consequence of the definition.
2. Let us assume that the literals of $\Gamma_{[a/b],D}(X)$ are pairwise distinct. Since we assume that $\Gamma_{[a/b],D}(X)$ is defined, we necessarily have $X \leq_{\mathrm{I}} D \vee a \not\simeq b$. Thus by Proposition ii.3, $X[a/b] \models D[a/b]$. If $X[a/b] \not\leq_{\mathrm{I}} D[a/b]_{\downarrow}$ then there are two distinct literals l_1 and l_2 in $X[a/b]$ and a literal $l \in D[a/b]_{\downarrow}$ such that $l_1 \equiv_{D[a/b]} l_2 \equiv_{D[a/b]} l$. Hence $\{l, l\} \subseteq \Gamma_{[a/b],D}(X)$ and a contradiction is reached, implying that $X[a/b] \leq_{\mathrm{I}} D[a/b]_{\downarrow}$
3. By definition, $\Gamma_{[a/b],D}(X_i)$, with $i \in \{1, 2\}$, is the multiset of literals $l_{\downarrow D \vee a \not\simeq b}$ where $l \in {X_i}^+$. There exists an injective function $\gamma : {X_1}^+ \to {X_2}^+$ such that for every $l \in {X_1}^+$, $\gamma(l) = l_{\downarrow X_2}$, hence, since $X_2 \leq_{\mathrm{I}} D \vee a \not\simeq b$, $l \equiv_{D \vee a \not\simeq b} \gamma(l)$. Thus $\Gamma_{[a/b],D}(X_1) \leq \Gamma_{[a/b],D}(X_2)$. ■

This notion of measure is used in the proof of the following proposition.

Proposition 2.53 *Let P and W be sets of clauses such that $P \cup W_{\mathrm{F}} \leq_{\mathrm{I}} P_{\vdash 1}$. Let D be a clause such that $a \simeq b \not\models D$. Let E be a clause in $P \cup W_{\mathrm{F}}$ such that $E \leq_{\mathrm{I}} D \vee a \not\simeq b$. Then there exists a clause $G \in P \cup W_{\mathrm{F}}$ such that $G[a/b]_{\downarrow} \leq_{\mathrm{I}} D[a/b]_{\downarrow}$.*

Proof. We reason by induction on $\Gamma_{[a/b],D}(E)$. If the literals of $\Gamma_{[a/b],D}(E)$ are pairwise distinct then, by Proposition 2.52(2), $E[a/b] \leq_{\mathrm{I}} D[a/b]_{\downarrow}$, thus $E[a/b]_{\downarrow} \leq_{\mathrm{I}} D[a/b]_{\downarrow}$. This is our base case. In the remaining cases $\Gamma_{[a/b],D}(E) \not\subseteq$

$D[a/b]_{\downarrow}$ (and $E[a/b]_{\downarrow} \not\leq_{\text{I}} D[a/b]_{\downarrow}$, or the result is trivial). Since, by Proposition ii.3, $E[a/b] \models D[a/b]$, there exist at least two literals l_1 and l_2 in E such that $l_1[a/b] \equiv_{D[a/b]} l_2[a/b]$. We use the rule $\texttt{F}$ on E to factorize these two literals, generating a new clause called E', so that $E' \leq_{\text{I}} D \vee a \not\approx b$ and $\Gamma_{[a/b],D}(E') = \Gamma_{[a/b],D}(E)/\{l\}$ with $l = l_1[a/b]_{\lfloor D[a/b]}$. Moreover, we have $P \cup W_{\texttt{F}} \leq_{\text{I}} E'$ because either $E \in W_{\texttt{F}}$, entailing $W_{\texttt{F}} \leq_{\text{I}} E'$, or $E \in P$, and given the hypothesis $P \cup W_{\texttt{F}} \leq_{\text{I}} P_{\vdash 1}$, the result is verified. Therefore there exists $F \in P \cup W_{\texttt{F}}$ such that $F \leq_{\text{I}} E'$. By transitivity, F also verifies $F \leq_{\text{I}} D \vee a \not\approx b$, and since $F \leq_{\text{I}} E'$, by Proposition 2.52(3), $\Gamma_{[a/b],D}(F) \leq \Gamma_{[a/b],D}(E')$. Moreover, we build E' so that $\Gamma_{[a/b],D}(E') \subset \Gamma_{[a/b],D}(E)$, thus $\Gamma_{[a/b],D}(E') < \Gamma_{[a/b],D}(E)$. We were able to find a clause F such that $F \in P \cup W_{\texttt{F}}$, $F \leq_{\text{I}} D \vee a \not\approx b$ and $\Gamma_{[a/b],D}(F) < \Gamma_{[a/b],D}(E)$. Therefore, by the induction hypothesis, we conclude that there exists a clause $G \in P \cup W_{\texttt{F}}$ such that $G[a/b]_{\downarrow} \leq_{\text{I}} D[a/b]_{\downarrow}$. ∎

We have now all the tools necessary to prove the rewrite-stability of the two atomic rewriting rules.

Theorem 2.54 *The rules* $\texttt{Rw}_{\texttt{P}}$ *and* $\texttt{Rw}_{\texttt{W}}$ *are rewrite-stable.*

Proof. Let $\langle P'; W'; A' \rangle$ be generated from $\langle P; W; A \rangle$ by applying either $\texttt{Rw}_{\texttt{P}}$ or $\texttt{Rw}_{\texttt{W}}$. We show that $(P \cup W_{\texttt{F}}) \leq_{\text{I}} P_{\vdash 1} \Rightarrow (P' \cup {W'}_{\texttt{F}}) \leq_{\text{I}} {P'}_{\vdash 1}$. To prove it, the main idea is to exhibit, for each considered case, the clause that fits our needs (i.e. the clause in $P' \cup {W'}_{\texttt{F}}$ that I-subsumes the considered clause of ${P'}_{\vdash 1}$). To do so, we reason by induction on the number of inferences that occur during the execution of the procedure. Our induction hypothesis is: ${P^{(i)} \cup W^{(i)}}_{\texttt{F}} \leq_{\text{I}} {P^{(i)}}_{\vdash 1}$, where $P^{(i)}$ and $W^{(i)}$ represent the set P and W respectively, after the i^{th} inference of the procedure. The base case is true because the procedure starts with input $\langle \emptyset; S; \emptyset \rangle$, thus $({P^{(0)} \cup W^{(0)}}_{\texttt{F}}) \leq_{\text{I}} {P^{(0)}}_{\vdash 1}$. For the recursive case, considering that $\texttt{I}$ and $\texttt{R}$ are trivially rewrite-stable, only the rules $\texttt{Rw}_{\texttt{P}}$ and $\texttt{Rw}_{\texttt{W}}$ are of interest to us. In both cases, given a clause $C \in {P^{(i)}}_{\vdash 1}$ there are three possibilities to consider:

1. C has been generated using $\texttt{F}$
2. C has been generated using $\texttt{P}^{+}$
3. C has been generated using $\texttt{M}$

By applying respectively the corollaries 2.45, 2.47, 2.50 to the given situations, we generate a clause $D \in {P^{(i-1)}}_{\vdash 1}$ such that $D[a/b]_{\downarrow} \leq_{\text{I}} C$. By the induction hypothesis, there exists a clause $E \in {P^{(i-1)} \cup W^{(i-1)}}_{\texttt{F}}$ such that $E \leq_{\text{I}} D$, hence $E \leq_{\text{I}} D \vee a \not\approx b$ and by Proposition 2.53 there exists a clause $G \in {P^{(i-1)} \cup W^{(i-1)}}_{\texttt{F}}$ such that $G[a/b]_{\downarrow} \leq_{\text{I}} D[a/b]_{\downarrow}$. Therefore, this clause $G \in {P^{(i-1)} \cup W^{(i-1)}}_{\texttt{F}}$ verifies also $G[a/b]_{\downarrow} \leq_{\text{I}} C$ and we can conclude that the rules $\texttt{Rw}_{\texttt{P}}$ and $\texttt{Rw}_{\texttt{W}}$ are rewrite-stable. ∎

4 Summary

This chapter presented the $\mathcal{K}$-paramodulation calculus, a form of "conditional paramodulation" where equality conditions are not checked statically but

asserted by adding new disequations to the derived clause. For instance, given a clause $C[a]$ and an equation $a' \simeq b$, $\mathcal{K}$-paramodulation generates the clause $a \not\simeq a' \vee C[b]$, which can be interpreted as $a \simeq a' \Rightarrow C[b]$ (the condition $a \simeq a'$ is asserted). We proved the soundness and deductive-completeness of this calculus and then proceeded to expose an improvement on this calculus impacting the number of inferences necessary to reach saturation. This improvement reduces the number of constants occuring in the problem, replacing all those that appear as the maximal term of an atomic implicate by the minimal one. A method to efficiently recover the prime implicates of the original input from the atomic implicates and the saturation of the simplified problem is also devised. This improvement is realized in two different ways, first as a preprocessing method relying on a different variant of the paramodulation calculus to generate all atomic prime implicates for the simplification of the input formula and then as a mechanism integrated to the $\mathcal{K}$-paramodulation saturation process. In both cases, the soundness and deductive-completeness of the methods were proved.

Chapter 3

Constrained Superposition Calculus

In this chapter we propose a different approach to the problem of prime implicate generation. The principle of the presented calculus is to apply the standard superposition calculus, enriched by new rules that allow the addition of ground unit clauses as new axioms (or hypotheses) during the search. As with $\mathcal{K}$-paramodulation, we still allow the assertion of equations, but this time a distinction is drawn between the literals that are asserted and the standard ones – the former being attached to the clauses as constraints. These constraints are taken into account when testing redundancy [1]. Once an empty clause has been generated, the negation of the conjunction of the hypotheses used to derive it can be returned as an implicate of the considered clause set. It is clear that from a theoretical point of view this approach can strongly increase the search space, since any clause of size n can now have 2^n distinct equivalent representatives, depending on which literals are stored in the constraints. However, it also has many advantages:

— First, all the usual ordering restrictions or selection strategies of the superposition calculus can be carried over to the new procedure. This was not the case for our previous calculus: their addition renders the $\mathcal{K}$-paramodulation calculus incomplete for consequence-finding.

— Second, this approach offers the possibility to control the literals that can be asserted during the search, for instance to limit the number of asserted literals, to impose additional syntactic restrictions on them, or even to test semantic conditions. This is especially important in practice since the number of implicates of a formula is typically huge, as soon as equality axioms are considered.

This method also has the advantage that it can easily be extended from $\mathbb{E}_0$ to $\mathbb{E}_1$. Such an extension is the subject of Section 3

1. For instance a clause $p \vee q$ with no assumed literals is not necessarily less general than a unit clause p with constraint r.

1 Constrained clauses

Constraints and constrained clauses are notions that are used here and appear also in Part II.

Definition 3.1 A *constraint* is a (possibly empty) conjunction of literals. A *constrained clause* (or *c-clause*) is a pair $[C \,|\, \mathcal{X}]$ where C is a clause and $\mathcal{X}$ is a constraint. The empty constraint is denoted by $\top$. $[C \,|\, \top]$ is often written simply as C and referred to as a standard clause. ◊

If $\mathcal{X} = \bigwedge_{i=1}^{n} l_i$ then $\mathcal{X}^c$ denotes the clause $\bigvee_{i=1}^{n} l_i^c$. The *normal form* of a constraint $\mathcal{X}$ is $\neg({\mathcal{X}^c}_{\downarrow})$. Similarly to clauses, we use the notation $\mathcal{X}_{\downarrow}$ to denote a constraint in normal form.

Proposition 3.2 *There exists a unique normalized constraint equivalent to each non-contradictory constraint.*

We also adapt the notion of projection to constraints.

Definition 3.3 Let C be a standard clause and $\mathcal{X}$ be a constraint, then $C_{\downarrow\mathcal{X}}$ denotes $C_{\downarrow\mathcal{X}^c}$ ◊

From a semantic point of view, a constrained clause $[C \mid \mathcal{X}]$ is equivalent to $\mathcal{X}^c \vee C$. For example the c-clause $[c \simeq b \mid a \simeq c \wedge c \not\simeq d]$ is equivalent to $c \simeq b \vee a \not\simeq c \vee c \simeq d$. More specifically, the intended meaning of a c-clause $[C \,|\, \mathcal{X}]$ is that the clause C can be inferred provided the literals in $\mathcal{X}$ are added as axioms to the considered clause set.

The definitions and results related to constrained clauses can be extended to $\mathbb{E}_1$ from $\mathbb{E}_0$ without modifications (but using the extended notion of normalization defined Chapter 1 Section 1.2).

2 Main calculus

In this section the superposition calculus is extended to sets of c-clauses (see Definition 3.1). First, the standard inference rules are extended in a straightforward way by adding the constraints of the premises to the conclusion (see Definition 3.4). As usual the calculus is parameterized by the ordering $\succ$ on terms and a selection function *sel*, where $sel(C)$ contains all maximal literals in C or (at least) one negative literal. A literal is *selected* in C if it occurs in $sel(C)$.

Definition 3.4 The following rules are the standard inference rules of the superposition calculus (see Definition ii.17), adapted to c-clauses.

$$\text{Superposition:} \quad \frac{[l \simeq r \vee C \,|\, \mathcal{X}] \quad [l \bowtie u \vee D \,|\, \mathcal{Y}]}{[r \bowtie u \vee C \vee D \,|\, \mathcal{X} \wedge \mathcal{Y}]}\ (1.),$$

$$\text{Factoring:} \quad \frac{[t \simeq u \vee t \simeq v \vee C \,|\, \mathcal{X}]}{[t \simeq v \vee u \not\simeq v \vee C \,|\, \mathcal{X}]}\ (2.),$$

with the following conditions:

1. $l \succ r$, $l \succ u$, and $(l \simeq r)$ and $(l \bowtie u)$ are selected in $(l \simeq r \vee C)$ and $(l \bowtie u \vee D)$ respectively,
2. $t \succ u$, $t \succ v$ and $(t \simeq u)$ is selected in $t \simeq u \vee t \simeq v \vee C$.

The clausal part of the consequents are systematically normalized, but the constraint part is not. ◊

The Assertion rules (see Definition 3.5) allow for the addition of new hypotheses in the constraint part of a c-clause. To this purpose the most simple solution would be to add to the clause set all tautological axioms of the form $[l \mid l]$ (meaning that l can be derived from l) where l is a ground literal, and then to let the inference rules of Definition 3.4 derive all the consequences of these axioms. However, this solution is not completely satisfactory since there are numerous axioms of the previous form, and that not all of them are relevant w.r.t. the considered clause set. It is preferable to avoid the blind enumeration of axioms, which is why we add rules simulating all possible inferences from these axioms and the already generated c-clauses [2].

Definition 3.5 The following rules are the Assertion rules of the constrained calculus.

$$\text{Positive Assertion:} \quad \frac{[t \bowtie s \vee C \,|\, \mathcal{X}]}{[u \bowtie s \vee C \,|\, \mathcal{X} \wedge t \simeq u]}\ (1.),$$

$$\text{Negative Assertion:} \quad \frac{[t \simeq s \vee C \,|\, \mathcal{X}]}{[s \not\simeq u \vee C \,|\, \mathcal{X} \wedge t \not\simeq u]}\ (2.),$$

with the following conditions:

1. $t \succ s$, $t \succ u$ and $t \bowtie s$ is selected in $t \bowtie s \vee C$,
2. $t \succ u$, $t \succ s$ and $t \simeq s$ is selected in $t \simeq s \vee C$.

In these rules, the clausal part of the generated clauses is also normalized. ◊

The Positive Assertion rule asserts an equation $t \simeq u$ as a new hypothesis in the constraint part of a c-clause. This is done if the addition of such a hypothesis enables the application of the superposition rule into the considered clause. Note that the term u does not necessarily occur in C: the condition is only that it must be strictly smaller than t. The negative assertion rule proceeds in a similar way for disequations, which allow for an application of the superposition rule into them. A literal $t \not\simeq u$ is added to the constraint part of a c-clause of the form $[t \simeq s \vee C \,|\, \mathcal{X}]$ that is replaced by $[s \not\simeq u \vee C \,|\, \mathcal{X} \wedge t \not\simeq u]$, only if this correspond to a Superposition inference into this new constraint (again, the term u does not necessarily occur in the premise).

Definition 3.6 The constrained superposition calculus in $\mathbb{E}_0$, denoted $c\mathcal{SP}_0$, is composed of the rules of Definition 3.4 and 3.5. ◊

2. From a purely theoretical point of view the two solutions are of course equivalent.

Example 3.7 The following example shows how to derive the implicate $a \not\simeq c \vee b \simeq d$ from $\{a \simeq b, c \simeq d\}$.

1	$[a \simeq b \mid \top]$	(hyp)
2	$[c \simeq b \mid a \simeq c]$	(Pos. AR, 1)
3	$[c \not\simeq d \mid a \simeq c \wedge b \not\simeq d]$	(Neg. AR, 2)
4	$[c \simeq d \mid \top]$	(hyp)
5	$[d \not\simeq d \mid a \simeq c \wedge b \not\simeq d]$	(Sup. 3, 4)
6	$[\Box \mid a \simeq c \wedge b \not\simeq d]$	(Ref. 5)

The negation of $a \simeq c \wedge b \not\simeq d$ is the desired implicate. ♣

The usefulness of the c-clause representation becomes apparent when looking at the inference rules (both standard and Assertion rules). It is a way to separate the literals that can be used for inferences from the "frozen" ones stored in the constraints, on which no inference should be applied.

Example 3.8 Let us consider the fifth step of the derivation in Example 3.7, i.e. the superposition between $[c \not\simeq d \mid a \simeq c \wedge b \not\simeq d]$ and $[c \simeq d \mid \top]$. Due to the freezing of $a \simeq c$ and $b \not\simeq d$, there are no standard inferences besides the previous one between these two clauses, and only two assertion inferences that generate the c-clauses $[\Box \mid a \simeq c \wedge b \not\simeq d \wedge c \simeq d]$ and $[\Box \mid c \not\simeq d]$. In contrast, using $\mathcal{K}$-paramodulation on the equivalent clauses $c \not\simeq d \vee a \not\simeq c \vee b \simeq d$ and $c \simeq d$, all the following clauses are generated:

$$
\begin{array}{lcl}
(c \not\simeq c \vee d \not\simeq d \vee a \not\simeq c \vee b \simeq d)_{\downarrow} & = & a \not\simeq c \vee b \simeq d \\
(c \not\simeq d \vee c \not\simeq d \vee a \not\simeq c \vee b \simeq d)_{\downarrow} & = & c \not\simeq d \vee a \not\simeq c \vee b \simeq d \\
(c \not\simeq d \vee a \not\simeq c \vee c \not\simeq d \vee b \simeq d)_{\downarrow} & = & c \not\simeq d \vee a \not\simeq c \vee b \simeq d \\
(c \not\simeq d \vee a \not\simeq d \vee c \not\simeq c \vee b \simeq d)_{\downarrow} & = & c \not\simeq d \vee a \not\simeq c \vee b \simeq d \\
(c \not\simeq d \vee a \not\simeq c \vee b \not\simeq c \vee d \simeq d)_{\downarrow} & = & \emptyset \text{ (tautology)} \\
(c \not\simeq d \vee a \not\simeq c \vee b \not\simeq d \vee c \simeq c)_{\downarrow} & = & \emptyset \text{ (tautology)}
\end{array}
$$

As observed, the additional inferences are either redundant or tautological, which makes c$\mathcal{SP}$ the more efficient calculus. ♣

Redundancy Elimination Rule

Redundancy testing is done as for the standard superposition calculus, except that the constraints must be taken into account. In particular, it is necessary to make sure that the constraints of the redundant c-clause include those of the considered c-clauses.

Definition 3.9 A c-clause $[C \mid \mathcal{X}]$ is *redundant* w.r.t. a set of c-clauses S if either $\mathcal{X}$ is unsatisfiable or there exist c-clauses $[D_i \mid \mathcal{Y}_i] \in S$ $(1 \leq i \leq n)$ such that $\forall i \in \{1 \ldots n\}\, C \succeq D_i$, $\forall i \in \{1 \ldots n\}\, \mathcal{Y}_i \subseteq \mathcal{X}$ and $\mathcal{X}', D_1, \ldots, D_n \models C$, where $\mathcal{X}'$ denotes the set of literals in $\mathcal{X}$ that are smaller than C. ◊

The *redundancy elimination rule* is the same as for standard clauses. For example, if $\mathcal{X}$ is unsatisfiable, then any clause $[C \mid \mathcal{X}]$ is redundant in any set. In practice, we extend the notion of E-subsumption to perform this test.

Definition 3.10 A c-clause $[C \mid \mathcal{X}]$ C-*subsumes* a clause $[D \mid \mathcal{Y}]$ (written $[C \mid \mathcal{X}] \leq_{\mathrm{C}} [D \mid \mathcal{Y}]$) iff $C \preceq D$, $C \leq_{\mathrm{E}} D$ and $\mathcal{X} \subseteq \mathcal{Y}$. ◊

Proposition 3.11 *If* $[C \mid \mathcal{X}] \leq_{\mathrm{C}} [D \mid \mathcal{Y}]$ *then* $[D \mid \mathcal{Y}]$ *is redundant with respect to* $\{[C \mid \mathcal{X}]\}$.

Proof. It suffices to remark that if $C \leq_{\mathrm{E}} D$, then $C \models D$ (Proposition 1.21). ■

Note that both parts of the c-clauses are handled in different ways: the inclusion relation $\subseteq$ used to compare constraints is clearly stronger than the E-subsumption relation $\leq_{\mathrm{E}}$ used for clauses, even enriched with an ordering constraint as here. For instance we have:

$$[a \not\simeq b \vee b \simeq d \mid \top] \leq_{\mathrm{C}} [a \not\simeq c \vee b \not\simeq c \vee c \simeq d \mid \top],$$

but

$$[\Box \mid a \simeq b \wedge b \not\simeq d] \not\leq_{\mathrm{C}} [\Box \mid a \simeq c \wedge b \simeq c \wedge c \not\simeq d].$$

Remark 3.12 *The definition of* C*-subsumption can be relaxed while preserving the truth of Proposition 3.11, so as to detect more redundancies, by replacing the condition* $C \leq_{\mathrm{E}} D$ *with* $C \leq_{\mathrm{E}} D \vee \mathcal{X}'$ *or* $C \leq_{\mathrm{E}} D \vee \mathcal{Y}'$ *where* $\mathcal{X}' = \{l \in \mathcal{X} \mid l \preceq C\}$ *and* $\mathcal{Y}' = \{l \in \mathcal{Y} \mid l \preceq C\}$. *For example, let us consider the clauses* $[C \mid a \simeq b]$ *and* $[D \mid a \simeq b]$ *where* $a \prec b$, $a \not\simeq b \preceq C$ *and* $C = D[a/b]$. *Clearly,* $[D \mid a \simeq b]$ *is redundant to* $[C \mid a \simeq b]$, *but with our current definition* $[C \mid a \simeq b] \not\leq_{\mathrm{C}} [D \mid a \simeq b]$ *because* $C \not\leq_{\mathrm{E}} D$. *In contrast,* $C \leq_{\mathrm{E}} D \vee a \not\simeq b$.

The impact of these modifications on the efficiency of the global method is unclear. On the one hand, a gain is expected due to a better detection of redundant clauses. On the other hand, their integration is not straightforward and may result in a loss of efficiency. These approaches have not been studied during the thesis and remain as future work.

In what follows we will prove the following result:

Theorem 3.13 *c*$\mathcal{SP}_0$ *is sound and deductive-complete.*

Soundness and completeness of c$\mathcal{SP}_0$

Lemma 3.14 *Let* $[C \mid \mathcal{X}]$ *be a c-clause derived from n premises* $[D_i \mid \mathcal{Y}_i]$ *with* $i \in \{1 \dots n\}$. *Then* C *is a logical consequence of* $D_1, \dots, D_n, \mathcal{X}$ *and for all* i, $\mathcal{Y}_i \subseteq \mathcal{X}$.

Proof. It is easy to verify that this property holds for each inference rule, the result follows by a straightforward induction on the length of the derivation. ■

Lemma 3.14 permits to deduce the following soundness result:

Corollary 3.15 *For any c-clause* $[C \mid \mathcal{X}]$ *deducible from a set of clauses* S *(i.e. c-clauses with empty constraint),* C *is a logical consequence of* $S \cup \mathcal{X}$. *In particular, if* $C = \Box$ *then* $S \models \mathcal{X}^c$.

We now prove that the calculus is deductive-complete, i.e., that it permits to generate every prime implicate of a given set of clauses. The proof relies on the following definitions and proposition.

Definition 3.16 For every set of c-clauses S and for every constraint $\mathcal{X}$, we denote by $S|_{\mathcal{X}}$ the set of clauses D (without constraint) such that $[D \,|\, \mathcal{Y}] \in S$ and $\mathcal{Y} \subseteq \mathcal{X}$. ◊

The following proposition is an immediate consequence of the definition.

Proposition 3.17 *Let S be a set of c-clauses in $\mathbb{E}_0$ and let $\mathcal{X}$ be a satisfiable constraint. If a c-clause $[C \,|\, \mathcal{Y}]$ is redundant in S and $\mathcal{Y} \subseteq \mathcal{X}$ then C is redundant in $S|_{\mathcal{X}} \cup \mathcal{X}$.*

Proof. By definition of the c-clause redundancy, there are two cases to consider.

— The first condition leading to redundancy is that $\mathcal{Y}$ is unsatisfiable. In this case, since $\mathcal{Y}$ is a conjunction of literals and $\mathcal{Y} \subseteq \mathcal{X}$, the constraint $\mathcal{X}$ is also unsatisfiable which contradicts the hypotheses.

— In the second case, there exist n c-clauses $[D_i \,|\, \mathcal{Y}_i] \in S$ $(1 \leq i \leq n)$ such that $\forall i \in [1,n]\, C \succeq D_i, \forall i \in [1,n]\, \mathcal{Y}_i \subseteq \mathcal{Y}$ and $\mathcal{Y}', D_1, \ldots, D_n \models C$, where $\mathcal{Y}'$ denotes the set of literals in $\mathcal{Y}$ that are lower than C. Since $\mathcal{Y} \subseteq \mathcal{X}$ we deduce that $\forall i \in [1,n]\, \mathcal{Y}_i \subseteq \mathcal{X}$, hence $\forall i \in [1,n]\, D_i \in S|_{\mathcal{X}}$. Since $\mathcal{Y}', D_1, \ldots, D_n \models C$ where $\mathcal{Y}', D_1, \ldots, D_n \preceq C$, and $\mathcal{Y}' \cup \{D_1, \ldots, D_n\} \subseteq S|_{\mathcal{X}} \cup \mathcal{X}$ then C is redundant in $S|_{\mathcal{X}} \cup \mathcal{X}$. ■

Definition 3.18 A set of c-clauses S is *saturated w.r.t. a constraint $\mathcal{X}$* if every c-clause $[C \,|\, \mathcal{Y}]$ such that $\mathcal{Y} \subseteq \mathcal{X}$ that is deducible from S by applying once one of the inference rules is redundant w.r.t. S. ◊

Theorem 3.19 *Let $\mathcal{X}$ be a normalized satisfiable constraint. Let S be a set of standard clauses and $S^\star$ be a set obtained from S by $c\mathcal{SP}_0$. If $S^\star$ is saturated w.r.t. $\mathcal{X}$ and $S \models \mathcal{X}^c$, then there exists a constraint $\mathcal{Y} \subseteq \mathcal{X}$ such that $[\Box \,|\, \mathcal{Y}] \in S^\star$.*

Proof. Let $S' = S^\star|_{\mathcal{X}} \cup \mathcal{X}$. We first remark that S' is unsatisfiable. Indeed, $S^\star|_\top \models S$ since by Proposition 3.17 all the standard clauses that are removed from S during the saturation process must be redundant in $S^\star|_\top$; furthermore, $S^\star|_\top \subseteq S^\star|_{\mathcal{X}}$, so that $S' \models S^\star|_\top \cup \mathcal{X} \models S \cup \mathcal{X} \models \mathcal{X}^c \cup \mathcal{X}$. We now prove that S' is saturated (in the standard way, see Definition ii.25). We only consider the case where the Superposition rule is applied, the proof for the other rules is similar. Let $l \simeq r \vee P_1$ and $l \bowtie u \vee P_2$ be two clauses occurring in S', where $l \succ r, u$, and assume that $l \simeq r$ and $l \bowtie u$ are selected in $l \simeq r \vee P_1$ and $l \bowtie u \vee P_2$ respectively. Let $r \bowtie u \vee P_1 \vee P_2$ be the clause deduced by superposition from the two previous clauses. We distinguish several cases.

— If both $l \simeq r \vee P_1$ and $l \bowtie u \vee P_2$ occur in $S^\star|_\mathcal{X}$, then S contains two c-clauses of the form $[l \simeq r \vee P_1 \,|\, \mathcal{X}_1]$ and $[l \bowtie u \vee P_2 \,|\, \mathcal{X}_2]$, where $\mathcal{X}_1, \mathcal{X}_2 \subseteq \mathcal{X}$. It is clear than the Superposition rule applies on these c-clauses, yielding $[(r \bowtie u \vee P_1 \vee P_2)_\downarrow \mid \mathcal{X}_1 \wedge \mathcal{X}_2]$. Since $S^\star$ is saturated w.r.t. $\mathcal{X}$, this c-clause is redundant w.r.t. $S^\star$, and since $\mathcal{X}_1 \wedge \mathcal{X}_2 \subseteq \mathcal{X}$ we deduce by Proposition 3.17 that $(r \bowtie u \vee P_1 \vee P_2)_\downarrow$ is redundant w.r.t. S', hence that $r \bowtie u \vee P_1 \vee P_2$ is also redundant.

— If $l \simeq r \vee P_1$ occurs in $S^\star|_\mathcal{X}$ and $l \bowtie u \vee P_2$ occurs in $\mathcal{X}$, then by definition P_2 must be empty, and $S^\star$ contains a c-clause of the form $[l \simeq r \vee P_1 \mid \mathcal{X}_1]$ with $\mathcal{X}_1 \subseteq \mathcal{X}$. Assume that $\bowtie = \not\simeq$. Then the Negative Assertion rule applies on the latter clause, yielding $[(r \not\simeq u \vee P_1)_\downarrow \mid \mathcal{X}_1 \wedge l \not\simeq u]$. Since $l \not\simeq u \in \mathcal{X}$, this c-clause must be redundant in S, and Proposition 3.17 permits to deduce that $(r \not\simeq u \vee P_1)_\downarrow$ is redundant in $S|_\mathcal{X}$. If $\bowtie = \simeq$, then the Positive Assertion rule applies on $[l \simeq r \vee P_1 \mid \mathcal{X}_1]$, yielding $[(r \simeq u \vee P_1)_\downarrow \,|\, \mathcal{X}_1 \wedge l \simeq u]$ and the result follows as in the previous case.

— If $l \simeq r \vee P_1$ occurs in $\mathcal{X}$ and $l \bowtie u \vee P_2$ occurs in $S^\star|_\mathcal{X}$, then the proof is similar to the previous case (using only the Positive Assertion rule).

— If both $l \simeq r \vee P_1$ and $l \bowtie u \vee P_2$ occur in $\mathcal{X}$, then $\mathcal{X}$ is not normalized since l occurs at least twice in $\mathcal{X}$, and also occurs as the maximal term of some equation, which contradicts the hypotheses of the theorem.

Since S' is unsatisfiable and saturated, this set necessarily contains $\square$ by completeness of the standard superposition calculus, which entails that $\square \in S^\star|_\mathcal{X}$ (since the constraints in $\mathcal{X}$ are unit and thus cannot be empty), hence the result. ∎

Remark 3.20 *Note that considering only normalized constraints is not restrictive since any constraint is equivalent to a normalized one. Moreover our goal is to eventually generate implicates that are in normal form, thus all c-clauses whose constraints are not in normal form can be discarded (since these constraints occur in all the descendants). This strategy strongly restricts the search space. For instance no rule will apply on* $[a \simeq b \,|\, c \simeq d]$ *and* $[a \not\simeq b \,|\, c \simeq e]$ *because the obtained constraint* $c \simeq d \wedge c \simeq e$ *is not in normal form. Note also that the handling of clauses and constraints differ: we* normalize *the clausal part of the c-clause, whereas we merely* check *that the constraint is in normal form.*

3 Extension to uninterpreted functions

We now extend the calculus $\mathrm{c}\mathcal{SP}_0$ to formulæ containing function symbols, i.e. to $\mathbb{E}_1$. The rules are almost the same, except that the Assertion rules must be adapted to allow superposition inferences at deep positions in asserted hypotheses. This entails that the branching factor of these rules is now infinite, since there are infinitely many terms and clauses (and also infinitely many prime implicate).

Definition 3.21 The standard inference rules are adapted to c-clauses in $\mathbb{E}_1$.

$$\text{Superposition} \quad \frac{[r \simeq l \vee C \,|\, \mathcal{X}] \quad [u \bowtie v \vee D \,|\, \mathcal{Y}]}{[u[l] \bowtie v \vee C \vee D \,|\, \mathcal{X} \wedge \mathcal{Y}]}\ (1.),$$

$$\text{Factoring} \quad \frac{[t \simeq u \vee t \simeq v \vee C \,|\, \mathcal{X}]}{[t \simeq v \vee u \not\simeq v \vee C \,|\, \mathcal{X}]}\ (2.),$$

with the following conditions:

1. $u_{|p} = r$, $r \succ l$, $u \succ v$, and $(r \simeq l)$ and $(u \bowtie v)$ are selected in $(r \simeq l \vee C)$ and $(u \bowtie v \vee D)$ respectively;
2. $t \succ u$, $t \succ v$ and $(t \simeq u)$ is selected in $t \simeq u \vee t \simeq v \vee C$.

Once again, the clausal part of the c-clauses generated is systematically normalized, however, the constraint part is not. ◊

Definition 3.22 The Assertion rules of the constrained calculus are extended to $\mathbb{E}_1$ in the following way:

$$\text{Positive Assertion} \quad \frac{[u \bowtie v \vee C \,|\, \mathcal{X}]}{[u[s] \bowtie v \vee C \,|\, \mathcal{X} \wedge t \simeq s]}\ (1.),$$

$$\text{Negative Assertion}\,[3] \quad \frac{[t \simeq s \vee C \,|\, \mathcal{X}]}{[u[s] \bowtie v \vee C \,|\, \mathcal{X} \wedge u \bowtie v]}\ (2.),$$

with the following conditions:

1. $u_{|p} = t$, $t \succ s$, $u \succ v$ and $(u \bowtie v)$ is selected in $(u \bowtie v \vee C)$;
2. $u_{|p} = t$, $t \succ s$, $u \succ v$, and $(t \simeq s)$ is selected in $(t \simeq s \vee C)$.

In the same way as for the standard rules, the clausal part of the generated c-clauses is systematically normalized. ◊

Note that the term u in the Negative Assertion rule is arbitrary, in particular there are infinitely many possible terms. Let us consider for example the c-clause $[b \simeq a \,|\, \mathcal{X}]$ defined over a signature containing the unary function symbol f. In this case, the terms b, $f(b)$, $f(f(b))$ and more generaly $f^n(b)$ for $n \in \mathbb{N}$ can all be used as the u of the Negative Assertion rule.

Definition 3.23 The constrained superposition calculus in $\mathbb{E}_1$, denoted c$\mathcal{SP}$, is composed of the rules of Definition 3.21 and 3.22. ◊

The principle of this calculus remains unchanged and the same definitions are used for redundancy detection as in $\mathbb{E}_0$.

Example 3.24 The following example shows how to derive the implicate $a \not\simeq d \vee f(c) \simeq f(b)$ from $\{a \simeq b, f(c) \simeq f(d)\}$, given the term ordering $a \prec b \prec c \prec$

3. The term "Negative" is not really appropriate to describe this assertion rule but is kept by analogy with c$\mathcal{SP}_0$

$d \prec f(a) \prec f(b) \prec f(c) \prec f(d)$.

1	$[f(c) \simeq f(d) \,\vert\, \top]$	(hyp)
2	$[f(c) \simeq f(a) \,\vert\, a \simeq d]$	(Pos. AR, 1)
3	$[f(a) \not\simeq f(b) \,\vert\, a \simeq d \wedge f(c) \not\simeq f(b)]$	(Neg. AR, 2)
4	$[a \simeq b \,\vert\, \top]$	(hyp)
5	$[f(a) \not\simeq f(a) \,\vert\, a \simeq d \wedge f(c) \not\simeq f(b)]$	(Sup. 3, 4)
6	$[\Box \,\vert\, a \simeq d \wedge f(c) \not\simeq f(b)]$	(Ref. 5)

The negation of $a \simeq d \wedge f(c) \not\simeq f(b)$ is the desired implicate. ♣

Theorem 3.25 *$c\mathcal{SP}$ is sound and deductive complete.*

Soundness and completeness of c$\mathcal{SP}$

The soundness proof of c$\mathcal{SP}_0$ is exactly the same as that of c$\mathcal{SP}_0$ (detailed in Lemma 3.14 and Corollary 3.15).

The schema of the deductive-completeness proof is the same as for c$\mathcal{SP}_0$. In particular, we reuse Definition 3.16 and Proposition 3.17 but in the context of $\mathbb{E}_1$. The notion of saturation w.r.t. a constraint is also unchanged (Definition 3.18).

Theorem 3.26 *Let $\mathcal{X}$ be a normalized satisfiable constraint and let S be a set of standard clauses in $\mathbb{E}_1$. Let $S^\star$ be a set obtained from S by $c\mathcal{SP}$. If $S^\star$ is saturated w.r.t. $\mathcal{X}$ and $S \models \mathcal{X}^c$, then there exists a constraint $\mathcal{Y} \subseteq \mathcal{X}$ such that $[\Box \,\vert\, \mathcal{Y}] \in S^\star$.*

Proof sketch. This proof has very few differences with that of Theorem 3.19 (deductive-completeness of c$\mathcal{SP}_0$). In the application of the Superposition rule, we consider the clauses $l \simeq r \vee P_1$ and $u \bowtie v \vee P_2$ where $l = u_{\vert p}$, $r \prec l$, $v \prec u$ and assume that $r \simeq l$ and $u \bowtie v$ are selected in $l \simeq r \vee P_1$ and $u \bowtie v \vee P_2$ respectively (instead of $l \simeq r \vee P_1$ and $l \simeq u \vee P_2$). The disjunction of cases that follows is unchanged but for two details:

- In the second case, when assuming that $\bowtie{=}\simeq$, the Negative Assertion rule must be invoked to generate the desired result instead of the Positive Assertion rule.
- For the last case, where both $l \simeq r \vee P_1$ and $u \bowtie v \vee P_2$ occur in $\mathcal{X}$, a contradiction to the hypothesis '$\mathcal{X}$ is in normal form' is raised using the following arguments: by Definition 1.1, $(u \bowtie v)^c{}_{\downarrow \mathcal{X}^c \setminus (u \bowtie v)^c} \preceq (u[l] \bowtie v)^c$ and $(u[l] \bowtie v)^c \neq (u \bowtie v)^c$, which contradicts point 6 of Definition 1.11 in the case $\bowtie{=}\simeq$ and point 2 if $\bowtie{=}\not\simeq$.

The rest of the proof is unchanged.

A seemingly natural idea is to relax the condition of Definition 3.10 by testing logical entailment instead of set inclusion when comparing constraints (i.e., replacing $\mathcal{Y}_i \subseteq \mathcal{X}$ by $\mathcal{Y}_i \models \mathcal{X}$. However, this makes the calculus incomplete. More precisely, this relaxed notion of redundancy is not compatible with the

previous restriction concerning the removal of clauses with non-normalized constraints. Experiments show that the restriction make the calculus more efficient, even with a more restrictive version of the redundancy elimination rule.

4 Restricting the class of implicates generated [4]

The number of implicates of a given formula is usually huge, and it is important in practice to be able to prune the search space by imposing additional restrictions on the implicates that are searched for. They can be for instance syntactic restrictions, e.g., if the user is interested only in positive implicates, or in implicates of small size. They can also be of semantic nature, for instance, the user may want to obtain all implicates that entail some formula. Using c$\mathcal{SP}$, this is possible if the considered class of implicates satisfies the following condition.

Definition 3.27 A set of constraints $\mathfrak{X}$ is $\subseteq$-*closed* (read "closed by inclusion") when for every $\mathcal{X} \in \mathfrak{X}$ and constraint $\mathcal{Y}$, if $\mathcal{Y} \subseteq \mathcal{X}$ then $\mathcal{Y} \in \mathfrak{X}$. The set $\mathfrak{X}$ is *normalized* if every constraint $\mathcal{X} \in \mathfrak{X}$ is normalized. ◊

Definition 3.28 Let $\mathfrak{X}$ be a set of constraints. A set of c-clauses S is $\mathfrak{X}$-*saturated* if every c-clause $[D \,|\, \mathcal{X}]$ such that $\mathcal{X} \in \mathfrak{X}$ that is deducible from S by applying one of the inference rules is redundant w.r.t. S. ◊

Theorem 3.29 *Let $\mathfrak{X}$ be a normalized and $\subseteq$-closed set of satisfiable constraints. Let S be a set of standard clauses (i.e. c-clauses with empty constraint) and $S^\star$ be a set of clauses obtained from S by c$\mathcal{SP}$. If $S^\star$ is $\mathfrak{X}$-saturated and $S \models \mathcal{X}^c$ for some $\mathcal{X} \in \mathfrak{X}$, then there exists $\mathcal{Y} \subseteq \mathcal{X}$ such that $[\Box \,|\, \mathcal{Y}] \in S^\star$.*

Remark 3.30 *The proof of Theorem 3.29 derives from that of Theorem 3.26 which is in essence a simplified variant of Theorem 3.29. Intuitively, the importance of the $\subseteq$-closed property can be understood by looking at the way new c-clauses are generated and at the growth of the constraints through the inference process. There are only two ways to modify a constraint. One is to use an assertion rule, adding (at most) one literal to the considered constraint. The other is to use the superposition rule which combines the constraints of the parents in the consequent c-clause. Now let us consider the non $\subseteq$-closed set made of the constraints of size exactly two literals. The generation of all such prime implicates of a formula and only these is not possible using c$\mathcal{SP}$ (except when this set is empty, like for contradictory formulæ). The reason is that the inference process starts with c-clauses with empty constraints and at least two assumptions are necessary to generate a first c-clause in the desired set. Thus these c-clauses are never generated, because the "intermediary steps" are systematically discarded. The $\subseteq$-closed property ensures that all such intermediate steps also possess the desired property, thus guaranteeing the completeness of the method.*

4. The results of this section are direclty presented for $\mathbb{E}_1$ but they also hold for $\mathbb{E}_0$.

Among others, interesting examples of $\subseteq$-closed sets of constraints, also referred to as *filters*, include positive constraints, negative constraints and constraints of size at most some fixed $k \in \mathbb{N}$. Moreover, since $\subseteq$-closeness is stable by intersection, it is possible to combine all these different filters. Practical examples are given in Section 4 of Chapter 10.

Another interesting filter relates to the simplification of formulæ. Contrarily to the already presented examples, it is not based on a syntactic criteria but rather on a semantic one. For any set of clauses S, the set of constraints $\mathfrak{S}$ such that $\mathcal{X} \in \mathfrak{S}$ if and only if there exists $C \in S$ with $\mathcal{X}^c \models C$, is $\subseteq$-closed. This remark allows us to use the filtering mechanism to efficiently compute a minimal (up to redundancy) equivalent representation of any set of clauses S. This is done as follows. First, S is $\mathfrak{S}$-saturated, and the set I of clauses $\mathcal{X}^c$ such that $[\Box \,|\, \mathcal{X}]$ occurs in the saturated set is constructed. By Theorem 3.29, I is the set of prime implicates of S that occur in $\mathfrak{S}$, i.e., that entail at least one clause in S. Then, for each clause $C \in S$, an implicate $C' \in I$ such that $C' \models C$ is selected [5]. The obtained clause set is equivalent to S and minimal in the sense that all the clauses are minimal w.r.t. logical entailment (in particular no literal can be deleted without affecting the semantics). This simplification method departs from the one described in [20] in which formulæ are reduced by removing literals occurring in them, provided they are useless in the context. For instance the literal l can be removed in $(l \vee \psi) \wedge \phi$ if $\phi, \neg\psi \models l$. Our technique allows for finer simplifications, taking into account equational axioms.

Example 3.31 In $\mathbb{E}_0$, using the ordering $a \prec b \prec c \ldots$, consider the clause set $S = \{a \not\simeq c \vee b \not\simeq c \vee d \simeq e, a \simeq c \vee a \simeq f, b \simeq c \vee a \simeq f, f \not\simeq b\}$. It is easy to check that $a \not\simeq b \vee d \simeq e$ is an implicate of S and this clause E-subsumes $a \not\simeq c \vee b \not\simeq c \vee d \simeq e$. Our approach computes the clause set $S' = \{a \not\simeq b \vee d \simeq e, a \simeq c \vee a \simeq f, b \simeq c \vee a \simeq f, f \not\simeq b\}$ that is equivalent to S and strictly smaller. In contrast, the approach in [20] cannot simplify S since there is no useless literal. ♣

5 Summary

In this chapter we introduced the c$\mathcal{SP}$ calculus for generating the prime implicates of a formula in $\mathbb{E}_0$ and $\mathbb{E}_1$. This calculus "freezes" the literals that represent asserted equalities, preventing further inferences to modify them. Restricted to $\mathbb{E}_0$, the strengths and weaknesses of c$\mathcal{SP}_0$ are very different from that of $\mathcal{K}$-paramodulation. On the one hand, the global search space of c$\mathcal{SP}_0$ is exponentially larger but on the other hand it is greatly reduced by the addition of ordering constraints to the rules of c$\mathcal{SP}_0$, that cannot be applied to the rules of $\mathcal{K}$-paramodulation without loss of completeness. An experimental comparison of the two calculi is presented Part III. The extension of c$\mathcal{SP}_0$ to $\mathbb{E}_1$ (namely c$\mathcal{SP}$) is completely straightforward and a great asset of this calculus. Another advan-

5. Such a clause necessarily exists since C itself is an implicate of S – albeit not necessarily minimal.

tage of c$\mathcal{SP}$ is the possibility to filter the implicates generated while preserving a partial notion of completeness. Using such filters, all the prime implicates verifying some criteria and only these ones are generated, which is of major interest in practice given the size of the targeted formulæ.

Part II

Redundancy Detection

This part of the thesis introduces a method to efficiently store and manipulate clauses and c-clauses in $\mathbb{E}_0$ and $\mathbb{E}_1$. The first chapter introduces clausal trees, data structures that permits the storage of (c-)clauses with a partial sharing of common literals, as well as efficient manipulations of the stored clauses. These manipulations (testing if a clause is redundant to a clause in a tree and removing redundant clauses from a tree) are formalized as algorithms respectively in the second and third chapter. This method is adapted from de Kleer's CLTMS algorithm for propositional logic [17] that is briefly introduced in the state of the art (Chapter i Section 1.2). The last chapter presents the extension of these manipulations to c-clauses.

Chapter 4

Clausal Trees

Prime implicate saturation methods will typically infer huge sets of clauses. It is thus essential to devise good data-structures for storing and retrieving the generated clauses, in such a way that the redundancy criterion (i.e. I-subsumption or E-subsumption) introduced in Part I, Chapter 1 (see Definitions 1.17 and 1.23) can be tested efficiently. We devise for this purpose a tree data structure, called a clausal tree, specifically tailored to store sets of clauses while taking into account the usual properties of the equality predicate. The goal is to share common prefixes of stored clauses, in a way that allows redundancy elimination algorithms to be applied simultaneously on all shared literals. Clausal trees are similar to the tries presented Chapter i corresponding to the CLTMS algorithm [17]. In such trees, the clauses are represented by the branches, i.e., by the disjunction of the literals labeling the edges from root to leaf. To ensure that operations on trees and clauses are as simple as possible, and that literals are well shared, restrictions are added to the trees.

Definition 4.1 In $\mathbb{E}_0$ and in $\mathbb{E}_1$, a *clausal tree* is inductively defined as either $\Box$, or a finite set of pairs of the form (l, T') where l is a literal and T' a clausal tree. Intuitively, a pair (l, T') represents an edge from the root of the tree to T', labeled by literal l. In addition, a clausal tree T with $(l, T') \in T$ must respect the following conditions:

— for all l' appearing in T', $l <_\pi l'$ [1],

— there is no clausal tree T'' such that $T'' \neq T'$ and $(l, T'') \in T$.

The set of clauses represented by a clausal tree T is denoted by $\mathcal{C}(T)$ and defined inductively as follows:

$$\mathcal{C}(T) = \begin{cases} \{\Box\} & \text{if } T = \Box, \\ \bigcup_{(l,T') \in T} \left(\bigcup_{D \in \mathcal{C}(T')} l \vee D \right) & \text{otherwise.} \end{cases}$$

◊

1. For the definition of $<_\pi$, see Definition ii.10

Remark 4.2 *By considering propositional literals instead of equational ones, the definition of a clausal tree becomes that of a propositional* trie.

Remark 4.3 *Employing the ordering $<_\pi$ to constrain the order in which literals occur along the branches of the trees has two uses*
- *it limits the number of repetitions of the same literal,*
- *it simplifies the application of the redundancy elimination algorithms presented in the subsequent chapters.*

The first point can be enforced using any total ordering on literals, but to ensure the second point, the use of $<_\pi$ is a necessity.

As the definition implies, leaves can be either $\Box$ or $\emptyset$, but in practice if a leaf is $\emptyset$ then the corresponding branch is irrelevant because by definition the trees T and T', where T' is T without the branches containing $\emptyset$, are such that $\mathcal{C}(T) = \mathcal{C}(T')$. In other words, a pair $(l, \emptyset)$ can be deleted from the tree without affecting $\mathcal{C}(T)$. The only exception is the empty tree, in which the root is labeled with $\emptyset$. Note that by definition $\mathcal{C}(\emptyset) = \emptyset$.

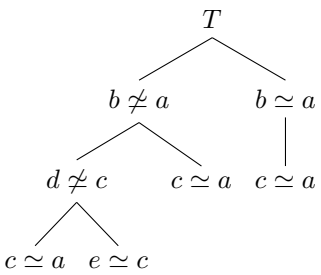
A functions describing the size of a tree is defined.

Definition 4.4 Let T be a clausal tree.

$$\text{size}(T) = \begin{cases} 0 & \text{if } T = \Box, \\ \sum\limits_{(l,T') \in T} 1 + \text{size}(T') & \text{otherwise.} \end{cases}$$

$\Diamond$

Example 4.5 The structure T in Figure 9 is a clausal tree in $\mathbb{E}_0$ with the constant ordering $a \prec b \prec \ldots$. For readability the labels are associated with the nodes rather than with the edges leading to them.



The represented clauses $\mathcal{C}(T)$ are:

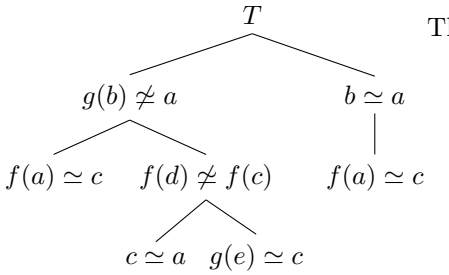
$b \not\simeq a \vee d \not\simeq c \vee c \simeq a$
$b \not\simeq a \vee d \not\simeq c \vee e \simeq c$
$b \not\simeq a \vee c \simeq a$
$b \simeq a \vee c \simeq a$

Figure 9 – A clausal tree in $\mathbb{E}_0$

Formally, this tree is defined as $\{(b \not\simeq a, T''), (b \simeq a, T')\}$, with

$$T' = \{(c \simeq a, \Box)\}$$
$$T'' = \{(d \not\simeq c, \{(c \simeq a, \Box), (e \simeq c, \Box)\}), (c \simeq a, \Box)\}.$$

♣



The clauses in $\mathcal{C}(T)$ are:

$g(b) \not\simeq a \vee f(a) \simeq c$
$g(b) \not\simeq a \vee f(d) \not\simeq f(c) \vee c \simeq a$
$g(b) \not\simeq a \vee f(d) \not\simeq f(c) \vee g(e) \simeq c$
$b \simeq a \vee f(a) \simeq c$

♣

Figure 10 – A clausal tree in $\mathbb{E}_1$

Example 4.6 The structure T in Figure 10 is a clausal tree in $\mathbb{E}_1$ with the term order $a \prec b \prec c \prec g(c) \prec g(e) \prec f(c) \prec f(d)$. Again the labels are associated with the nodes rather than with the edges leading to them.

Definition 4.7 In $\mathbb{E}_0$ and in $\mathbb{E}_1$, a clausal tree is in *normal form* (or simply *normal*) if all the clauses in $\mathcal{C}(T)$ are in normal form. ◊

In $\mathbb{E}_0$, it is possible to impose additional conditions on clausal trees in order to ensure that the represented clauses are in normal form.

Proposition 4.8 *In $\mathbb{E}_0$, a clausal tree T is a normal clausal tree iff for any pair (l, T') in T, all the following conditions hold:*
- *l is not of the form $a \simeq a$ or $a \not\simeq a$,*
- *if $l = a \not\simeq b$ with $a \prec b$ then b does not occur in T',*
- *T' is a normal clausal tree.*

It is easy to see that if T verifies the conditions of the previous proposition then all the clauses in $\mathcal{C}(T)$ are in normal form. The tree of Example 4.5 satisfies these requirements. For example the constants b and d do not occur below the literals $b \not\simeq a$ and $d \not\simeq c$ respectively. In $\mathbb{E}_1$, the more complex definition of the clausal normal form prevents us from defining simple syntactic criteria for normal clausal trees.

Notation 4.9 Let C be a clause in normal form in $\mathbb{E}_0$ or $\mathbb{E}_1$ and T be a normal clausal tree such that $\forall D \in \mathcal{C}(T)$, $C \vee D$ is in normal form and $\forall l \in D$, $C <_\pi l$. In this case, $C.T$ denotes the clausal tree such that: if $C = \Box$ then $C.T = T$, otherwise $C.T = \{(l_1, C'.T)\}$ where $C = l_1 \vee C'$ and $l_1 = \min_{<_\pi} \{l \in C\}$. ◊

Notation 4.10 We extend the notation $[a/b]$ to trees: For any tree T, $T[a/b]$ denotes the tree obtained from T by replacing all occurrences of b by a. Note that this replacement may interfere with the ordering constraints imposed on clausal trees, hence $\top[a/b]$ may not be a clausal tree even if T is one. ◊

No matter the logic or the nature of the clauses, there are three main operations on clausal trees. The first one consists in checking whether a new clause is redundant w.r.t. an existing one already stored in a clausal tree. In propositional logic, this operation corresponds to Algorithm 2

Algorithm 2 isEntailedProp(C, T)

Require: C is a **propositional** clause and T is a **trie**.
Ensure: isEntailedProp$(C, T) = \top \Leftrightarrow \exists D \in \mathcal{C}(T), D$ subsumes C
1: **if** $T = \square$ **then**
2: **return** $\top$
3: **end if**
4: **if** $C = \square$ **then**
5: **return** $\bot$
6: **end if**
7: $m_1 \leftarrow \min_{<} \{m \in C\}$ // *given* $<$*, an ordering on propositional literals*
8: $T_1 \leftarrow \{(p, T') \in T \mid p = m_1\}$
9: $T_2 \leftarrow \{(p, T') \in T \mid m_1 < p\}$
10: **return** $\bigvee_{(p,T') \in T_1}$ isEntailedProp$(C \setminus \{m_1\}, T')$
$\vee \bigvee_{(p,T') \in T_2}$ isEntailedProp$(C \setminus \{m_1\}, (p.T'))$

The second one removes from a clausal tree all clauses that are redundant w.r.t. a given clause. In propositional logic, this action is realized by Algorithm 3

Algorithm 3 pruneEntailedProp(C, T)

Require: C is a **propositional** clause, T is a **trie** and isEntailedProp$(C, T) = \bot$.
Ensure: $\forall D \in \mathcal{C}($pruneEntailed$_{i0}(C, T))$, C does not subsume D.
1: **if** $C = \square$ **then**
2: **return** $\emptyset$
3: **end if**
4: **if** $T = \square$ **then**
5: **return** T
6: **end if**
7: $m_1 \leftarrow \min_{<} \{m \in C\}$ // *given* $<$*, an ordering on propositional literals*
8: $T_1 \leftarrow \{(p, T') \in T \mid p = m_1\}$
9: $T_{out1} \leftarrow \{(p,$ pruneEntailedProp$(C \setminus \{m_1\}, T'))|(p, T') \in T_1\}$
10: $T_2 \leftarrow \{(p, T') \in T \mid p < m_1\}$
11: $T_{out2} \leftarrow \{(p,$ pruneEntailedProp$(C, T'))|(p, T') \in T_2\}$
12: **return** $T_{out1} \cup T_{out2} \cup (T \setminus (T_1 \cup T_2))$

The last one is the insertion of a new clause into a clausal tree. This last

operation is straightforward and thus will not be described here. On the other hand, the first two operations are not trivial and significantly more complex in equational logic than in propositional logic. Thus they are carefully described in the remaining chapters of this part of the thesis. So as to expose all the variants of these manipulations, the algorithms are presented once for $\mathbb{E}_0$ and I-subsumption, and a second time for $\mathbb{E}_1$ and E-subsumption. This choice is justified by the fact that I-subsumption is the redundancy criterion associated with the $\mathcal{K}$-paramodulation calculus (presented in Part I, Chapter 2), that is only defined in $\mathbb{E}_0$, while E-subsumption, associated to the c$\mathcal{SP}$ calculus (presented Part I Chapter 3) is useful both in $\mathbb{E}_0$ and $\mathbb{E}_1$. Although c$\mathcal{SP}$ manipulates c-clauses, for simplicity we present the algorithms for standard clauses both in $\mathbb{E}_0$ and in $\mathbb{E}_1$. The final chapter of this part is dedicated to the extension of the standard algorithms presented to c-clauses.

Chapter 5

Entailment by a clausal tree

This chapter presents the first manipulation expected of clausal trees, i.e. a test, given a normal clause C and a normal clausal tree T, that detects if T contains a clause D such that C is redundant with respect to D. This step is essential to the generic saturation process (see Algorithm 1 page 34). No matter the logic or the redundancy criterion considered, this test proceeds under a same principle: a depth-first traversal of T and attempts to project every encountered literal on C. If a literal cannot be projected, the exploration of the subtree associated to this literal is useless, so the algorithm switches to the following literal. As soon as a clause entailing C is found, the traversal halts and $\top$ is returned.

1 Algorithm isEntailed for i-subsumption and $\mathbb{E}_0$

We denote by isEntailed$_{i0}$ the algorithm isEntailed for i-subsumption and $\mathbb{E}_0$ (see Algorithm 4). In this algorithm, the specificity of i-subsumption, namely the injectivity of the mapping between the positive literals of the subsuming and subsumed clauses, appears line 14. Replacing in this line's recursive call the clause $C \setminus \{l\}$ by C is enough to obtain isEntailed$_{e0}$, the same test for e-subsumption. During the traversal of a tree T, the branches (l, T') are dealt with differently depending on l and its relation to C.

— If, considering a branch (l, T'), it is clear that $l \in C$ then the exploration of the branch continues on T'. The sets T_1 and T_3 regroup such branches depending on the circumstances.
— If the relation between l and C is not currently determined (this happens only while $|C^-|$ is not empty), a literal is projected before restarting the exploration of the branch. Such branches are grouped in T_2.
— Lastly, if it is clear that $l \not\models C$, which is the case e.g. when l is $\leq_\pi$-smaller than all the literals in C, then the exploration of the branch is halted.

The requirements of isEntailed$_{i0}$ mention a relaxed notion of normal form for clauses and trees. These are formaly introduced in Definition 5.3 and 5.4.

Algorithm 4 isEntailed$_{i0}(C, T)$

Require: C is a clause in normal form and T is a relaxed normal clausal tree.
Ensure: isEntailed$_{i0}(C, T) = \top \Leftrightarrow \exists D \in \mathcal{C}(T), D \leq_{\text{I}} C$

1: **if** $T = \square$ **then**
2: **return** $\top$
3: **end if**
4: **if** $C = \square$ **then**
5: **return** $\bot$
6: **end if**
7: $m_1 \leftarrow \min_{<_\pi} \{m \in C\}$
8: **if** m_1 is of the form $b \not\succ a$ where $a \prec b$ **then**
9: $T_1 \leftarrow \{(l, T') \in T \mid l = m_1\}$
10: $T_2 \leftarrow \{(l, T') \in T \mid (\nexists c,\ l = b \not\succ c$, where $b \succ c)$ and $m_1 <_\pi l\}$
11: **return** $\bigvee_{(l,T') \in T_1}$ isEntailed$_{i0}(C \setminus \{m_1\}, T')$
$\vee \bigvee_{(l,T') \in T_2}$ isEntailed$_{i0}(C \setminus \{m_1\}, (l.T')[a/b])$
12: **else**
13: $T_3 \leftarrow \{(l, T') \in T \mid l \in C\}$
14: **return** $\bigvee_{(l,T') \in T_3}$ isEntailed$_{i0}(C \setminus \{l\}, T')$
15: **end if**

Remark 5.1 *To implement this algorithm, a different approach can be adopted. Instead of going through the literals of C one by one begining with the $<_\pi$-smallest one as described in Algorithm 4, all the negative literals of C can be recovered and projected onto T at the same time. Then it is possible to deal with the positive literals of C as here. The advantage of using this approach lies in a reduction of the number of comparisons between literals of C and T relatively to Algorithm 4. Its drawback is that all the negative literals of C might not be necessary to detect the absence of* I- *or* E*-subsumption between C and the clauses of $\mathcal{C}(T)$. In other words, the choice of one algorithm over the other should be motivated by the relative cost of projecting literals and comparing them (which was not done in our case since this possibility occured to us long after the developpement of our prototypes presented Part III). In addition, this procedure has no interesting equivalent for the pruning of trees presented in the next chapter, where the roles of C and T are reversed, and where projecting all the negative literals of a branch of T at the same time would be a waste of the tree-like structure of T.*

The input trees in this algorithm may not be normal clausal trees because of the rewriting step at line 11 which does not preserve normal forms as mentioned in Notation 4.9. Example 5.2 illustrates this possibility.

Example 5.2 The following sequence of recursive calls shows how the constraints on the positive literals of a normal clausal tree may no longer hold after a recursive call. Let $T = \{(b \simeq a, \{(c \simeq a, \{(c \simeq b, \Box)\})\})\}$ and $C = c \not\simeq a$ where $a \prec b \prec c$.

1	$\textsc{isEntailed}_{i0}(C, T)$	
2	$\textsc{isEntailed}_{i0}(\Box, \{(b \simeq a, \{(a \simeq a, \{(b \simeq a, \Box)\})\})\})$	(called at line 11)
3	$\bot$	(returned at line 5)

In the second call, the clausal tree $\{(b \simeq a, \{(a \simeq a, \{(b \simeq a, \Box)\})\})\}$ is not in normal form because the literal $a \simeq a$ is a tautology, the literal $b \simeq a$ appears twice and the ordering $<_\pi$ is not respected. ♣

Hence, before proving that the algorithm is correct, we must verify that its requirements are always respected, namely, that for all recursive calls, the input clause is in normal form and the input tree is in *relaxed* normal form. For the clauses, this is obvious because no rewriting operation is ever performed, and literals are deleted from the smallest to the greatest, which is sufficient to ensure that the resulting clause remains in normal form. For the clausal trees, the notion of relaxation (formally introduced in the next definition) describes the kind of tree that occurs inside recursive calls of the algorithm due to the rewriting at line 11, that interferes with the requirements associated to normal clausal trees, as illustrated in Example 5.2. In particular, the positive literals may not respect the ordering constraints or may even become tautological after a rewriting.

Definition 5.3 A *clausal tree in relaxed normal form* is inductively defined as either $\Box$ or a set of pairs (l, T') where l is a literal and T' an relaxed normal clausal tree such that:

- for any pair $(l, T') \in T$, if l is a negative literal then:
  - l is not of the form $a \not\simeq a$;
  - for all literals l' occuring in T', we have $l <_\pi l'$;
  - if $l = b \not\simeq a$, with $a \prec b$, then the constant b does not occur in T';
- for any pair $(l, T') \in T$, if l is a positive literal then all the literals labeling T' are also positive. ◊

Note that a clausal tree in relaxed normal form is not formally a clausal tree, although its definition is very similar. This is due to the relaxation of the ordering constraints on the positive literals it contains.

The clauses represented by a relaxed normal clausal tree are not necessarily in normal form because they can contain tautological literals, as well as multiple occurrences of the same literal, hence the following definition:

Definition 5.4 A clause C is in *relaxed normal form* if $C^- = {C_\downarrow}^-$ and if, moreover, for all positive literals $l \in C^+$, either $l \in {C_\downarrow}^+$ or l is of the form $a \simeq a$. ◊

We also need to introduce additional propositions, stating some basic properties of the clauses and clausal trees in relaxed normal form.

Proposition 5.5 *If $T_1, \ldots, T_n$ are relaxed normal clausal trees distinct from $\square$, then $\bigcup_{i=1}^{n} T_i$ is also a relaxed normal clausal tree.*

Proof. This is a direct consequence of the definition of a relaxed normal clausal tree. ∎

Proposition 5.6 *Let C be a clause in relaxed normal form. For any constant a, we have $a_{\downarrow C} = a$ iff for all literals l occurring in C, if l is of the form $a \not\simeq b$ then $b \succ a$.*

Proof. Let a be a constant and C be a clause in relaxed normal form; assume $a_{\downarrow C} = a$. If $l \in C$ is of the form $a \not\simeq b$ with $a \succeq b$, then by definition $b = a_{\downarrow C} = a$ and l is a contradiction; but this is impossible since C is a clause in relaxed normal form. Now assume that there exists an $l \in C$ such that l is of the form $a \not\simeq b$ with $a \succ b$, then by definition $b = a_{\downarrow C}$, thus $a \succ a_{\downarrow C}$. ∎

Lemma 5.7 proves that the requirements of the isEntailed$_{i0}$ algorithm are met at every recursive call.

Lemma 5.7 *Let C be a clause and T a relaxed normal clausal tree. All the trees appearing in the recursive calls of* isEntailed$_{i0}(C, T)$ *are also relaxed normal clausal trees.*

Proof. (Please refer to the algorithm for the notations.) For any $(l, T') \in T_1 \cup T_3$, T' is a relaxed normal clausal tree by definition. It is also straightforward to see that for any $(l, T') \in T_2$, $l.T'$ is a relaxed normal clausal tree. We then show that the trees of the form $(l.T')[a/b]$ (that appear as arguments of some of the recursive calls at line 11) are relaxed normal clausal trees. Since we consider only trees in T_2, we are in the case where m_1 is of the form $b \not\simeq a$ with $a \prec b$, $m_1 <_\pi l$ and l is not of the form $b \not\simeq c$ with $b \succ c$.

We suppose that: $\forall (l', T'') \in T'$, $(l'.T'')[a/b]$ is a relaxed normal clausal tree. Then by Proposition 5.5, $T'[a/b]$, as the union of relaxed normal clausal trees is also a relaxed normal clausal tree. Moreover, if l is a positive literal then by definition so is any literal l' occurring in T' thus $(l.T')[a/b]$ is also a relaxed normal clausal tree. If l is of the form $u \not\simeq v$ with $u \succ v$, then necessarily $u \succ b$, because $b \not\simeq a <_\pi l$ and $l \neq b \not\simeq c$ with $b \succ c$, thus $l[a/b]$ is not a contradiction. Furthermore, for all literals l' labeling an edge starting from the root of T', if l' is positive, then by definition of the order on literals, $l[a/b] <_\pi l'[a/b]$. If l' is negative, then $l' = s \not\simeq t$ with $s \succ u$ (and $s \succ t$), so $s \succ b$, hence $l[a/b] <_\pi l'[a/b]$. In addition, since u does not appear in T', it does not appear in $T'[a/b]$ either, because $u \neq a$). Since all the properties are verified, we conclude that $(l.T')[a/b]$ is a relaxed normal clausal tree. ∎

The following theorem states the main property of isEntailed$_{i0}$.

Theorem 5.8 *If C is a clause in normal form and T is a relaxed normal clausal tree then* isEntailed$_{i0}(C, T)$ *returns $\top$ iff $\mathcal{C}(T)$ contains a clause D such that $D \leq_{\mathrm{I}} C$.*

PROOF. We first assume that $\text{isEntailed}_{i0}(C, T) = \top$ and show by induction on size(T) that there exists a clause $D \in \mathcal{C}(T)$ such that $D \leq_{\text{I}} C$. We examine all the cases in which $\text{isEntailed}_{i0}(C, T)$ returns $\top$ in their order of appearance in the algorithm.

- If $T = \square$ then it represents the empty clause and since $\square \leq_{\text{I}} C$, the property holds.
- Assume $m_1 = \min_{<_\pi} \{l_i \in C\}$ is of the form $b \not\succ a$ with $a \prec b$.
  - If there exists a $(l, T') \in T_1$, i.e. such that $l = m_1$, and if the call $\text{isEntailed}_{i0}(C \setminus \{m_1\}, T')$ returns $\top$, then by induction, there exists a $D \in \mathcal{C}(T')$ such that $D \leq_{\text{I}} C \setminus \{m_1\}$. Therefore $m_1 \vee D \leq_{\text{I}} C$ and since $m_1 \vee D \in \mathcal{C}(T)$, we have the result.
  - Suppose there exists a $(l, T') \in T_2$, i.e. a pair $(l, T') \in T$ such that l is not of the form $b \not\succ c$ with $b \succ c$ and $\text{isEntailed}_{i0}(C \setminus \{m_1\}, l.T'[a/b]) = \top$. Then by induction there exists a clause $D' \in \mathcal{C}(l.T'[a/b])$ such that $D' \leq_{\text{I}} C \setminus \{m_1\}$, and therefore there exists a $D \in \mathcal{C}(l.T')$ such that $D[a/b] \leq_{\text{I}} C \setminus \{m_1\}$. Thus we must have $D \leq_{\text{I}} C$ and since $\mathcal{C}(l.T') \subseteq \mathcal{C}(T)$, the property is verified.
- Now assume that $m_1 = b \simeq a$ where $a \prec b$ and that there exists a pair $(l, T') \in T_3$, i.e. where $l \in C$ such that $\text{isEntailed}_{i0}(C \setminus \{l\}, T')$ is true. By induction there exists a $D \in T'$ such that $D \leq_{\text{I}} C \setminus \{l\}$. Hence $l \vee D \leq_{\text{I}} C$, so the property is verified.

Suppose that there exists a clause $D \in \mathcal{C}(T)$ such that $D \leq_{\text{I}} C$, we prove by induction on size(T) that $\text{isEntailed}_{i0}(C, T) = \top$. If $T = \square$ then the result is clear; otherwise, D is of the form $l \vee D'$, for some $(l, T') \in T$ and $D' \in \mathcal{C}(T')$. Let $m_1 = \min_{<_\pi} \{m \in C\}$, so that $C = m_1 \vee C'$. Note that necessarily, $m_1 \leq_\pi l$. Indeed, assume this is not the case. If m_1 is of the form $b \not\succ a$ where $a \prec b$, then l must be of the form $u \not\succ v$, where $u \succ v$. Since $b \not\succ a$ is minimal in C, necessarily $v_{\lfloor C} = v$ and $u_{\lfloor C} \neq v$ because either $u = b$ and $u_{\lfloor C} = a \succ v$ or $u \prec b$ and $u_{\lfloor C} = u \succ v$. Hence $(u \not\succ v)_{\lfloor C}$ cannot be a contradiction and $D \not\leq_{\text{I}} C$. If m_1 is a positive literal then C must be a positive clause by definition of the ordering $<_\pi$, and by Proposition 1.6(2), $l_{\lfloor C} = l$ cannot belong to C (since all literals in C are $\leq_\pi$-greater than l if $l <_\pi m_1$ holds) and we cannot have $D \leq_{\text{I}} C$.

If m_1 is of the form $b \not\succ a$ with $a \prec b$, then there are two cases to consider. If $l = m_1$, then D is of the form $m_1 \vee D'$ and since T' is a relaxed normal clausal tree, constant a cannot occur in D' and it is straightforward to verify that $D' \leq_{\text{I}} C'$; hence $(l, T') \in T_1$ and the call to isEntailed_{i0} on C' and T' at line 11 returns $\top$. If $m_1 <_\pi l$ then, since $l_{\lfloor C}$ is a contradiction, l cannot be of the form $b \not\succ c$ with $b \succ c$ because $b_{\lfloor C} = a$ and since $b \not\succ a$ is minimal in C, we cannot have $c_{\lfloor C} = a$, thus $(l, T') \in T_2$. By Proposition ii.4, $D[a/b] \leq_{\text{I}} C'$, and since $D[a/b] \in \mathcal{C}(l.T'[a/b])$, the call to isEntailed_{i0} on C' and $l.T'[a/b]$ at line 11 returns $\top$.

If m_1 is a positive literal, then C must be a positive clause and by Proposition 1.6(2), $D_{\lfloor C} = D$. Necessarily $l \in C$ thus $(l, T') \in T_3$ and $D' \subseteq C \setminus \{l\}$. Therefore, the call to isEntailed_{i0} on $C \setminus \{l\}$ and T' returns $\top$. ∎

Theorem 5.9 *The Algorithm* isEntailed$_{i0}$ *terminates in* $\mathcal{O}(\text{size}(\mathcal{C}(T))+|C|\times|\mathcal{C}(T)|)$.

Proof. We assume that the replacement of the constant b by the constant a performed in the recursive call isEntailed$_{i0}(C\setminus\{m_1\}, l.T'[a/b])$ is not carried out by going through the whole tree $l.T'$, but simply taken into account in the following recursive calls (with a constant cost [1]). We can then estimate that in the worst case, we have one recursive call per edge in the tree T, plus one recursive call per literal in the clause C for each branch of T. Moreover, there are at most as many edges in T than there are literals in the clauses of $\mathcal{C}(T)$. Thus, the complexity of isEntailed$_{i0}(C,T)$ is in $\mathcal{O}(\text{size}(\mathcal{C}(T))+|C|\times|\mathcal{C}(T)|)$.■

2 Algorithm isEntailed for e-subsumption and $\mathbb{E}_1$

The algorithm isEntailed$_{e1}$ is the transposition of isEntailed$_{e0}$ for $\mathbb{E}_1$ (and e-subsumption). The handling of branches (l, T') during the exploration of a tree T is roughly unchanged:

- T_1 and T_3 regroup those such that $l \models C$ holds, on which the exploration is pursued on T';
- T_2 contains those of undetermined status relative to C, that need more projections of the literals of C to be analyzed;
- the other branches are not explored further because $l \not\models C$ is obvious.

However, due to the adaptation of the projection technique to $\mathbb{E}_1$, there are several major differences between isEntailed$_{e0}$ and isEntailed$_{e1}$.

- It is necessary to keep the negative literals of C in recursive calls after having used them a first time in a projection. They are stored in a clause M that is empty in the main call and filled along the recursive calls. In isEntailed$_{e0}$, these literals could be discarded after being used, but this is no longer possible because they must also be used in the projections of positive literals (in the constraint of the set T_3 defined line 16), as in Example 5.10. A consequence of this evolution is the simplification of the proof of correctness that does not rely anymore on the notion of 'relaxed' normal form.
- The removal of contradictory literals appearing due to the projections cannot be anticipated as in isEntailed$_{e0}$ (line 9), where these literals are isolated in T_1 and omitted from the subsequent recursive calls even before a rewriting step changes them into contradictory literals. Hence, in the nested calls of isEntailed$_{e1}$, there can be literals that are projected into contradictions just below the root of the tree, and they must be considered before the case where C is empty to preserve the correctness of the algorithm, as illustrated in Example 5.11.

1. Because necessarily $b = b_{|C}$, thus each constant a is rewritten at most only once; and assuming a constant access to each rewriting, stored for example in a hashtable

Algorithm 5 $\textsc{isEntailed}_{e1}(C,\ T,\ M)$

Require: T is a clausal tree in normal form, $M \vee C$ is a clause in normal form.
Ensure: $\textsc{isEntailed}_{e1}(C,\ T,\ M) = \top$ iff $\exists D \in \mathcal{C}(T), D \leq_{\mathrm{E}} M \vee C$

1: **if** $T = \square$ **then**
2: **return** $\top$
3: **end if**
4: $T_1 \leftarrow \{(l, T') \in T \mid l_{\downarrow M}$ is a contradiction$\}$
5: **if** $\bigvee_{(l,T') \in T_1} \textsc{isEntailed}_{e1}(C,\ T',\ M)$ **then**
6: **return** $\top$
7: **end if**
8: **if** $C = \square$ **then**
9: **return** $\bot$
10: **end if**
11: $m_1 \leftarrow \min_{<_\pi} \{m \in C\}$
12: **if** m_1 is of the form $u \not\simeq v$, with $u \succ v$ **then**
13: $T_2 \leftarrow \{(l, T') \in T \mid l_{\downarrow M} \not<_\pi m_1$ and $\nexists w, (l_{\downarrow M} = u \not\simeq w$, with $u \succ w)\}$
14: **return** $\bigvee_{(l,T') \in T_2} \textsc{isEntailed}_{e1}(C \setminus \{m_1\},\ l.T',\ M \vee m_1)$
15: **else**
16: $T_3 \leftarrow \{(l, T') \in T \mid C_{\downarrow M \vee l^c}$ contains a tautological literal$\}$
17: **return** $\bigvee_{(l,T') \in T_3} \textsc{isEntailed}_{e1}(C,\ T',\ M)$
18: **end if**

Example 5.10 We show a sequence of $\textsc{isEntailed}_{e1}$ recursive calls in which the variable M is used in the projection of a positive literal. Let $T = \{(e \simeq b, \square)\}$ and $C = b \not\simeq a \vee f(e) \simeq f(a)$ where $a \prec b \prec e \prec f(a) \prec f(e)$.

1	$\textsc{isEntailed}_{e1}(C, T, \square)$	
2	$\textsc{isEntailed}_{e1}(f(e) \simeq f(a), T, b \not\simeq a)$	(call line 14)
3	$\textsc{isEntailed}_{e1}(f(e) \simeq f(a), \square, b \not\simeq a)$	(call line 17)
4	$\top$	(ret. line 2)

After the main call (1), $T \neq \square$ and $T_1 = \emptyset$ because $e \simeq b_{\downarrow M}$ can never be a contradiction since it is a positive literal, and $C \neq \emptyset$. Thus line 11 is reached and m_1 is set to $b \not\simeq a$. The test at line 12 returns true and T_2 is set to $\{(e \simeq b, \square)\}$, hence the nested call (2) is reached at line 14. This time again $T \neq \square$ and $T_1 = \emptyset$ and $C \neq \square$ thus line 11 is reached again and m_1 is set to $f(e) \simeq f(a)$. Then T_3 is set to $\{(e \simeq b, \square)\}$ because $(f(e) \simeq f(a))_{\downarrow M \vee e \not\simeq b} = f(a) \simeq f(a)$, a tautology. In the final call $T = \square$ thus $\top$ is returned. ♣

Example 5.11 We show a sequence of $\textsc{isEntailed}_{e1}$ recursive calls that illustrates the importance of the lines 4 to 6. Let $T = \{(d \not\simeq c, \{(f(b) \not\simeq f(a), \square)\})\}$

and $C = b \not\approx a \vee d \not\approx c$ where $a \prec b \prec c \prec d \prec f(a) \prec f(b)$.

1	$\text{isEntailed}_{e1}(C, T, \square)$	
2	$\text{isEntailed}_{e1}(d \not\approx c, T, b \not\approx a)$	(call 14)
3	$\text{isEntailed}_{e1}(\square, T, b \not\approx a \vee d \not\approx c)$	(call line 14)
4	$\text{isEntailed}_{e1}(\square, \{(f(b) \not\approx f(a), \square)\}, b \not\approx a \vee d \not\approx c)$	(call line 5)
5	$\text{isEntailed}_{e1}(\square, \square, b \not\approx a \vee d \not\approx c)$	(call line 5)
6	$\top$	(ret. line 2)

In the second call we have $M = b \not\approx a$ thus $(f(b) \not\approx f(a)_{\downarrow M}$ is already a contradiction, but the literal $f(b) \not\approx f(a)$ occurring in T is not accessible at the time, therefore it cannot be removed before $d \not\approx c$. In the third call $M = b \not\approx a \vee d \not\approx c$, hence both $(d \not\approx c)_{\downarrow M}$ and $(f(b) \not\approx f(a)_{\downarrow M}$ are contradictions. Thus the entailment test is successful. Since in this call $C = \square$ and $T \neq \square$, these two contradictions have to be removed without reaching line 8, rendering the presence of the lines 4 to 6 necessary. The problem these lines solve is that it is possible to have several successive literals labeling a branch projected into contradictions, even when $C = \square$. If there could only be one at a time, the method of isEntailed_{e0} (anticipating the removal of such literals by one recursive call) would suffice. In practice both methods can be combined, resulting in a slightly more efficient algorithm. We chose not to develop this possibility further here because it is not interesting from a theoretical point of view. ♣

The termination and correction of isEntailed_{e1} are presented in Theorem 5.14. The two propositions below are steps of the proof of correctness. Proposition 5.12 shows that all the "interesting" branches starting with a negative literal are necessarily in $T_1 \cup T_2$, and for T_2, Proposition 5.13 justifies the restriction imposed on the form of the first literal of the selected branches.

Proposition 5.12 *Let C be a non-empty clause such that $M \vee C$ is in normal form. Let l be a literal such that $l_{\downarrow M} \models M \vee C$. Let $m_1 = \min_{<_\pi}\{m \in C\}$. If l is a negative literal, then either $l_{\downarrow M}$ is a contradiction or $l_{\downarrow M} \not<_\pi m_1$.*

Proof. Let $l_{\downarrow M} = u \not\approx v$. Assuming that $l_{\downarrow M} <_\pi m_1$, since m_1 is minimal in C, $v_{\downarrow M \vee C} = v_{\downarrow M} = v$ and by Theorem 1.27, $u_{\downarrow M \vee C} = v_{\downarrow M \vee C}$.

- If m_1 is of the form $u \not\approx v'$ with $v' \prec u$, then $u_{\downarrow C \vee M} = u_{\downarrow C} = v'$, because $C \vee M$ is in normal form and $m_1 \in C$. But then $v' = v_{\downarrow C}$, thus $v' \preceq v$ and $l_{\downarrow M} \not<_\pi m_1$, contradicting our assumption.
- Otherwise $m_1 = s \not\approx t$ with $s \succ u$ and thus $u_{\downarrow C} = u$ (because the smallest term to be rewritten, namely s, is greater than u) thus $u = v$. ∎

Proposition 5.13 *Let $M \vee C$ be a clause in normal form with M negative. Let $m_1 = \min_{<_\pi}\{m \in C\}$ and assume m_1 is a negative literal $u \not\approx v$ with $u \succ v$. Let l be a literal such that $l_{\downarrow M} \not<_\pi m_1$ and $l_{\downarrow M} \models C \vee M$. In these conditions, the literal $l_{\downarrow M}$ cannot be of the form $u \not\approx w$ with $u \succ w \succ v$*

PROOF. Assume that $l_{\downarrow M} = u \not\simeq w$ with $u \succ w \succ v$. Then because $C \vee M$ is in normal form $u_{\downarrow C \vee M} = v$. In addition, by Theorem 1.27, since $l_{\downarrow M} \models C \vee M$, we know that $w_{\downarrow C \vee M} = u_{\downarrow C \vee M}$, thus $w_{\downarrow C \vee M} = v$. The fact that $w \prec u$ entails $w_{\downarrow C \vee M} = w_{\downarrow M} = w$ (because the maximal terms in the disequations in C are all greater than w, hence w cannot be rewritten by C, cf. Proposition 1.15). Thus we have $w = v$ which contradicts the hypothesis $w \succ v$. ∎

Theorem 5.14 *If T is a clausal tree in normal form, $M \vee C$ is a clause in normal form and M is negative then the call* $\text{isEntailed}_{e1}(C, T, M)$ *terminates and* $\text{isEntailed}_{e1}(C, T, M) = \top$ *iff* $\exists D \in \mathcal{C}(T), D \leq_{\text{E}} M \vee C$.

PROOF. The termination proof is trivial, because for all recursive calls, the positive value $|C| + \text{depth}(T)$ strictly decreases.

The proof of correctness requires two inductions, one for each implication. In the direct direction the proof consists in going through the different cases enumerated by the algorithm to verify that the requirements of the recursive calls are indeed met and that it is possible to derive the desired property in all cases. In the converse direction the different cases that validate the right-hand side of the equivalence are considered and matched with the different cases of the algorithm.

Direct implication: assuming $\text{isEntailed}_{e1}(C, T, M) = \top$, then one of the "**return**" instructions (except that of line 9) has been triggered and returned true. Let us consider each of them in their order of appearance.

1. Line 2, $T = \square$. In this case $\mathcal{C}(T) = \{\square\}$ and the property $\square \leq_{\text{E}} M \vee C$ is verified.
2. Line 5, $T \neq \square$ and $\bigvee_{(l,T') \in T_1} \text{isEntailed}_{e1}(C, T', M)$ returns true, with $T_1 = \{(l, T') \in T \mid l_{\downarrow M} \text{ is a contradiction}\}$. Thus there must exist a pair $(l, T') \in T_1$ such that $\text{isEntailed}_{e1}(C, T', M, N \vee l)$ returns true and $l_{\downarrow M}$ is a contradiction. The preconditions of the corresponding recursive call are trivially met. By induction $\exists D \in \mathcal{C}(T')$ s.t. $D \vee l \vee N \leq_{\text{E}} M \vee C$. Considering that $l \vee D \in \mathcal{C}(T)$, the result is verified in this case.
3. Line 14, $T \neq \square$, $C \neq \square$ and $\bigvee_{(l,T') \in T_2} \text{isEntailed}_{e1}(C \setminus \{m_1\}, l.T', M \vee m_1)$ where T_2 is defined at the previous line and $m_1 = \min_{<_\pi} \{m \in C\}$ is of the form $u \not\simeq v$, with $u \succ v$. Thus there is a pair $(l, T') \in T$ such that $l \neq u \not\simeq w$ with $u \succ w$, and $m_1 \leq_\pi l_{\downarrow M}$ for which $\text{isEntailed}_{e1}(C \setminus \{m_1\}, l.T', M \vee m_1)$ returns true. Since $M \vee C = M \vee m_1 \vee C \setminus m_1$ and $M \vee m_1$ is negative, the pre-conditions of this recursive call are verified. By induction, its returning true entails $\exists D \in \mathcal{C}(l.T')$ such that $D \leq_{\text{E}} M \vee m_1 \vee C \setminus m_1$. Since $\mathcal{C}(l.T') \subseteq \mathcal{C}(T)$, the clause D also belongs to $\mathcal{C}(T)$, whence the result in this case.
4. Line 17, $m_1 = u \simeq v$, $l = s \simeq t$ and $\bigvee_{(l,T') \in T_3} \text{isEntailed}_{e1}(C, T', M)$ returns true, with $T_3 = \{(l, T') \in T \mid C_{\downarrow M \vee l^c} \text{ contains a tautological literal}\}$. In this case there exists a pair $(l, T') \in T$ and $m_2 \in C$ s.t. ${m_2}^c{}_{\downarrow M \vee l^c}$ is a

contradiction thus, by Theorem 1.27, ${l_2}^c \models M \vee l^c$ which is equivalent to $l \models M \vee l_2$. In addition the preconditions of the corresponding recursive call are met. Thus by induction $D \vee l \leq_{\text{E}} M \vee C$ with $D \in T'$. Since $l \vee D \in \mathcal{C}(T)$, we have the desired result for this case.

Converse implication: Assuming there exists a D in $\mathcal{C}(T)$ such that $D \leq_{\text{E}} M \vee C$ (where C, T and M verifying the algorithm's pre-conditions), there are several cases to consider.

— If $T = \square$, then line 2 is reached and $\top$ is returned.

— Otherwise, $D = l \vee D'$ with $(l, T') \in T$ and $D' \in \mathcal{C}(T')$. Some sub-cases must be distinguished.

  — If $C = \square$ then we have $l \vee D' \leq_{\text{E}} M$, thus also $l \vee D' \models M$. Since M is negative, Theorem 1.27 ensures that $l_{\downarrow M}$ is a contradiction thus $(l, T') \in T_1$. Moreover $D' \vee l \leq_{\text{E}} M$ and the pre-conditions of $\text{isEntailed}_{e1}(C, T', M)$, called line 5, are verified. Thus by induction this call returns true, ensuring that $\text{isEntailed}_{e1}(C, T, M)$ also returns true at line 6.

  — If $C = m_1 \vee C'$ with $m_1 = \min_{<_\pi}\{m \in C\}$ and if l is a negative literal, then by Proposition 5.12 either 1) $l_{\downarrow M}$ is a contradiction or 2) $l_{\downarrow M} \not<_\pi m_1$. Indeed, the conditions of application of this proposition are verified, since $l \models M \vee C$ and Proposition 1.3 ensures that $l_{\downarrow M} \models M \vee C$.

    1. If $l_{\downarrow M}$ is a contradiction then $(l, T') \in T_2$, thus the recursive call $\text{isEntailed}_{e1}(C \setminus \{m_1\}, l.T', M \vee m_1)$, line 14, returns true by induction.

    2. If $l_{\downarrow M} \not<_\pi m_1$ then m_1 is a negative literal (since l is negative and by definition of $<_\pi$), and since $l \vee D' \leq_{\text{E}} M \vee C$, we have $l \vee D' \leq_{\text{E}} M \vee m_1 \vee C \setminus \{m_1\}$. Here, by Proposition 5.13, $(l, T') \in T_2$ thus the recursive call $\text{isEntailed}_{e1}(C \setminus m_1, l.T', M \vee m_1)$ reached line 14 returns true by induction (the preconditions are verified and $D \leq_{\text{E}} M \vee C = M \vee m_1 \vee C \setminus \{m_1\}$).

  — The last case is when $C = m_1 \vee C'$ with $m_1 = \min_{<_\pi}\{m \in C\}$ and $l = s \simeq t$ is a positive literal. If m_1 is a negative literal then $(l, T') \in T_2$ and line 14 is reached as in the previous case. It returns true for the same reason. Otherwise by Theorem 1.27 there exists a positive literal l_2 in C such that $l_{2\downarrow M \vee C \vee s \not\simeq t}$ is a tautology, thus $(l, T') \in T_3$. Since the preconditions of the call $\text{isEntailed}_{e1}(C, T', M)$ reached line 17 are verified, by induction it returns true. ∎

Remark 5.15 *Using the same reasoning as for* isEntailed_{i0}*, it is possible to estimate the complexity upper-bound of* isEntailed_{e1} *to* $\mathcal{O}(\text{size}(\mathcal{C}(T)) + |C^-| \times |\mathcal{C}(T)|)$*. Although this bound is slightly better than the one for* isEntailed_{i0} *because* $|C|$ *is replaced by* $|C^-|$ *in the upper-bound, in practice* isEntailed_{e1} *is the less efficient algorithm because all the comparisons and projection operations*

are more costly (due to the possible involvement of nested terms in these operations). Instead of assuming these operations can be performed in constant time (which was fully legitimate in $\mathbb{E}_0$), if we add as a parameter d the maximal term depth encountered during the execution of isEntailed$_{e1}$*, a better approximation of the upper-bound complexity of this algorithm is $\mathcal{O}(d \times |C| \times \text{size}(\mathcal{C}(T)) + |C^-| \times |\mathcal{C}(T)|)$, taking the projections and comparisons into account.*

Chapter 6

Pruning of a clausal tree

The second operation on clausal trees necessary to any saturation procedure (see Definition ii.32) is the removal from a tree T of all the clauses $D \in \mathcal{C}(T)$ that are redundant with respect to a given clause C, under the assumption that C itself is not redundant with respect to any of the clauses stored in T. The principle of this algorithm is very similar to that of the isEntailed family of algorithms, it is a depth-first traversal of T while projections of literals are performed. The main difference is that the roles of C and T are reversed (the literals of a branch of T are projected on the literals of C). In addition, when a redundancy is detected instead of returning $\top$ the algorithm cuts the corresponding branch of T and goes on to explore the remaining branches.

1 Algorithm pruneEntailed for i-subsumption and $\mathbb{E}_0$

The algorithm pruneEntailed$_{i0}$ (Algorithm 6) deletes from a tree T all clauses that are i-subsumed by C. We handle every branch (l, T') separately and distinguish several cases according to the form of the minimal literal m_1 in the clause C. If $m_1 = l$ then it is clear that it suffices to call the algorithm recursively on $C \setminus \{m_1\}$ and T'. This is done at line 9. T_1 corresponds to the set of branches of the form (m_1, T') and T_{out1} is the result of the recursive call on those branches. The other branches that are likely to be i-subsumed by C are grouped according to the polarity of l: in T_2 if l is negative and in T_3 if l is positive. The l's from branches in T_2 are used to rewrite C before recursive calls pursue their exploration resulting in T_{out2}, while T_{out3} results from the pursued exploration of the branches stored in T_3. In the output tree, T_{out1}, T_{out2} and T_{out3} respectively replace T_1, T_2 and T_3 while the other branches are left as before. Branches ending with $\emptyset$ are systematically deleted from the output tree. In this algorithm, the specificities of i-subsumption manifest themselves in the way positive literals l such that $(l, T') \in T_1$ are handled. To implement e-subsumption instead, these positive literals must be handled differently from

the negative ones. In PRUNEENTAILED$_{e0}$ we have to replace line 9 by the sequence of instruction ensuring that $T_1 = \{(l, T') \in T_1 \mid l = m_1 \wedge l \text{ is negative}\}$ and $T_{1p} = \{(l, T') \in T_1 \mid l = m_1 \wedge l \text{ is positive}\}$ then compute the set $T_{out1p} = \bigcup_{(l,T') \in T_{1p}} \text{PRUNEENTAILED}_{e0}(C \setminus \{m_1\}, l.T')$ so that the literals l from T_{1p} can be used again in other projections if necessary. The branches remaining in T_1 can be handled as in PRUNEENTAILED$_{i0}$ and line 14 the pair (T_{1p}, T_{out1p}) must be handled as the other (T_i, T_{outi}) pairs.

Algorithm 6 PRUNEENTAILED$_{i0}(C, T)$

Require: C is a clause in relaxed normal form, T is a normal clausal tree and ISENTAILED$_{i0}(C, T) = \bot$.
Ensure: $\forall D \in \mathcal{C}(\text{PRUNEENTAILED}_{i0}(C, T))$, $C \not\leq_{\text{I}} D$.

1: **if** $C = \square$ **then**
2: **return** $\emptyset$
3: **end if**
4: **if** $T = \square$ **then**
5: **return** T
6: **end if**
7: $m_1 \leftarrow \min_{<_\pi} \{m \in C\}$
8: $T_1 \leftarrow \{(l, T') \in T \mid l = m_1\}$
9: $T_{out1} \leftarrow \{(l, \text{PRUNEENTAILED}_{i0}(C \setminus \{m_1\}, T'))|(l, T') \in T_1\}$
10: $T_2 \leftarrow \{(l, T') \in T | l = b \not\simeq a \text{ where } a \prec b \wedge (\nexists c \prec b, m_1 = b \not\simeq c) \wedge l <_\pi m_1\}$
11: $T_{out2} \leftarrow \{(l, \text{PRUNEENTAILED}_{i0}(C[a/b], T'))|(l, T') \in T_2\}$
12: $T_3 \leftarrow \{(l, T') \in T \mid l \text{ is positive } \wedge l <_\pi m_1\}$
13: $T_{out3} \leftarrow \{(l, \text{PRUNEENTAILED}_{i0}(C, T'))|(l, T') \in T_3\}$
14: **return** $T_{out1} \cup T_{out2} \cup T_{out3} \cup (T \setminus (T_1 \cup T_2 \cup T_3))$

As with the previous algorithm, before proving its soundness, we must ensure that the requirements of the algorithm are met by all the recursive calls. Note that PRUNEENTAILED$_{i0}(C, T)$ is necessarily a normal clausal tree. Indeed, it is clear that PRUNEENTAILED$_{i0}$ does not add or modify nodes or labels in T: the only operations performed by the algorithm are replacing subtrees with empty sets and removing elements. Thus, all the conditions in Definition 4.7 are preserved.

Lemma 6.1 *Let C be a clause in relaxed normal form and T a normal clausal tree. All the clauses arguments of a recursive call in* PRUNEENTAILED$_{i0}(C, T)$ *are in relaxed normal form.*

PROOF. Since C is in relaxed normal form, clearly $C \setminus \{m\}$ is also in relaxed normal form for all literals m in C. The only cases that must be detailed are the recursive calls of the form PRUNEENTAILED$_{i0}(C[a/b], T')$ which are invoked for $(l, T') \in T_2$, i.e. where l is of the form $b \not\simeq a$ with $a \prec b$; $m_1 = \min_{<_\pi} \{m \in C\}$ is not of the form $b \not\simeq c$ with $b \succ c$; and $l <_\pi m_1$.

By induction on $|C^-|$, we prove that if C is in relaxed normal form then so is $C[a/b]$. If $|C^-| = 0$ then all the literals in C and $C[a/b]$ are positive. Thus by Proposition 1.6(2), for all $m \in C[a/b]$, $m = m_{\downharpoonright C[a/b]}$, and so $C[a/b]$ is in relaxed normal form. Otherwise, C contains at least one negative literal and by definition of $<_\pi$, m_1 is necessarily negative (of the form $u \not\simeq v$ with $u \succ v$). By definition, $C \setminus \{m_1\}$ is in relaxed normal form and by the induction hypothesis, so is $(C \setminus \{m_1\})[a/b]$. The literal $m_1[a/b]$ (denoted by m'_1 in the rest of the proof) verifies the following properties:

- $m'_1 = u' \not\simeq v'$ **is not a contradiction.** Since $m_1 = u \not\simeq v$ is not a contradiction and $u \succ b$ by hypothesis, $u \neq b$, thus $u' = u$ and $v' \leq v < u$; hence $u' \neq v'$ and m'_1 cannot be a contradiction.
- m'_1 **is unique in** $C[a/b]$**.** For any negative literal $m'_2 \in (C \setminus \{m_1\})[a/b]$, the corresponding literal $m_2 \in C \setminus \{m_1\}$ is of the form $s \not\simeq t$ (where $s \succ t$), with $s \succ u$, so $m'_1 <_\pi m'_2$ (since $u = u'$). Thus, m'_1 is unique in $C[a/b]$.
- $m'_1 = u' \not\simeq v'$ **with** $u'_{\downharpoonright C[a/b]} = v'$**.** Let $D = C[a/b]$ and assume $v'_{\downharpoonright D} = w$, where $w \neq v'$. Since $D[a/b] = D = C[a/b]$, by Proposition ii.4, $D \models a \not\simeq b \vee C$. Since $v' \neq w$, this means that $\neg C \not\models v' \simeq w$ and $\neg C \cup \{a \simeq b\} \models v' \simeq w$. Thus by Proposition 1.7, either $\neg C \models v' \simeq a, w \simeq b$, or $\neg C \models v' \simeq b, w \simeq a$. But since C is in relaxed normal form, this means that C should contain either $b \not\simeq w$ or $a \not\simeq w$, and both cases are impossible since $b \not\simeq a$ is minimal in C.

In addition, since u does not appear in any literal in $C \setminus \{m_1\}$ and since $a \neq u$, for all literals $l_i \in C \setminus \{m_1\}$, $l_{i\downharpoonright C[a/b]} = l_{i\downharpoonright (C\setminus\{m_1\})[a/b]}$. Thus, the properties verified by induction by the literals of $(C \setminus \{m_1\})[a/b]$ are also verified in $C[a/b]$. ∎

Theorem 6.2 *Let C be a clause in relaxed normal form and T be a normal clausal tree. Then* $\text{pruneEntailed}_{i0}(C, T)$ *is a normal clausal tree that contains exactly the clauses $D \in \mathcal{C}(T)$ such that $C \not\leq_\text{I} D$.*

Proof. If D occurs in $\text{pruneEntailed}_{i0}(C, T)$, then it necessarily also occurs in $\mathcal{C}(T)$, because $\text{pruneEntailed}_{i0}(C, T)$ is obtained by removing some branches from T. If $C = \square$, then we have $C \models D$ for every clause D, thus any clause in $\mathcal{C}(T)$ must be removed and in this case, the algorithm ensures that $\text{pruneEntailed}_{i0}(C, T) = \emptyset$. Now assume that $C \neq \square$. We prove that $\mathcal{C}(\text{pruneEntailed}_{i0}(C, T)) = \{D \in \mathcal{C}(T) \mid C \not\leq_\text{I} D\}$ by proving both inclusions.

Let $D \in \mathcal{C}(T)$ such that $C \leq_\text{I} D$. We show $D \notin \mathcal{C}(\text{pruneEntailed}_{i0}(C, T))$ by induction. If $D = \square$ (i.e. $T = \square$), then C must be a contradiction and since C is in relaxed normal form, $C = \square$. From now on, we assume that $D \neq \square$ and $C \neq \square$. Let $m_1 = \min_{<_\pi}\{l_i \in C\}$, $l = \min_{<_\pi}\{m \in D\}$ and $D' = D \setminus \{l\}$. Since T is a normal clausal tree and $D \in \mathcal{C}(T)$, by definition there exists a unique normal clausal tree T' such that $(l, T') \in T$ and $D' \in \mathcal{C}(T')$. There are several cases to consider:

1. $m_1 <_\pi l$, in which case $(l, T') \in T \setminus (T_1 \cup T_2 \cup T_3)$ and no recursive call is done,

2. l is negative and $m_1 = l$, in which case $(l, T') \in T_1$ and the recursive call $\textsc{pruneEntailed}_{i0}(C \setminus \{m_1\}, T')$ is invoked,
3. $l = b \not\succ a$ and $m_1 = b \not\succ c$, where $b \succ a, c$, in which case $(l, T') \in T \setminus (T_1 \cup T_2 \cup T_3)$ and no recursive call is done,
4. $l = b \not\succ a$, $l <_\pi m_1$ and m_1 is not of the form $b \not\succ c$ with $b \succ a, c$, in which case $(l, T') \in T_2$ and $\textsc{pruneEntailed}_{i0}(C[a/b], T')$ is invoked,
5. l is positive and $m_1 = l$, in which case $(l, T') \in T_1$ and the recursive call $\textsc{pruneEntailed}_{i0}(C \setminus \{m_1\}, T')$ is invoked,
6. l is positive and $l <_\pi m_1$, in which case $(l, T') \in T_3$ and the recursive call $\textsc{pruneEntailed}_{i0}(C, T')$ is invoked.

These cover all the possible relations between l and m_1.

1. Assume $m_1 <_\pi l$. We distinguish two cases depending on the polarity of m_1:
   — If $m_1 = c \not\succ d$ with $c \succ d$, then $m_{1\downarrow D}$ is a contradiction by Theorem 1.21, so $c_{\downarrow D} = d_{\downarrow D}$. But for all $l' \in D$, $m_1 <_\pi l'$, and $d_{\downarrow D} = d$ since D which is in relaxed normal form cannot contain a literal $d \not\succ d_{\downarrow D}$ which would be smaller than m_1. Therefore $c_{\downarrow D} = d$ and by definition of a clause in normal form, $m_1 \in D$.
   — If m_1 is positive, then $m_{1\downarrow D} \in D_{\downarrow D}$ by Theorem 1.21, and because D is in normal form, $m_{1\downarrow D} \in D$. But since $m_1 <_\pi l'$ for all $l' \in D$, D must only contain positive literals. Hence by Proposition 1.6(2), $m_{1\downarrow D} = m_1$, and $m_1 \in D$.
   Thus, in both cases, $m_1 \in D$, which is impossible since for all $l' \in D$, $m_1 < l \leq l'$.
2. Assume $l = b \not\succ a$ with $a \prec b$ and $m_1 = l$. In this case, the recursive call $\textsc{pruneEntailed}_{i0}(C \setminus \{m_1\}, T')$ is invoked. Since $C \leq_{\text{I}} D$, for any literal $m \in C$ such that $m \neq m_1$:
   — If m is negative, then $m_{\downarrow D}$ is a contradiction. By definition of a clause in relaxed normal form, the constant a cannot appear in any literal other than m_1 in C, hence $C \setminus \{m_1\}_{\downarrow D} = C \setminus \{m_1\}_{\downarrow D'}$. Thus $m_{\downarrow D'}$ is also a contradiction.
   — If m is positive then $m_{\downarrow D} \in D_{\downarrow D}$. But by definition of a normal clausal tree, the positive literals of $D_{\downarrow D}$ are the same as those of D, D' and ${D'}_{\downarrow D'}$. Hence $m_{\downarrow D'} \in {D'}_{\downarrow D'}$.
   This means that $C \setminus \{m_1\} \leq_{\text{I}} D'$ and $D' \notin \mathcal{C}(\textsc{pruneEntailed}_{i0}(C, T'))$ by the induction hypothesis, thus $D \notin \mathcal{C}(\textsc{pruneEntailed}_{i0}(C, T))$.
3. If $l = b \not\succ a$ and $m_1 = b \not\succ c$, where $b \succ a, c$, then $m_{1\downarrow D} = a \not\succ c$ is not a contradiction. Thus by Theorem 1.21, $C \not\leq_{\text{I}} D$, which contradicts our hypothesis.
4. If $l = b \not\succ a$ where $a \prec b$, $l <_\pi m_1$ and m_1 is not of the form $b \not\succ c$ with $b \succ c$, then $\textsc{pruneEntailed}_{i0}(C[a/b], T')$ is invoked. By Proposition ii.4, $C[a/b] \leq_{\text{I}} D[a/b]$, hence $C[a/b] \leq_{\text{I}} D'$. By induction $D' \notin \mathcal{C}(\textsc{pruneEntailed}_{i0}(C, T'))$, hence $D \notin \mathcal{C}(\textsc{pruneEntailed}_{i0}(C, T))$.

5. Assume l is positive and $m_1 = l$. In this case, both C and D contain only positive literals, thus for any $m \in C$ such that $m \neq m_1$, by Proposition 1.6(2), $m_{\lfloor D} = m_{\lfloor D'} = m$ and $D_{\lfloor D} = D$. Furthermore, $m \in D_{\lfloor D}$, hence $m \in D'$, and $C \setminus \{m_1\} \leq_{\text{I}} D'$, so by induction $D' \notin \mathcal{C}(\text{PRUNEENTAILED}_{i0}(C \setminus \{m_1\}, T'))$ and $D \notin \mathcal{C}(\text{PRUNEENTAILED}_{i0}(C, T))$.
6. If l is positive and $l <_\pi m_1$, then the same reasoning as for the previous point holds for any $m \in C$, including m_1, thus $C \leq_{\text{I}} D'$ and $D' \notin \mathcal{C}(\text{PRUNEENTAILED}_{i0}(C, T'))$ by induction.

Now assume that $D \in \mathcal{C}(T)$ is such that $C \not\leq_{\text{I}} D$ (this necessarily entails that $C \neq \square$). We show by induction that $D \in \mathcal{C}(\text{PRUNEENTAILED}_{i0}(C, T))$. If $D = \square$, then $T = \square$ and $\text{PRUNEENTAILED}_{i0}(C, T) = T$, proving the result. Otherwise, as before we write $m_1 = \min_{<_\pi}\{m \in C\}$, $l = \min_{<_\pi}\{l' \in D\}$, $D' = D \setminus \{l\}$ and we consider the unique couple $(l, T') \in T$. According to the definition of I-subsumption, there are two reasons that can explain why $C \not\leq_{\text{I}} D$, denoted respectively by (r_1) and (r_2). The first one (r_1) is when $C \not\models D$, i.e. $C \not\leq_{\text{E}} D$, meaning that there exists a literal $m \in C$ that cannot be projected on D. The second one (r_2) is when $C \models D$ but the mapping from C^+ to D^+ is not injective. We consider the same cases as before:

1. If $m_1 <_\pi l$ then $(l, T') \in T \setminus (T_1 \cup T_2 \cup T_3)$ and no recursive call is done on T'. Moreover $T' \neq \emptyset$ because $D' \in \mathcal{C}(T')$, and this implies that $D \in \mathcal{C}(\text{PRUNEENTAILED}_{i0}(C, T))$.
2. Assume $m_1 = l$ where $l = b \not\succ a$ and $a \prec b$, so that $(l, T') \in T_1$. If our hypothesis is due to (r_1) then $m \neq m_1$ since $m_1 \in D$, thus $m \in C \setminus \{m_1\}$ and m also cannot be projected on D', because $D'_{\lfloor D'} = (D_{\lfloor D}) \setminus \{l_{\lfloor D}\}$. In the case (r_2) the hypothesis originates from the positive literals of C and D and thus, the same problem occurs between $C \setminus \{m_1\}$ and D'. In both cases, $C \setminus \{m_1\} \not\leq_{\text{I}} D'$ and by the induction hypothesis $D' \in \mathcal{C}(\text{PRUNEENTAILED}_{i0}(C, T'))$. By definition, $D \in \mathcal{C}(T_{out1}) \subseteq \mathcal{C}(\text{PRUNEENTAILED}_{i0}(C, T))$.
3. If $l = b \not\succ a$ and $m_1 = b \not\succ c$, with $b \succ a, c$, then $(l, T') \in T \setminus (T_1 \cup T_2 \cup T_3)$ and as seen above, $C \not\leq_{\text{I}} D$. No recursive call is done, so $\text{PRUNEENTAILED}_{i0}(C, T') = T'$, and $D \in \mathcal{C}(\text{PRUNEENTAILED}_{i0}(C, T))$.
4. Assume $l = b \not\succ a$, where $a \prec b$, $l <_\pi m_1$ and m_1 is not of the form $b \not\succ c$ where $b \succ c$, i.e. $(l, T') \in T_2$. If (r_1) then by Proposition ii.4, $C \not\models b \not\succ a \vee D'$ thus $C[a/b] \not\models D'$. Hence $C[a/b] \not\leq_{\text{E}} D'$. Otherwise our hypothesis originates from (r_2). Let m_{k1} and m_{k2} be two literals in C^+ that are mapped to the same literal l_k in D^+. In this case $m_{k1} \equiv_D m_{k2}$, hence by Proposition 2.42 $m_{k1}[a/b] \equiv_{D'} m_{k2}[a/b]$. This means that the same mapping problem of type (r_2) occurs between $C[a/b]$ and D', ensuring that $C[a/b] \not\leq_{\text{I}} D'$. Hence in both cases, by the induction hypothesis $D' \in \mathcal{C}(\text{PRUNEENTAILED}_{i0}(C, T'))$ and so $D \in \mathcal{C}(T_{out2}) \subseteq \mathcal{C}(\text{PRUNEENTAILED}_{i0}(C, T))$.
5. Assume l is positive and $m_1 = l$ so that $(l, T') \in T_1$. For (r_1) as in Point 2, $m \neq m_1$, hence $m \in C \setminus \{m_1\}$. Furthermore, all the literals in D are

positive, thus $D'_{\lfloor D'} = D \setminus \{l\}$. This implies that m cannot be projected on D' and by Theorem 1.21, $C \setminus \{m_1\} \not\leq_{\text{I}} D'$. For (r_2) the situation is the same as in Point 2 with just one additional possibility: let m_2 be a literal in $C \setminus \{m_1\}$ such that $m_1 \equiv_D m_2$, i.e. m_1 and m_2 are both mapped to l in D. In this case, assuming that $C \setminus \{m_1\} \leq_{\text{I}} D'$, we could extend the injective mapping existing between the literals of $C \setminus \{m_1\}$ and those of D' into an injective mapping of the literals of C to those of D simply by associating m_1 to l (because m_2 cannot be mapped to $l \notin D'$), ensuring $C \leq_{\text{I}} D$, a contradiction. Thus in this case as in the others $C \setminus \{m_1\} \not\leq_{\text{I}} D'$ and by the induction hypothesis $D' \in \mathcal{C}(\text{PRUNEENTAILED}_{i0}(C, T'))$ and so $D \in \mathcal{C}(T_{out1}) \subseteq \mathcal{C}(\text{PRUNEENTAILED}_{i0}(C, T))$.

6. If l is positive and $l <_\pi m_1$ then $(l, T') \in T_3$. In this case, wether the main hypothesis originates from (r_1) or (r_2), it is clear that l has no relation to the problem: since D contains only positive literals, the projection of D on any literal is the identity, thus no mapping of the literals of C to those of D, injective or not, involves l, i.e. C is mapped to D'. For this reason $C \not\leq_{\text{I}} D'$. By the induction hypothesis $D' \in \mathcal{C}(\text{PRUNEENTAILED}_{i0}(C, T'))$ hence $D \in \mathcal{C}(T_{out3}) \subseteq \mathcal{C}(\text{PRUNEENTAILED}_{i0}(C, T))$ ∎

Theorem 6.3 *The procedure* PRUNEENTAILED$_{i0}$ *terminates in* $\mathcal{O}(\text{size}(\mathcal{C}(T)))$.

The two algorithms ISENTAILED$_{i0}$ and PRUNEENTAILED$_{i0}$ have a similar structure in terms of recursive calls, hence they also have a similar complexity. However, even in the worst case the recursive calls to PRUNEENTAILED$_{i0}$ always reduce the tree, which is not the case in ISENTAILED$_{i0}$. Thus these recursive calls are not influenced by the number of literals in C, which ensure a slightly better theoretical complexity for PRUNEENTAILED$_{i0}$ than for ISENTAILED$_{i0}$: $\mathcal{O}(\mathcal{C}(T))$ in the worst case.

2 Algorithm PRUNEENTAILED for E-subsumption and $\mathbb{E}_1$

The algorithm PRUNEENTAILED$_{e1}$ is the adaptation of PRUNEENTAILED$_{e0}$ to $\mathbb{E}_1$ and E-subsumption. It presents exactly the same differences with its parent that ISENTAILED$_{e1}$ does with ISENTAILED$_{e0}$ with an inversion of the roles of the clause and the tree.

— The clause N stores the already used negative literals of the branch of T that is being explored.

— The detection of contradictory literals occurs one recursive call later than in PRUNEENTAILED$_{e0}$ and thus needs to be done before the case $T = \square$.

In this algorithm, the cases distinguished also depend on the form of the minimum literal m_1 in C. If $m_{1\lfloor N}$ is a contradiction then m_1 is removed from C and the exploration restarts on all the branches. Among the branches (l, T') where l is negative, the ones that can possibly be E-subsumed by C are grouped in T_1

and the exploration advances to T' on these branches. Of these, only the ones not entailed by C are recovered in T_{out1}. If m_1 is negative and line 18 is reached, the other branches (where l is positive) can be left untouched because the projection $m_{1\downarrow N}$ is not a contradiction thus it is clear that for $D \in \mathcal{C}(\{(l, T')\})$, $C \not\leq_{\mathrm{E}} D \vee N$. On the contrary, if m_1 is positive, these branches have to be explored further and thus are grouped in T_2. Only the ones verifying the property are recovered in T_{out2}. As in the algorithm PRUNEENTAILED$_{i0}$, the branches ending with $\emptyset$ are systematically deleted from the output tree. An interesting feature of PRUNEENTAILED$_{e1}$ is that several positive literals of C can be deleted at once in each recursive call that define T_{out2}, as illustrated in Example 6.4.

Algorithm 7 PRUNEENTAILED$_{e1}(C,\ T,\ N)$

Require: T is a clausal-tree in normal form, C and N are clauses in normal form.
Ensure: $\mathcal{C}(T_{out}) = \{D \in \mathcal{C}(T) \mid C \not\leq_{\mathrm{E}} D \vee N\}$, where $T_{out} =$ PRUNEENTAILED$_{e1}(C,\ T,\ N)$.

1: **if** $C = \square$ **then**
2: **return** $\emptyset$
3: **end if**
4: select $m_1 \in C$ s.t. $m_{1\downarrow N} = \min_{<_\pi} \{m_{\downarrow N} \mid m \in C\}$
5: **if** $m_{1\downarrow N}$ is a contradiction **then**
6: **return** PRUNEENTAILED$_{e1}(C \setminus \{m_1\},\ T,\ N)$
7: **end if**
8: **if** $T = \square$ **then**
9: **return** T
10: **end if**
11: $T_1 \leftarrow \{(l, T') \in T \mid l = u \not\simeq v \wedge l \leq_\pi m_{1\downarrow N}\}$
12: $T_{out1} \leftarrow \{(l,$ PRUNEENTAILED$_{e1}(C,\ T',\ N \vee l) \mid (l, T') \in T_1\}$
13: **if** m_1 is positive **then**
14: $T_2 \leftarrow T \setminus T_1$
15: $T_{out2} \leftarrow \{(l,$ PRUNEENTAILED$_{e1}(C \setminus L_l,\ T',\ N)) \mid (l, T') \in T_2 \wedge L_l = \{m \in C \mid l_{\downarrow N \vee m^c}$ is tautological$\}\}$
16: **return** $T_{out1} \cup T_{out2}$
17: **else**
18: **return** $T_{out1} \cup (T \setminus T_1)$
19: **end if**

Example 6.4 The sequence of recursive calls of PRUNEENTAILED$_{e1}$ presented below illustrates the fact that several positive literals of C can be projected and deleted from subsequent calls at the same time. Let $C = c \simeq b \vee f(c) \simeq f(a)$

and $T = \{(b \not\simeq a, \{(f(c) \simeq f(a), \square)\})\}$ where $a \prec b \prec c \prec f(a) \prec f(c)$.

1	$\text{PRUNEENTAILED}_{e1}(C, T, \square)$	
2	$\text{PRUNEENTAILED}_{e1}(C, \{(f(c) \simeq f(a), \square)\}, b \not\simeq a)$	(called at line 12)
3	$\text{PRUNEENTAILED}_{e1}(\square, \square, b \not\simeq a)$	(called at line 15)
4	$\emptyset$	(returned at line 2)

In the main call (1), $C \neq \square$ and at line 4 m_1 is set to $c \simeq b$. Since $m_{1\downarrow N}$ cannot be a contradiction (it is positive) and $T \neq \square$, line 11 is reached and T_1 is set to $\{(b \not\simeq a, \{(f(c) \simeq f(a), \square)\})\}$, triggering the nested call (2) at line 12. In this call again $C \neq \square$, m_1 is positive and $T \neq \square$, but this time $T_1 = \emptyset$ and at line 14 T_2 is set to $\{(f(c) \simeq f(a), \square)\}$. For the third nested call, the set $L_{f(c)\simeq f(a)}$ defined line 15 is computed. This set contains both literals of C, because $f(c) \simeq f(a)_{\downarrow b\not\simeq a \vee c \not\simeq b} = f(a) \simeq f(a)$ and $f(c) \simeq f(a)_{\downarrow b\not\simeq a \vee f(c)\not\simeq f(b)} = f(a) \simeq f(a)$ are both tautologies. Finally $\emptyset$ is returned because $C = \square$. ♣

Theorem 6.5 *Let C and N be clauses in normal form and T be a clausal tree in normal form verifying the preconditions of* $\text{PRUNEENTAILED}_{e1}$. *Then the call* $\text{PRUNEENTAILED}_{e1}(C, T, N)$ *always terminates. Moreover if we write* $T_{out} = \text{PRUNEENTAILED}_{e1}(C, T, N)$ *then* $\mathcal{C}(T_{out}) = \{D \in \mathcal{C}(T) \mid C \not\leq_{\text{E}} D \vee N\}$.

Proof. The termination of this algorithm is ensured by the following argument: for all recursive calls, the value of $|C| + \text{depth}(T)$ strictly decreases.

The principle of the proof of correctness of $\text{PRUNEENTAILED}_{e1}$ is identical to that of Theorem 5.14. Let D be a clause in $\mathcal{C}(T_{out})$.

- If $C = \square$ then line 2 was reached to generate T_{out}. Then $T_{out} = \emptyset$, whence the result in this case.
- Otherwise, C is of the form $m_1 \vee C'$, where m_1 is such that $m_{1\downarrow N} = \min_{<_\pi} \{m_{\downarrow N} \mid m \in C\}$.
  - If $m_{1\downarrow N}$ is a contradiction then T_{out} is returned at line 6. By induction $C \setminus \{m_1\} \not\leq_{\text{E}} D \vee N$ thus $C \not\leq_{\text{E}} D \vee N$.
  - Otherwise if $T = \square$ then $T_{out} = T$ and $m_1 \not\models N$ by Theorem 1.27, hence $C \not\leq_{\text{E}} N$ (and $D = \square$).
  - Else $m_{1\downarrow N}$ is not a contradiction and $T \neq \square$ thus $D = l \vee D'$, where $(l, T'_{out}) \in T_{out}$ and $D' \in \mathcal{C}(T'_{out})$ such that one of the following holds:
    1. $(l, T'_{out}) \in T_{out1}$, in which case $l = u \not\simeq v$, $m_{1\downarrow N} \succeq l$ and $T'_{out} = \text{PRUNEENTAILED}_{e1}(C, T', N \vee l)$;
    2. l and m_1 are positive literals and $(l, T'_{out}) \in T_{out2}$ in which case $T'_{out} = \text{PRUNEENTAILED}_{e1}(C \setminus L_l, T', N)$ with $(l, T') \in T$ and $L_l \leftarrow \{m \in C \mid l_{\downarrow N \vee m^c} \text{ is tautological}\}$;
    3. m_1 is negative and $m_{1\downarrow N} \prec l$ in which case $T'_{out} = T'$ with $(l, T') \in T \setminus T_1$.

    In all cases $m_1 \not\models N$ by Theorem 1.27. In the first case, the preconditions of the recursive call of line 12 are respected, therefore $C \not\leq_{\text{E}} D' \vee l \vee N$. In the second case, the preconditions of the recursive call line 15 (reached because m_1 is positive) are verified and

by induction $C \setminus L_l \not\preceq_{\text{E}} D' \vee N$ thus $C \not\preceq_{\text{E}} D' \vee l \vee N$. Finally in the last case, by Proposition 1.15 and Theorem 1.27, $m_1 \not\models D$ thus $C \not\models N \vee D$.

For the second inclusion, let D be a clause in $\mathcal{C}(T)$ such that $C \not\preceq_{\text{E}} D \vee N$.

— If $C = \square$, then C entails all clauses thus T_{out} must be empty, which is the case at line 2.

— Otherwise C is of the form $m_1 \vee C'$ where m_1 is such that $m_{1\downarrow N} = \min_{<_\pi} \{m_{\downarrow N} \mid m \in C\}$.

— If $m_{1\downarrow N}$ is a contradiction then $m_1 \models N$ by Theorem 1.27 and T_{out} is returned at line 6. Since $C \not\preceq_{\text{E}} D \vee N$, $C \setminus \{m_1\} \not\preceq_{\text{E}} D \vee N$, thus by induction $D \in \mathcal{C}(T_{out})$.

— If $m_{1\downarrow N}$ is not a contradiction then either $T = \square$, in which case $T_{out} = T$ and the result is straightforward, or the same three cases as in the other direction of the proof can be studied separately (with $D = l \vee D'$, $(l, T') \in T$ and $D' \in \mathcal{C}(T')$).

1. If l is negative and $m_{1\downarrow N} \succeq l$, then $(l, T') \in T_1$. Set $T'_{out1} = \text{pruneEntailed}_{e1}(C, T', N \vee l)$, as in line 12. By induction $D' \in \mathcal{C}(T'_{out1})$ thus $D \in \mathcal{C}(T_{out1})$.
2. If l and m_1 are both positive literals then the execution path goes through line 15. The clause $L_l = \{m \in C \mid l_{\downarrow N \vee m^c} \text{ is tautological}\}$ is such that $L_l \models N$ by Theorem 1.27. The preconditions of the recursive call $\text{pruneEntailed}_{e1}(C \setminus L_l, T', N) = T'_{out2}$ are respected and $C \setminus L_l \not\preceq_{\text{E}} D' \vee N$ since $C \not\preceq_{\text{E}} D \vee N$, thus by induction $D' \in \mathcal{C}(T'_{out2})$ and $D \in \mathcal{C}(T_{out2})$.
3. Finally, if m_1 is negative and $m_{1\downarrow N} \prec l$ then $(l, T') \in T \setminus T_1$ and line 18 is triggered, thus $D \in \mathcal{C}(T_{out})$. ∎

Chapter 7

Constrained clausal tree manipulations

The algorithms considered in the last two chapters can be adapted to c-clauses. This process is detailed in the current chapter for $\mathbb{E}_1$ only. The principle is exactly the same in $\mathbb{E}_0$ thus we do not present this case here. The redundancy detection associated to the c$\mathcal{SP}$ calculus (the one that manipulates c-clauses) is C-subsumption, an extension of E-subsumption to c-clauses (see Definition 3.10) that involves three comparisons. The test $[C \,|\, \mathcal{X}] \leq_{\text{C}} [D \,|\, \mathcal{Y}]$ returns true iff the following holds:

1. $C \leq_{\text{E}} D$,
2. $C \preceq D$,
3. $\mathcal{X} \subseteq \mathcal{Y}$.

Thus testing the C-subsumption of c-clauses in a c-tree (clausal tree for c-clauses, defined Section 1 of this chapter) can begin by using one of the isEntailed algorithms presented in the previous chapters and when it returns true the second and third tests are checked. Similarly, the action of pruning a c-tree according to a c-clause can begin with a pruneEntailed algorithm. Note that in both cases (testing and pruning) although they compare the same clause, the two first steps cannot be merged (at least in our setting) because the clauses stored in c-trees have their literals sorted according to the ordering $<_\pi$ and not $\prec$. Thus, it is necessary to have recovered all the literals of such clauses before performing the second comparison. This means also that these literals cannot be discarded while going through a c-tree or a c-clause during the algorithms execution. Since in c-trees the constraints are stored below the corresponding clauses, the third step has to be performed last. These observations are applied to isEntailed and pruneEntailed respectively in the two sections of this chapter following the introduction of c-trees just below.

1 Constrained clausal trees

The storage of constrained clauses in $\mathbb{E}_1$ and $\mathbb{E}_0$ is similar to that of standard clauses (see Chapter 4). A main normal clausal tree is used to store the clausal part of constrained clauses and at each leaf of this tree, another tree is appended to store the different constraints associated to the same clause.

Definition 7.1 A *constraint tree* is defined inductively in the same way as a clausal tree, but it represents constraints instead of clauses. The set of constraints represented by a constraint tree T is denoted $\mathcal{C}'_c(T)$ and is inductively defined by:

$$\mathcal{C}'_c(T) = \begin{cases} \{\top\} & \text{if } T = \square, \\ \bigcup\limits_{(l,T')\in T} \left(\bigcup\limits_{\mathcal{X}\in\mathcal{C}'_c(T')} \mathcal{X} \wedge l \right) & \text{otherwise.} \end{cases}$$

◊

Definition 7.2 A *constrained clausal tree* or *c-tree* is inductively defined as either a set of pairs of the form (l, T') where T' is a c-tree, or the concatenation of a single pair $(\boxdot, T')$ where T' is a constraint tree with a (possibly empty) set of pairs of the form (l, T') where T' is a c-tree. In addition, a c-tree T has to respect the following conditions:

- the label $\boxdot$ occurs exactly once in any path from the root of T to a leaf,
- for every $(l, T') \in T$,
  - for all l' appearing in T', $l <_\pi l'$,
  - there is no clausal tree T'' such that $T'' \neq T'$ and $(l, T'') \in T$,

The set of clauses represented by a c-tree T is denoted by $\mathcal{C}_c(T)$ and defined inductively as follows:

$$\mathcal{C}_c(T) = \begin{cases} \emptyset & \text{if } T = \emptyset \\ \{[\square \,|\, \mathcal{X}] \,|\, \mathcal{X} \in \mathcal{C}'_c(T')\} & \text{if } T = \{(\boxdot, T')\}, \\ \{[l \vee C \,|\, \mathcal{X}] \,|\, [C \,|\, \mathcal{X}] \in \mathcal{C}_c(T')\} & \text{if } T = \{(l, T')\}, \\ \mathcal{C}_c(T_1) \cup \mathcal{C}_c(T_2) & \text{if } T = T_1 \cup T_2,\ T_1 \neq \emptyset \text{ and } T_2 \neq \emptyset. \end{cases}$$

◊

Example 7.3 The structure T in Figure 11 is a constrained clausal tree in $\mathbb{E}_1$ with the term order $a \prec b \prec c \prec g(c) \prec g(e) \prec f(c) \prec f(d)$. Once more the labels are associated with the nodes rather than with the edges leading to them.

Remark 7.4 *As mentioned in the introduction, the order in which the tests are performed is linked to the c-tree data structure: the tests on the clausal part are performed first and only then the test on the constraints (stored in constraint trees at the leaves of the c-tree) are performed. It is possible to reverse the storage order of c-clauses (redefining c-trees by appending clausal trees at the leaves of*

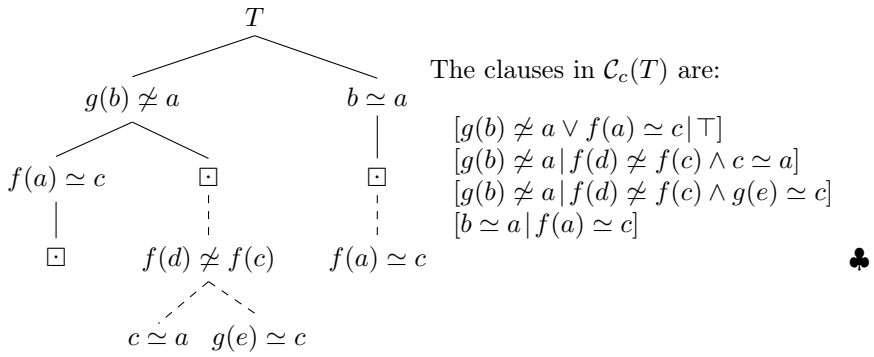


Figure 11 – A constrained clausal tree in $\mathbb{E}_1$

constraint trees) with the effect that the third test has to be performed before the two other tests.

Whether or not this modification generates a gain in efficiency has not been experimentally tested and remains as future work. Nevertheless, the reason behind our choice for the c-tree data structure is twofold. First, given the form of the cSP rules, we intuitively expect a greater sharing of literals in the clausal parts of the generated c-clauses than in the constraints. Second, once the saturation process is completed, this structure renders the recovery of the prime implicates easier. They are all stored in the sub-tree $(\boxdot, T')$ from the root because they have an empty clausal part. With the converse ordering, it would be necessary to go through all the constraints to test whether the appended clausal tree is empty.

2 Algorithm isEntailed for c-trees

As explained in the introduction, the main modification necessary to turn the algorithm isEntailed$_{e1}$ into isEntailed$_{cons}$ (Algorithm 8) for c-trees and c-clauses, is to launch the steps two and three of c-subsumption as soon as the e-subsumption of the clausal part of the c-clauses is confirmed, i.e. modify line 1 of isEntailed$_{e1}$, replacing it as in isEntailed$_{cons}$. The rest of Algorithm 8 contains only the minor adaptations described in the introduction (especially the storage of "used" literals in M for C and in N for T through the recursive calls), thus we focus solely on the changes occurring in this first line. This part of the algorithm addresses the case where the clausal part of the tree is empty (or rather where the clausal part of a branch of the tree has already been explored and is currently stored in the variable N). The two subsequent tests composing c-subsumption are added at this line. Note that the $\preceq$-test is reduced to a simple comparison between standard clauses (since the corresponding tree branch has already been explored), hence we do not detail the corresponding algorithm.

In contrast, the $\subseteq$-test requires going through the constraint tree associated to the clausal branch that has just been explored. This test (detailed in Algorithm

Algorithm 8 isEntailed$_{cons}([C \,|\, \mathcal{X}],\ T,\ M,\ N)$

Require: T is a c-tree in normal form, $M \vee C$ and N are clauses in normal form such that $N \models M \vee C$.

Ensure: isEntailed$_{cons}([C \,|\, \mathcal{X}],\ T,\ M,\ N) = \top$ iff $\exists [D \,|\, \mathcal{Y}] \in \mathcal{C}_c(T), [D \vee N \,|\, \mathcal{Y}] \leq_{\mathrm{C}} [M \vee C \,|\, \mathcal{X}]$.

1: **if** $(\boxdot, T') \in T \wedge (N \preceq M \vee C) \wedge$ isIncluded$(\mathcal{X}, T')$ **then**
2: **return** $\top$
3: **end if**
4: $T_1 \leftarrow \{(l, T') \in T \,|\, l_{\downarrow M}$ is a contradiction$\}$
5: **if** $\bigvee_{(l,T') \in T_1}$ isEntailed$_{cons}([C \,|\, \mathcal{X}],\ T',\ M,\ N \vee l)$ **then**
6: **return** $\top$
7: **end if**
8: **if** $C = \square$ **then**
9: **return** $\bot$
10: **end if**
11: $m_1 \leftarrow \min_{<_\pi} \{m \in C\}$
12: **if** m_1 is of the form $u \not\succeq v$, with $u \succ v$ **then**
13: $T_2 \leftarrow \{(l, T') \in T \,|\, l_{\downarrow M} \not<_\pi m_1$ and $\nexists w, (l_{\downarrow M} = u \not\succeq w$, with $u \succ w)\}$
14: **return** $\bigvee_{(l,T') \in T_2}$ isEntailed$_{cons}([C \setminus \{m_1\} \,|\, \mathcal{X}],\ l.T',\ M \vee m_1,\ N)$
15: **else**
16: $T_3 \leftarrow \{(l, T') \in T \,|\, C_{\downarrow M \vee l^c}$ contains a tautological literal$\}$
17: **return** $\bigvee_{(l,T') \in T_3}$ isEntailed$_{cons}([C \,|\, \mathcal{X}],\ T',\ M,\ N \vee l)$
18: **end if**

9: isIncluded) is a lot simpler than isEntailed$_{cons}$ but its principle is the same, a depth-first traversal of the tree with recursive calls deleting the literals one after the other until the inclusion becomes obvious (or obviously false). In isIncluded the branches $(l, T') \in T$ are grouped and dealt with in accordance with their relation to $m_1 = \min_{<_\pi} \{m \in \mathcal{X}\}$. If $l = m_1$, only the inclusion of a branch of T' in $\mathcal{X} \setminus \{m_1\}$ is tested (line 9). Otherwise if $l <_\pi m_1$ clearly no branch of $l.T'$ is included in $\mathcal{X}$ but if $l <_\pi m_1$ it is possible that one of these branches is included in $\mathcal{X} \setminus \{m_1\}$, which is tested line 13.

The correctness of isEntailed$_{cons}$ is stated in Theorem 7.5. The corresponding proof is not given due to its similarity with the proof of Theorem 5.14 stating the correctness of isEntailed$_{e1}$.

Theorem 7.5 *If T is a c-tree in normal form, $M \vee C$ and N are clauses in normal form then the call* isEntailed$_{cons}([C \,|\, \mathcal{X}],\ T,\ M,\ N)$ *terminates and* isEntailed$_{cons}([C \,|\, \mathcal{X}],\ T,\ M,\ N) = \top$ *iff* $\exists [D \,|\, \mathcal{Y}] \in \mathcal{C}_c(T),\ [D \vee N \,|\, \mathcal{Y}] \leq_{\mathrm{C}} [M \vee C \,|\, \mathcal{X}]$.

Algorithm 9 isIncluded($\mathcal{X}$, T)

Require: T is a constraint tree, $\mathcal{X}$ is a constraint.
Ensure: isIncluded($\mathcal{X}$, T) $= \top$ iff $\exists \mathcal{Y} \in \mathcal{C}'_c(T)$, $\mathcal{Y} \subseteq \mathcal{X}$.

```
1:  if T = □ then
2:      return ⊤
3:  end if
4:  if X = ⊤ then
5:      return ⊥
6:  end if
7:  m_1 ← min_{<_π} {m ∈ X}
8:  T_1 ← {(l, T') ∈ T | l = m_1} // T_1 is of size 0 or 1
9:  if ⋁_{(l,T')∈T_1} isIncluded(X \ {m_1}, T') then
10:     return ⊤
11: end if
12: T_2 ← {(l, T') ∈ T | m_1 <_π l}
13: return ⋁_{(l,T')∈T_2} isIncluded(X \ {m_1}, l.T')
```

The correctness and termination of isIncluded are stated in the following proposition.

Proposition 7.6 *Let $\mathcal{X}$ be a constraint and T be a constraint tree. The call to* isIncluded$(\mathcal{X}, T)$ *terminates and* isIncluded$(\mathcal{X}, T) = \top$ *iff there exists a constraint $\mathcal{Y} \in \mathcal{C}'_c(T)$ such that $\mathcal{Y} \subseteq \mathcal{X}$.*

Proof. The termination of isIncluded$(\mathcal{X}, T)$ is due to the fact that $|\mathcal{X}|$ decreases at each recursive call. To prove the equivalence property, we consider each implication by induction.

Let isIncluded$(\mathcal{X}, T) = \top$. If $T = \square$ then $\mathcal{C}'_c(T) = \{\top\}$ and the direct implication is verified in this case because $\top \subseteq \mathcal{X}$. Otherwise, there exists a literal $m_1 = \min_{<_\pi} \{m \in \mathcal{X}\}$ because $\mathcal{X}$ is not empty or line 5 would be triggered. If the call terminates line 10 then there exists a pair $(m_1, T') \in T$ such that isIncluded$(\mathcal{X} \setminus \{m_1\}, T')$ returns true. By induction there exists $\mathcal{Y} \in \mathcal{C}'_c(T')$ such that $\mathcal{Y} \subseteq \mathcal{X} \setminus \{m_1\}$, hence $\mathcal{Y} \wedge m_1 \subseteq \mathcal{X}$. Finally, it is also possible that line 13 returns $\top$, in which case there exists a pair $(l, T') \in T$ such that $m_1 <_\pi l$ (i.e. $(l, T') \in T_2$) and isIncluded$(\mathcal{X} \setminus \{m_1\}, l.T')$ returns $\top$. Then by induction there exists a constraint $\mathcal{Y} \in \mathcal{C}'_c(l.T')$ such that $\mathcal{Y} \subseteq \mathcal{X} \setminus \{m_1\}$, hence $\mathcal{Y} \in \mathcal{C}'_c(T)$ and $\mathcal{Y} \subseteq \mathcal{X}$.

For the converse implication, let us assume that T and $\mathcal{X}$ are such that there exists a constraint $\mathcal{Y} \in \mathcal{C}'_c(T)$ where $\mathcal{Y} \subseteq \mathcal{X}$. If $\mathcal{X} = \top$ then $\mathcal{Y} = \top$, thus $T = \square$ by definition of a constraint tree and line 2 is triggered. As expected, $\top$ is returned. Otherwise $\mathcal{X}$ is not empty and we define $m_1 = \min_{<_\pi} \{m \in \mathcal{X}\}$. If $\mathcal{Y}$ is empty, line 2 is triggered again and we can conclude. Let us assume that

$\mathcal{Y}$ contains at least a literal. We define $l_1 = \min_{<_\pi}\{l \in \mathcal{Y}\}$. Given the ordering constraints imposed on constraint trees, necessarily $(l_1, T') \in T$ where $\mathcal{Y}\setminus\{l_1\} \in \mathcal{C}'_c(T')$.

— If $l_1 = m_1$ then $(l_1, T') \in T_1$ and $\mathcal{Y} \setminus \{l_1\} \subseteq \mathcal{X} \setminus \{m_1\}$. Hence by induction ISINCLUDED$(\mathcal{X} \setminus \{m_1\}, T')$ returns $\top$. Since this call is made line 9, ISINCLUDED$(\mathcal{X}, T)$ also returns $\top$.

— If $l_1 <_\pi m_1$ then $\mathcal{Y} \not\subseteq \mathcal{X}$, raising a contradiction with our hypothesis.

— If $m_1 <_\pi l_1$ then $(l_1, T') \in T_2$ and $\mathcal{Y} \subseteq \mathcal{X}\setminus\{m_1\}$. By induction ISINCLUDED$(\mathcal{X}\setminus\{m_1\}, l_1.T')$ returns $\top$, thus if line 13 is triggered (the only alternative being that line 10 has been reached, which is enough to validate the converse implication), then ISINCLUDED$(\mathcal{X}, T)$ returns $\top$ as desired. ∎

3 Algorithm PRUNEENTAILED for c-trees

Converting the algorithm PRUNEENTAILED$_{e1}$ into PRUNEENTAILED$_{cons}$ (Algorithm 10) involves the same mechanism as for ISENTAILED$_{cons}$ in the previous section. The $\preceq$- and $\subseteq$-tests are concatenated to the one using E-subsumption at line 2 and the rest of the algorithm is modified to recover used literals in M and N through the recursive calls. However, in this case the $\preceq$-comparison involves a c-tree and a clause instead of two clauses, thus we detail its process in PRUNEINF (Algorithm 11). The inclusion test (Algorithm 12: PRUNEINCLUDED) is launched at line 5 of Algorithm 11.

The only role of the algorithm PRUNEINF is to complete the exploration of each clausal branch of the c-tree so that the $\preceq$-comparison between these and C can be done (as explained in the previous section, the incompatibility of $<_\pi$ and $\preceq$ prevents the $\preceq$-test to be done directly on the tree). In practice, once the test $C \not\preceq N$ fails, it also fails in all the subsequent recursive calls. Thus a subprocedure containing only the lines 1 and 5 is used from this point, producing the same result more efficiently.

PRUNEINCLUDED can be seen as a simpler version of PRUNEENTAILED$_{cons}$, where inclusion is tested instead of E-subsumption. It is the pendant of the algorithm ISINCLUDED used in ISENTAILED$_{cons}$, and functions under the exact same principle but with a swap in the roles of the constraint and the constraint tree.

The correction of PRUNEENTAILED$_{cons}$ is stated in Theorem 7.7.

Theorem 7.7 *If T is a c-tree in normal form, $M \vee C$ and N are clauses in normal form, $M \models N$ and* ISENTAILED$_{cons}([C \vee M \mid \mathcal{X}],\ N.T,\ \square,\ \square) = \bot$ *then $\mathcal{C}_c(T_{out}) = \{[D\,|\,\mathcal{Y}] \in \mathcal{C}_c(T)\,|\,[C \vee M\,|\,\mathcal{X}] \not\leq_c [D \vee N\,|\,\mathcal{Y}]\}$, with T_{out} =* PRUNEENTAILED$_{cons}([C\,|\,\mathcal{X}],\ T,\ M,\ N)$.

The corresponding proof is omitted due to its similarity with that of Theorem 6.5. The termination and correction results concerning PRUNEINF and PRUNEINCLUDED are stated respectively in Proposition 7.8 and Proposition 7.9.

Algorithm 10 $\textsc{pruneEntailed}_{cons}([C \,|\, \mathcal{X}],\ T,\ M,\ N)$

Require: T is a c-tree in normal form, $M \vee C$ and N are clauses in normal form, $M \models N$ and $\textsc{isEntailed}_{cons}([C \vee M \,|\, \mathcal{X}],\ N.T,\ \square,\ \square) = \bot$.

Ensure: $\mathcal{C}_c(T_{out}) = \{[D \,|\, \mathcal{Y}] \in \mathcal{C}_c(T) \,|\, [C \vee M \,|\, \mathcal{X}] \not\leq_{\text{C}} [D \vee N \,|\, \mathcal{Y}]\}$, with $T_{out} = \textsc{pruneEntailed}_{cons}([C \,|\, \mathcal{X}],\ T,\ M,\ N)$.

1: **if** $C = \square$ **then**
2: **return** $\textsc{pruneInf}([M \,|\, \mathcal{X}],\ T,\ N)$
3: **end if**
4: select $m_1 \in C$ s.t. $m_{1\downarrow N} = \min_{<_\pi} \{m_{\downarrow N} \,|\, m \in C\}$
5: **if** $m_{1\downarrow N}$ is a contradiction **then**
6: **return** $\textsc{pruneEntailed}_{cons}(C \setminus m_1,\ T,\ M \vee m_1,\ N)$
7: **end if**
8: $T_{\boxdot} \leftarrow \{(\boxdot, T') \in T\}$
9: $T_1 \leftarrow \left\{(l, T') \in T \setminus T_{\boxdot} \,|\, l = u \not\preceq v \wedge m_{1\downarrow N} \succeq l\right\}$
10: $T_{out1} \leftarrow \{(l, \textsc{pruneEntailed}_{cons}(C,\ T',\ M,\ N \vee l) |$
$(l, T') \in T_1 \wedge \textsc{pruneEntailed}_{cons}(C,\ T',\ M,\ N \vee l) \neq \emptyset\}$
11: **if** m_1 is positive **then**
12: $T_2 \leftarrow T \setminus (T_1 \cup T_{\boxdot})$
13: $T_{out2} \leftarrow \{(l, \textsc{pruneEntailed}_{cons}(C \setminus L_l,\ T',\ M \vee L_l,\ N \vee l)) |$
$(l, T') \in T_2 \wedge L_l = \{m \in C \,|\, l_{\downarrow N \vee m} \text{ is tautological}\} \wedge$
$\textsc{pruneEntailed}_{cons}(C \setminus L_l,\ T',\ M \vee L_l,\ N \vee l) \neq \emptyset\}$
14: **return** $T_{out1} \cup T_{out2} \cup T_{\boxdot}$
15: **else**
16: **return** $T_{out1} \cup T \setminus T_1$
17: **end if**

Proposition 7.8 *Let $[C \mid \mathcal{X}]$ be a c-clause, N be clauses and T be a c-tree, all three in normal form. The call $\textsc{pruneInf}([C \mid \mathcal{X}], T, N)$ terminates and the output c-tree $T_{out} = \textsc{pruneInf}([C \mid \mathcal{X}], T, N)$ is such that $\mathcal{C}_c(T_{out}) = \{[D_T \,|\, \mathcal{Y}_T] \in \mathcal{C}_c(T) \,|\, (C \not\preceq D_T \vee N) \vee \mathcal{Y}_T \not\subseteq \mathcal{X}\}$.*

Proof. We proceed by double inclusion and induction. Let $[D \,|\, \mathcal{Y}] \in \mathcal{C}_c(T_{out})$. If $D = \square$ then:

- if $C \not\preceq N$ then T_{out} is returned at line 3 and $C \not\preceq D \vee N$,
- otherwise T_{out} is generated at line 5 and $(\boxdot, \textsc{pruneIncluded}(\mathcal{X}, T')) \in T_{out}$ where $(\boxdot, T') \in T$ and $\mathcal{Y} \in \mathcal{C}'_c(T')$; in this case, Proposition 7.9 ensures that $\mathcal{X} \not\subseteq \mathcal{Y}$.

If $D = l \vee D'$ where $l <_\pi D'$ then $[D \,|\, \mathcal{Y}] \in T_{out1}$ and thus $D' \in \mathcal{C}_c(\textsc{pruneInf}([C \,|\, \mathcal{X}], T', N \vee l))$. By induction, either $C \not<_\pi D' \vee N \vee l$, i.e. $C \not<_\pi D \vee N$, or $\mathcal{X} \not\leq_{\text{E}} \mathcal{Y}$ justifying the direct inclusion. Let $[D \,|\, \mathcal{Y}] \in \mathcal{C}_c(T)$ such that $C \not\preceq D \vee N$ or $\mathcal{Y} \not\subseteq \mathcal{X}$. If $D = \square$ then $[D \,|\, \mathcal{Y}] \in \mathcal{C}_c((\boxdot, T')) \subseteq \mathcal{C}_c(T)$.

- If $C \preceq N$ then line 5 is reached and by hypothesis $\mathcal{X} \not\subseteq \mathcal{Y}$ thus $\mathcal{Y} \in \textsc{pruneIncluded}(\mathcal{X}, T')$ ensuring that $[D \,|\, \mathcal{Y}] \in \mathcal{C}_c(T_{out})$.
- Otherwise line 3 is reached and again $[D \,|\, \mathcal{Y}] \in \mathcal{C}_c(T_{out})$. ∎

Algorithm 11 pruneInf$([C \,|\, \mathcal{X}], T, N)$

Require: T is a clausal-tree in normal form, C in a clause in normal form, N is a clause in normal form.

Ensure: $\mathcal{C}_c(T_{out}) = \{[D \,|\, \mathcal{Y}] \in \mathcal{C}_c(T) | (C \not\preceq D \vee N) \ \vee \ \mathcal{X} \not\subseteq \mathcal{Y}\}$, where $T_{out} =$ pruneInf$([C \,|\, \mathcal{X}], T, N)$.

1: $T_{out1} \leftarrow \{(l, \text{pruneInf}([C \,|\, \mathcal{X}], T', N \vee l)) | (l, T') \in T\}$
2: **if** $C \not\preceq N$ **then**
3: **return** $T_{out1} \cup \{(\boxdot, T') \,|\, (\boxdot, T') \in T\}$
4: **else**
5: **return** $T_{out1} \cup \{(\boxdot, \text{pruneIncluded}(\mathcal{X}, T')) \,|\, (\boxdot, T') \in T\}$
6: **end if**

Algorithm 12 pruneIncluded$(\mathcal{X},\ T)$

Require: T is a constraint tree, $\mathcal{X}$ is a constraint.

Ensure: $\mathcal{C}'_c(T_{out}) = \{\mathcal{Y} \in \mathcal{C}'_c(T) | \mathcal{X} \not\subseteq \mathcal{Y}\}$, with $T_{out} =$ pruneIncluded$(\mathcal{X},\ T)$.

1: **if** $\mathcal{X} = \top$ **then**
2: **return** $\emptyset$
3: **end if**
4: **if** $T = \square$ **then**
5: **return** T
6: **end if**
7: $m_1 \leftarrow \min_{<_\pi} \{m \in \mathcal{X}\}$
8: $T_1 \leftarrow \{(l, T') \in T \,|\, l = m_1\}$
9: $T_{out1} \leftarrow \{(l, \text{pruneIncluded}(\mathcal{X} \setminus \{m_1\},\ T') \,|\, (l, T') \in T_1\}$
10: $T_2 \leftarrow \{(l, T') \in T \setminus T_1 \,|\, l <_\pi m_1\}$
11: $T_{out2} \leftarrow \{\text{isIncluded}(\mathcal{X} \setminus \{m_1\},\ l.T') \,|\, (l, T') \in T_2\}$
12: **return** $T_{out1} \cup T_{out2} \cup (T \setminus (T_1 \cup T_2))$

If $D = D' \vee l$ where $l <_\pi D'$ and $(l, T') \in T$ then either $C \not\preceq D \vee N$ thus $C \not\preceq D' \vee N \vee l$, or $\mathcal{X} \not\subseteq \mathcal{Y}$. In both cases by induction $[D' \,|\, \mathcal{Y}] \in \mathcal{C}_c($pruneInf$([C \,|\, \mathcal{X}], T', N \vee l))$ thus $[D \,|\, \mathcal{Y}] \in \mathcal{C}_c(T_{out1}) \subseteq \mathcal{C}_c(T_{out})$.

Proposition 7.9 *Let T be a constraint tree and $\mathcal{X}$ be a constraint. The call* pruneIncluded$(\mathcal{X}, T)$ *terminate and $\mathcal{C}'_c(T_{out}) = \{\mathcal{Y} \in \mathcal{C}_c(T) \,|\, \mathcal{X} \not\subseteq \mathcal{Y}\}$ where $T_{out} =$* pruneIncluded$(\mathcal{X}, T)$.

Proof. Note that the termination of pruneIncluded is the consequence of the strict decrease of $|\mathcal{X}|$ between each recursive call.

Let $T_{out} =$ pruneIncluded$(\mathcal{X}, T)$. We show by induction that $\mathcal{C}'_c(T_{out}) \subseteq \{\mathcal{Y} \in \mathcal{C}_c(T) \,|\, \mathcal{X} \not\subseteq \mathcal{Y}\}$. Let $\mathcal{Y} \in \mathcal{C}'_c(T_{out})$. If $\mathcal{X} = \top$ then $T_{out} = \emptyset$ thus $\mathcal{Y}$ does not exists. Otherwise let $m_1 = \min_{<_\pi} \{m \in \mathcal{X}\}$ and let $\mathcal{Y} = l \wedge \mathcal{Y}'$ where $(l, T'_{out}) \in T_{out}$ and $\mathcal{Y}' \in \mathcal{C}'_c(T'_{out})$. There are three cases to examine:

— If $\mathcal{Y} \in \mathcal{C}'_c(T_{out1})$ then $T'_{out} = \text{PRUNEINCLUDED}(\mathcal{X} \setminus \{m_1\}, T')$ where $(l, T') \in T_1$, hence $l = m_1$ and $\mathcal{Y} = m_1 \wedge \mathcal{Y}_1$. By the induction hypothesis $\mathcal{Y}' \in \mathcal{C}'_c(T')$ and $\mathcal{X} \setminus \{m_1\} \not\subseteq \mathcal{Y}'$ thus $\mathcal{Y} \in \mathcal{C}'_c(T_1) \subseteq \mathcal{C}'_c(T)$ and $\mathcal{X} \not\subseteq \mathcal{Y}$.
— If $\mathcal{Y} \in \mathcal{C}'_c(T_{out2})$ then $T_{out2} = \text{PRUNEINCLUDED}(\mathcal{X} \setminus \{m_1\}, l.T')$ where $(l, T') \in T_2$. By the induction hypothesis $\mathcal{Y} \not\subseteq \mathcal{X} \setminus \{m_1\}$, thus $\mathcal{Y} \not\subseteq \mathcal{X}$.
— If $\mathcal{Y} \in \mathcal{C}'_c(T \setminus (T_1 \cup T_2))$ then $\mathcal{Y} \in \mathcal{C}'_c(l.T') \subseteq \mathcal{C}'_c(T)$ where $(l, T') \in (T \setminus (T_1 \cup T_2))$ and $m_1 <_\pi l$ by definition of T_1 and T_2, ensuring that $\mathcal{X} \not\subseteq \mathcal{Y}$.

To prove the converse inclusion, let us consider a constraint $\mathcal{Y} \in \{\mathcal{Y} \in \mathcal{C}_c(T) \mid \mathcal{X} \not\subseteq \mathcal{Y}\}$. As in the previous direction, if $\mathcal{X} = \top$ then $\{\mathcal{Y} \in \mathcal{C}_c(T) \mid \mathcal{X} \not\subseteq \mathcal{Y}\} = \emptyset$ and in this case, $T_{out} = \emptyset$ is returned line 2. If $\mathcal{X} \neq \top$ and $\mathcal{Y} = \top$ then by definition of a constraint tree $T = \square$ and $T_{out} = T$ is returned line 5, thus $\mathcal{Y} \in \mathcal{C}'_c(T_{out})$. Otherwise, let $m_1 = \min_{<_\pi} \{m \in \mathcal{X}\}$ and $(l, T') \in T$ such that $\mathcal{Y} \in \mathcal{C}_c(l.T')$. The same three cases as in the converse implication appear:
— If $l = m_1$ then $\mathcal{X} \setminus \{m_1\} \subseteq \mathcal{Y}'$ where $\mathcal{Y} = l \wedge \mathcal{Y}'$ and $(l, T') \in T_1$ thus by the induction hypothesis $(l, T') \in T_{out1}$.
— If $l <_\pi m_1$ then $\mathcal{X} \not\subseteq \mathcal{Y}'$ and $(l, T') \in T_2$ hence by the induction hypothesis $(l, T') \in T_{out2}$.
— If $m_1 <_\pi l$ then $\mathcal{Y} \in T \setminus (T_1 \cup T_2)$.

In the three cases $\mathcal{Y} \in \mathcal{C}'_c(T_{out})$. ∎

4 Summary

All the algorithms exposed in this part of the thesis present the same underlying structure, allowing a depth-first traversal of the tree manipulated. The ISENTAILED family of algorithms allows to detect the redundancy of a clause to the ones stored in a clausal tree, while the PRUNEENTAILED family, as its name indicates, regroups the algorithms that cut branches of a tree if they are redundant to a given clause. In $\mathbb{E}_0$ we used I-subsumption (and E-subsumption succinctly) as redundancy criterion while we focused on E-subsumption in $\mathbb{E}_1$. For c-clauses, C-subsumption was used. All the algorithms were presented in an abstract way. Concrete implementations should benefit from technical improvements such as a lazy evaluation of expressions. Such details, although they cannot improve the overall worst-case complexity of an algorithm, can make a real difference in terms of execution time in many cases.

Part III

Experimental results

This part of the thesis introduces the programs that were developed during the Ph.D. to evaluate the performance of the algorithms described in the first two parts. These programs are described in detail in Chapter 8. The tools implementing other algorithms, used as reference in the experiments and the benchmarks are presented Chapter 9. Finally, Chapter 10 exposes the different experiments and their results.

Chapter 8

Implementation details

Two programs have been realized in the OCaml language [1] during this PhD. The first one, namely `Kparam`, implements the $\mathcal{K}$-paramodulation calculus and its variants, as well as a first version of c$\mathcal{SP}_0$, all of them restricted to $\mathbb{E}_0$. The second one, `cSP`, implements c$\mathcal{SP}$ in $\mathbb{E}_1$. The code is in an archive available at `http://lig-membres.imag.fr/tourret/index.php?&tab=3&slt=tools`.

Both programs accept a subset of the TPTP syntax v5.4.0.0 [75] in their inputs, the restriction to purely equational flat ground CNF formulæ for `Kparam` extended to non-flat formulæ for `cSP`. This subset is presented in the appendix B. Every clause begins with the keyword '`cnf`' and ends with a point. In between are parentheses, that contain three fields. The first one is the identifier of the clause. The second one is its type (e.g. axiom, hypothesis, definition...). Both fields have no impact on the execution of `Kparam` and `cSP`. The third field contains the clause itself. The disjunctive operator is represented by '`|`', equations use '`=`' and disequations use '`!=`'. Terms are represented in the natural way (e.g. '`a`', '`f(a)`') and, in `Kparam`, functional terms like '`f(a)`' are not accepted. Here is an example of (non-flat) input.

Example 8.1 `cnf(cl1,plain,f(a)!=b|a=c).`
`cnf(cl2,plain,g(c,a)=f(b)|g(c,b)=d).`

It represents the formula $(f(a) \not\simeq b \vee a = c) \wedge (g(c, a) \simeq f(b) \vee g(c, b) \simeq d)$.♣

1 `Kparam`

To distinguish between the different calculi implemented in `Kparam`, we use the following notations:

— `Kparam_s` refers to the simple $\mathcal{K}$-paramodulation calculus (see page 46). It is invoked with the `-b` option.

— `Kparam_u` refers to the $\mathcal{K}$-paramodulation calculus with rewriting beforehand using unordered paramodulation (see page 56). It is invoked with

1. `http://ocaml.org/`

the `-up` and `-ur` options.
— `Kparam_r` refers to the $\mathcal{K}$-paramodulation calculus with rewriting on the fly (see page 64). It is invoked with the `-o2` and `-o5` options.
— `cSP_flat` refers to the constrained Superposition calculus restricted to $\mathbb{E}_0$, i.e. c$\mathcal{SP}_0$ (see page 78). It is invoked with all the `-cs*` options.

1.1 Implementation details

`Kparam` contains several features that are worth mentioning. Some of them are common to all the variants.

Terms and ordering. The first notable feature of `Kparam` is the way it handles terms. When an input file is parsed, terms are ordered as they appear. This produces a strict order because all the terms in $\mathbb{E}_0$ are constants.

Example 8.2 The following TPTP input:

```
cnf(cl1,plain,b!=a|c=d).
cnf(cl2,plain,a=e).
```

induces the ordering $b < a < c < d < e$ in `Kparam`. ♣

Due to this fact, it is possible to associate a unique integer with each term, in such a way that comparing two terms amounts to comparing the corresponding integers. Thanks to the use of the aforementioned ordering, the term-integer association is done at parse time and `Kparam` can immediately forget the terms original names. In `Kparam` the type of terms is thus:

```
type term = T of int
```

This lightens considerably the following operations. The terms' original names are stored in an array so that they can be recovered at the time of the output file generation.

Normalization. Another interesting feature of `Kparam` is the normalization of clauses. To realize this operation, a Union-Find data structure (implemented in OCaml over a persistent array data structure [14]) is used to compute the equivalence classes corresponding to the clause. First `Kparam` scans the literals contained in the clause. If the literal under consideration is positive then `Kparam` stores it in a list. Otherwise `Kparam` recovers the terms in the (negative) literal and fuses them in the Union-Find data structure where this merge is propagated appropriately. Once this is done, it is possible to query the Union-Find data structure for the representative of any term in the clause. The negative literals in normal form are directly extracted from this data structure and the positive ones are rewritten using the term representatives. Then the Union-Find data structure is discarded.

Storage of clauses. Normal clausal tree data structures (see Definition 4.7) are used to represent both the waiting and processed sets. This allows `Kparam` to perform redundancy detection in both sets. A first version of `Kparam` had been developed in which only the processed set was a normal clausal tree and the memory was constantly overflowing due to redundancies in the waiting set.

Selection of clauses in the waiting set. The clause selected in the waiting set is arbitrarily chosen among those with the smallest number of literals.

Options. In terms of options specific to a given variant of `Kparam`, we have:

- For `Kparam_u`, the option `-up` corresponds exactly to the method described in Section 2 of Chapter 2, and the option `-ur` is similar except that it intertwines the unordered paramodulation with the rewriting steps (like the method described in Section 3 of the same chapter does with $\mathcal{K}$-paramodulation). Thus instead of computing the full set of atomic prime implicates of a formula, this method computes only a fraction of them that is enough to recover the whole set by transitivity, e.g. where `-up` computes $a \simeq b$, $b \simeq c$ and $c \simeq a$, the option `-ur` only computes $a \simeq b$ and $b \simeq c$. Then, in both cases, the rewritten formula is fed to the $\mathcal{K}$-paramodulation calculus that computes the remaining prime implicates (atomic and non-atomic) as described in said section.
- For `Kparam_r`, the options `-o2` and `-o5` correspond to different implementations of the collision criterion of Chapter 2 Section 3 that decides which clauses from the processed set need to return in the waiting set after a rewriting. Using `-o2`, the collision criterion tests exactly the $\langle a, b\rangle$-neutrality criterion, i.e. if $D[a/b]^+ \neq D[a/b]_\downarrow{}^+$ then a collision between the saturation process and the rewriting is detected and D is transferred from the processed set to the waiting set. Since this operation is costly (the positive literals must be compared one by one) an overapproximation was created in `-o5` where the collision criterion simply tests if the maximal term of the unit prime implicate found appears in the clause, i.e. if $a \simeq b$ where $a \succ b$ is used for rewriting and D contains the term a, then a collision occurs. This criterion is (less costly and) more general than $\langle a, b\rangle$-neutrality, thus it preserves the completeness of the calculus. Other criteria where developed (`-o1`, `-o3`, `-o4`) but they did not cover the $\langle a, b\rangle$-neutrality criterion and thus jeopardized the completeness of the whole process, hence they are now deprecated.
- for `cSP_flat`, seven options controlling how a clause in the waiting set is selected, one (`-csi`) that implements an additional index (it is presented later) and three implementing filters:
  - `-cov` that keeps only the clauses entailing one of the formula's original clauses,
  - `-csize k` that keeps only the clauses that have a length smaller or equal to `k`, and
  - `-isize` that combines the `-csize` and `-csi` options.

Index. The index introduced in `cSP_flat` by the `-csi` option is a hashtable that stores the same clauses as the processed set but groups them according to their maximal term, that appears in the literal selected for the c$\mathcal{SP}$ calculus. This speeds up the generation of new clauses because only relevant clauses are paired by the rules, e.g. the pair of clauses $(a \not\simeq b, c \simeq d)$ is never considered by the c$\mathcal{SP}$ calculus, even if one is located in the processed set and one is the selected clause.

1.2 Preprocessing tools

A number of tools were developed in parallel of `Kparam` to preprocess inputs. They are available in the tool folder of the archive previously mentioned.

`Equationalizer` accepts flat ground cnf TPTP inputs containing booleans and returns fully equational formulæ. To do so, it uses t as a constant standing for 'true' and converts propositional terms p and $\neg p$ respectively into $p \simeq t$ and $p \not\simeq t$.

Remark 8.3 *The constraint $t < x$ for any $x \in \setminus\Sigma_0\{t\}$, that permits to simulate resolution can be artificially enforced in the program by adding the tautological clause $t \simeq t$ at the beginning of the file.*

`Flattener_for_kparam` is a tool that flattens non-flat TPTP cnf inputs by replacing non-flat terms with fresh constants and instantiating the substitutivity axiom when necessary. For example, if c and d are introduced to replace $f(a)$ and $f(b)$ then the clause $a \not\simeq b \vee c \simeq d$ must also be added.

2 cSP

2.1 Implementation details

`LogTk`. The main difference between `cSP` and `cSP_flat` is the use of the `LogTk` library [16]. In `cSP` its main use is to manage everything related to terms. Given that `cSP` handles non-flat terms, the simple order implemented in `Kparam` is no longer usable because it is not necessarily a reduction order in this case, as the following example shows.

Example 8.4 The order of appearance in the following input file is $a \prec b \prec c \prec d \prec f(b) \prec f(a)$.

```
cnf(cl1,plain,a=b|c=d).
cnf(cl2,plain,f(b)!=f(a)).
```

In a reduction order, given $a \prec b$ necessarily $f(a) \prec f(b)$ because it is a rewrite order (see page 29). Here instead $a \prec b$ and $f(b) \prec f(a)$, thus this order is not a reduction order. ♣

Fortunately, LogTk implements several reduction orders, e.g. the KBO (see Example ii.9) and the RPO (Recursive Path Ordering [2]) in its multiset and lexicographic variants. Among those, cSP relies (arbitrarily) on the KBO.

Normalization. For the normalization of clauses, a Union-Find data structure is no longer sufficient.

Example 8.5 In the following input, the equivalence classes associated with cl are $\{a, b\}, \{f(a), f(b)\}$.

```
cnf(cl,plain,a!=b|f(a)=f(b)).
```

In a Union-Find data structure, the propagation of the identity $a \equiv_{\texttt{cl}} b$ to $f(a) \equiv_{\texttt{cl}} f(b)$ is not automatically computed. ♣

Instead, a congruence closure algorithm [51], also implemented in LogTk, is used. The advantage of such an algorithm is that the propagation of unification from the subterms to the superterms is done automatically. Going back to the previous example where a and b are unified, using a congruence closure algorithm ensures that $f(a)$ and $f(b)$ are put in the same equivalence class as well. Any such other terms, e.g. $g(c, a)$ and $g(c, b)$ are also unified.

Options. The options of cSP are:

- -max-size,-max-neg and -max-depth, three filters respectively limiting the number of literals appearing in the prime implicates, their number of negative literals and the depth of the terms.
- -cov, another filter accepting only prime implicates that entail one of the clauses of the input formula (see Example 3.31 and preceding paragraph).
- -odiff, an option that, when a timeout is reached, compares the set of processed implicates to the original input formula and counts how many new implicates (not in the original input) have been processed. This option also sets the function that selects clauses in the processed set (see description below) to take into account only the clause part of constrained clauses.

A difference with Kparam is that the options of cSP can be combined.

Example 8.6 A call to cSP with the options -max-depth 1 and -max-size 2 returns only the prime implicates of size 1 or 2 made of terms of depth 0 or 1. If $f(a) \simeq b$, $a \simeq c \vee b \simeq d \vee g(e) \simeq d$ and $f(g(f(a))) \simeq d \vee g(e) \simeq b$ are prime implicates of a formula, only the first one is computed. ♣

Clause selection. The order in which the clauses are extracted from the waiting set is controlled by the parameter cs_comp_func. It is easily modifiable in the code and three orders are considered.

- a sort based only on the size of the clausal part of constrained clauses,
- a lexicographic sort based on the size of both parts of tree clauses starting with the clausal parts,

— the same as the previous one but starting with the constraints.
All three are used in increasing order, i.e. the smallest clause is the one selected. By default, the third one is used so as to find the simplest prime implicates first.

Assertable terms. To ensure the termination of the calculus, `cSP` generates only implicates built on a finite number of terms that are introduced by the Assertion rules (see Chapter 3, Section 2). By default, these assertable terms are the terms already occuring in the input formula. However it is possible to specify additional assertable terms to `cSP`. To do so, a TPTP input file with the extension '.conf' that contains a cnf formula containing all assertable terms can be provided to `cSP` along with the input formula.

2.2 Preprocessing tools

A flattener has been developed in `cSP` but is not available as a standalone tool (the `Kparam` flattener can be used in this case). Instead, it is used in the `full_array_preprocessing`, a script that converts TPTP cnf files about the array theory into TPTP (untyped) ground cnf files, based on the method described in [11] to generate equisatisfiable problems free of the axioms of the theory of arrays with extensionality. The conversion proceeds in three steps. First the `Array_preprocessing` with option `-arr1` flattens and decomposes the input into an operational and a definitional part. The definitional part contains all the clauses of the form 'store$(a, i, e) \simeq b$' plus the axioms of the array theory. The clauses allowed in the operational part are strictly flat clauses and clauses of the form 'select$(a, i) \simeq e$'. Then the E theorem prover is used to saturate the definitional part. Finally, the `Array_preprocessing` realizes the necessary instantiations of the axioms of the array theory.

Example 8.7 (Example 66 in [11]) The following input:

```
cnf(a1,axiom,select(store(A,I,E),I)=E).
cnf(a2,axiom,select(store(A,I,E),J)=select(A,J)|
                I=J).
cnf(c1,plain,store(b,i,d)=a).
cnf(c2,plain,select(b,i2)=e).
cnf(c3,plain,select(c,i2)=e2).
cnf(c4,plain,a=c).
cnf(c5,plain,i!=i2).
cnf(c6,plain,e!=e2).
```

is first split in two. The operational part is:

```
cnf(c2,plain,select(b,i2)=e).
cnf(c3,plain,select(c,i2)=e2).
cnf(c4,plain,a=c).
cnf(c5,plain,i!=i2).
cnf(c6,plain,e!=e2).
```

Since the ground clauses of the original formula are flat, no flattening was necessary to generate the operational part. The definitional part is:

```
cnf(a1,plain,select(store(X0,X1,X2),X1)=X2).
cnf(a2,plain,select(store(X0,X1,X2),X3)=select(X0,X3)|
                X1=X3).
cnf(c1,plain,store(b,i,d)=a).
```

The renaming of the variables (in the definitional part) is an internal mechanism of `LogTk`. Then E saturates the definitional part:

```
cnf(i_0_2,plain,(select(store(X1,X2,X3),X2)=X3)).
cnf(i_0_1,plain,(select(store(X1,X2,X3),X4)=select(X1,X4)|
                  X2=X4)).
cnf(i_0_3,plain,(store(b,i,d)=a)).
cnf(i_0_4,plain,(select(a,i)=d)).
cnf(i_0_5,plain,(select(a,X1)=select(b,X1)|i=X1)).
```

Again, the renaming of the clauses and variables is an internal process of E. In this saturation two new clauses have been created. After the instantiation the final result is:

```
cnf(pi2,plain,select(b,i2)=e).
cnf(pi3,plain,select(c,i2)=e2).
cnf(pi4,plain,a=c).
cnf(pi5,plain,i!=i2).
cnf(pi6,plain,e!=e2).
cnf(pi1,plain,store(b,i,d)=a).
cnf(pi0,plain,select(a,i)=d).
cnf(pi7,plain,select(a,i2)=select(b,i2)|i=i2).
cnf(pi8,plain,select(a,i)=select(b,i)|i=i).
```

The non-ground clause `i_0_5` from the previous step has been used to generate the clauses `pi7` and `pi8`. The other clauses are extracted unchanged from the operational and saturated definitional part. ♣

3 Summary

Table 2 & Table 3 summarize the different options available in `Kparam` and `cSP`.

`Kparam_s`	$\mathcal{K}$-paramodulation calculus (see Part I Chapter 2 Section 1)	`-b`	basic implementation
`Kparam_u`	$\mathcal{K}$-paramodulation calculus with atomic implicate generated in advance using unordered paramodulation (see Part I Chapter 2 Section 2)	`-up`	rewriting between the unordered paramodulation and the $\mathcal{K}$-paramodulation.
		`-ur`	rewriting propagated during the unordered paramodulation.
`Kparam_r`	$\mathcal{K}$-paramodulation calculus with atom rewriting on the fly	`-o2`, `-o5`	different collision criterion (see page 78)
`cSP_flat`	c$\mathcal{SP}$ calculus in $\mathbb{E}_0$ (see Chapter 3)	`-cs1,... -cs7`	different selection function used on the waiting set
		`-csi`	c$\mathcal{SP}$ calculus with index (see page 130)
		`-csize, -cov`	different filters (see page 129)
		`-isize`	`-csize` filter plus index

Table 2 – Summary of the different options of `Kparam` and `cSP`

`cSP`	c$\mathcal{SP}$ calculus in $\mathbb{E}_1$ (see Chapter 3)	`-max-size, -max-neg, -max-depth, -cov`	different filters (see page 131)
		`-reg`	regression option for comparison with `Kparam`and `cSP_flat`
		`-odiff`	tells how many new clauses have been generated if a timeout is reached

Table 3 – Summary of the different options of `Kparam` and `cSP`

Chapter 9

Experimental context

To assess the behavior of our algorithms, we searched for appropriate benchmarks and programs with similar functionalities to compare ours with.

1 Reference tools

The difficulty in selecting programs of reference to conduct our experiments lied in the small number of such programs that were available. In fact, we found only two prime implicate generation tools on the web [1], namely `ritrie` [2] [47], a prime implicate generator in propositional logic , and `Mistral` [19], an SMT solver which "decides satisfiability of formulas in the theory of linear arithmetic over integers and theory of equality with uninterpreted functions [...] which can be used for performing abductive inference" [3].

It turns out that we did not use these tools in our experiments. For the first one, the reason is that it is not efficient enough. This may be explained by the fact that it was built to perform efficient querying of an already generated set of prime implicates, rather than to compute efficiently the said set (which it can nevertheless do). As for the `Mistral` SMT solver, it cannot be compared with our work, because its prime implicate generation is not complete. More generally the approach in [19] applies to any theory admitting quantifier-elimination. Thus this property does not hold for the logic considered here since the elimination of function symbols would require to handle second-order quantification.

Thanks to the kindness of their respective authors, we were also able to obtain two other tools:

1. Since then, a third one has been added (summer 2015): the propositional prime implicate generator `primer` [56], available at `http://logos.ucd.ie/web/doku.php?id=primer` . Since it is posterior to the experiments presented in this thesis, results about this tool are not included. They remain as future work.
2. `http://www.cs.albany.edu/ritries/prototype.html`
3. `http://www.cs.utexas.edu/~tdillig/mistral/index.html`

`Zres` is another prime implicate generator in propositional logic that clearly outperformed `ritrie` on all of our benchmarks. Among the different options available in `Zres`, the one we chose is the "Tison" strategy since it gives the best results. `Zres` accepts input files in the DIMACS cnf format, which is a standard for cnf propositional logic. It is a very simple text format originating from the DIMACS challenges [4].

`SOLAR`, "(SOL for Advanced Reasoning) is a first-order clausal consequence finding system based on the SOL (Skip Ordered Linear) tableau calculus." [5]. It can handle equational reasoning through the use of modification methods [37], although it remains a very difficult problem for systems based on the connection tableau method. There are several experimental settings in which `SOLAR` can deal with equational reasoning. Among those, we used the fastest for which completion was guaranteed, i.e. the '`-eq snmt`' option without any simplifications ('`c`', '`cc`' or '`ccc`'). The `SOLAR` input format is inspired from an old version of TPTP where the syntax of the clauses is slightly different from the current one. They are represented between brackets with the literals separated by a comma. Equations use the '`equal`' function preceded by '`+`' or '`-`' to indicate the sign of the literal, e.g. '`-equal(a,b)`' stands for $a \not\simeq b$. In addition, at the end of the file, a line is added that controls which filters must be applied.

Example 9.1 A `SOLAR` input.

```
cnf(cl1,top_clause,[-equal(f(a),b),+equal(a,c)]).
cnf(cl2,top_clause,[+equal(g(c,a),f(b)),+equal(g(c,b),d)]).
pf([ALL]).
```

Note that this is the same formula as the one in Example 8.1 ♣

The keyword '`top_clause`' is specific to `SOLAR`. It indicates that the clause can be used from the start to derive implicates. The last line starting with '`pf`' is also a specificity of `SOLAR` input files. `pf` stands for production field. It acts as a powerful filtering mechanism that allows for a precise control of the form of the literals in the clauses generated and can also enforce conditions on said clauses. For example '`pf([ALL]).`' generates all the possible prime implicates. Unfortunately, using this production field introduces a bias in the comparisons with our programs because the modification methods used introduce non-equational and non-ground terms in the implicates generated. A filter better suited to our purpose is '`pf([+-equal(_,_)])`', where '`+-`' means positive and negative terms. It ensures that the terms generated are equational. This way, `SOLAR` produces outputs that are very close to `cSP` outputs, with the notable difference that `SOLAR`'s may contain variables that are added during the transformation process. These non-ground clauses may subsume several ground clauses. A solution to recover directly the exact desired (ground) result could be obtained by computing beforehand all the literals that could appear in ground clauses and adding them to the production field, e.g. replacing '`pf([ALL]).`' in the former example with:

4. http://dimacs.rutgers.edu/Challenges/
5. http://solar.nabelab.org/

```
pf([+-equal(a,a),
    +-equal(a,b),
    ...
    +-equal(b,f(a)),
    +-equal(b,f(b)),
    ...
    +-equal(c,g(a,a)),
    ...
    ]).
```

We rejected this solution as too impractical. Besides, the size of the production field would become roughly exponential with regard to the size of the formula. The script `tptp2solar` (available in the tool folder of the archive) was created to convert TPTP cnf formulæ to `SOLAR` inputs.

2 Benchmarks

Flat random benchmarks. In parallel with the development of `Kparam`, we looked for benchmarks in ground flat equational logic and found none. Our attempts with flattened ground problems from the TPTP library were failures, because `Kparam` cannot generate all the prime implicates of such big formulæ with reasonable time (and memory) constraints. Thus we decided to create our own benchmark. It is made of a thousand randomly generated formulæ in TPTP format. Their propositional equivalents (DIMACS) were obtained by instantiating the transitivity axioms for all constant symbols appearing in them – the reflexivity and commutativity axioms are encoded directly in the transformation by orienting and simplifying the equations. These formulæ were generated using `gen.pl`, a Prolog program developed by Nicolas Peltier. The command used to generate the benchmark in the Prolog interpreter after loading `gen.pl` was '`multi-random(6,8,1,5,200).`' creating in one run 200 formulæ of 6 clauses build on 8 constants and containing between 1 and 5 literals.

Non-flat random benchmarks. To evaluate the performances of `cSP`, we created another benchmark by relying again on `gen.pl`. The formulæ of this benchmark are non-flat and were generated using the following commands:

```
generate_random_signature(S,0,a,a*5,a*5),
multi_random_non_flat(c,c,S,d,1,l,n).
```

where

- `a` $\in \{1, 2\}$, is the maximal arity of the terms and `a` $* 5$ is the size of the signature [6],
- `c` $\in \{2, 3, 4\}$ is the number of clauses in a formula,
- `d` $\in \{1, 2\}$ is the maximal depth of the terms,

6. By design, the signature contains at least two literals of arity zero.

— `l` $\in \{2, 3\}$ is the maximal number of literals in a clause (the minimum being 1),
— `n` $= 6$, is the number of formulæ generated with the given parameters.

The formulæ are stored in files that are named with the following convention: `sa_d_c_l` (e.g. `s1_2_4_2`). In total, this benchmark contains 144 formulæ.

Other benchmarks. In the SMT-LIB database [5], we found families of formulæ that, with some preprocessing, could be used in our experiments.

In the QF_UF logic, i.e. "unquantified formulas built over a signature of uninterpreted (i.e., free) sort and function symbols"[7], we selected eight formulæ that we converted into TPTP inputs using the `SMTtoTPTP` tool[8]. They were chosen for being ground and satisfiable. After clausification, the translated formulæ contain from a few hundreds to more than a thousand clauses. The aim of this benchmark is to help evaluate the behavior of `cSP` on large formulæ.

Another logic of interest was QF_AX, i.e. "closed quantifier-free formulas over the theory of arrays with extensionality"[9]. They are synthetic benchmarks that model some properties in the SMT theory of arrays with extensionality, namely:

— the order in which the elements are stored in an array do not matter (`storecomm` benchmarks),
— some swapping of elements between cells of an array are commutative (`swap` benchmarks),
— swapping elements between identical cells of equal arrays generate equal arrays (`storeinv` benchmarks)

The benchmarks labeled with `invalid` have been tweaked to falsify the property so that the problem becomes satisfiable (hence has prime implicates distinct from the empty clause). Given that `cSP` cannot handle SMT-LIB inputs or the theory of arrays, we preprocessed the benchmarks by first converting them to TPTP and then applying `full_array_preprocessing` (see previous chapter, Section 2.2). As shown in [1], these problems can be nontrivial to solve even for state-of-the-art theorem provers like `E` [69] and one cannot expect the entire set of prime implicates to be generated in reasonable time. We use them mainly to evaluate the impact of our redundancy-pruning technique on the number of superposition inferences carried out by blocking the Assertion rules inferences (i.e. applying a filter blocking all implicates apart from the empty one), allowing the comparison of `cSP` with the `E` theorem prover.

7. http://smtlib.cs.uiowa.edu/logics.shtml
8. http://users.cecs.anu.edu.au/~baumgart/systems/smttotptp/
9. http://smtlib.cs.uiowa.edu/logics.shtml

Chapter 10

Results

In this chapter, we present the results of the different experiments that were conducted during the PhD. In a first set of experiments, we established the superiority of `Kparam_r` over `Kparam_u`, `Kparam_s` and `Zres` by performing systematic time comparisons on the random flat benchmark, along with other observations that explain the results. A second set of experiments concerns `cSP_flat`, the impact of its selection function (the one that chooses the next clause to use in the waiting set) as well as the impact of the index from the `-csi` option plus a comparison with `Kparam`. The following experiments focus on `cSP`, comparing it to `cSP_flat` and `SOLAR`. Finally we examine the possibilities offered by the filtering mechanism of `cSP`. All of the experiments were run on a machine equipped with an Intel core i5-3470 CPU and 4×2 GB of RAM running Ubuntu 14.04.

1 Kparam

The thousand formulæ of the random flat benchmark were used to evaluate the performance of `Kparam` and its variants.

1.1 Kparam vs Zres

In this experiment, we first compare the performance of `Kparam_s` against that of `Zres`. The results are shown in the graphs of Figure 12. Plot (12a) is a comparison (using a logarithmic scale for the X axis) of the number of prime implicates found by `Zres` for the propositional formulæ (X axis) with those found by `Kparam_s` for the equivalent equational problems (Y axis). Our results indicate that the number of prime implicates is exponentially smaller in equational logic than in propositional logic. This observation is understandable if we take into account the numerous instantiations of the transitivity axiom that are necessary to translate equational formulæ to propositional logic, the many instances of equivalent clauses that are generated in a purely propositional

setting as well as those that are logically entailed by the equality axioms. This means that the propositional output contains a lot of redundancy that has to be deleted in a post-processing step, a problem that our method averts.

The results shown in Plot (12b) depict the compared run time (in seconds) of `Kparam_s` (Y axis) and `Zres` (X axis) on our benchmark. Note that the run time for `Zres` represented here does not include the aforementioned post-processing step (not implemented). On Plot (12b) the results are only slightly in favor of `Kparam_s`. In fact, `Kparam_s` is at least twice as fast as `Zres` 46% of the time, and globally faster 55% of the time. In comparison `Zres` is twice as fast as `Kparam_s` in 34% of the benchmark. Thus our method is globally more efficient. By examining the worst case results where `Zres` outperforms `Kparam_s`, we observe that these formulæ are mostly those where unit clauses are computed. `Kparam_s` is not well-suited for this class of formulæ because it does not use equational unit propagation techniques. If we focus only on formulæ with no positive unit implicates, then `Kparam_s` is faster 80% of the time.

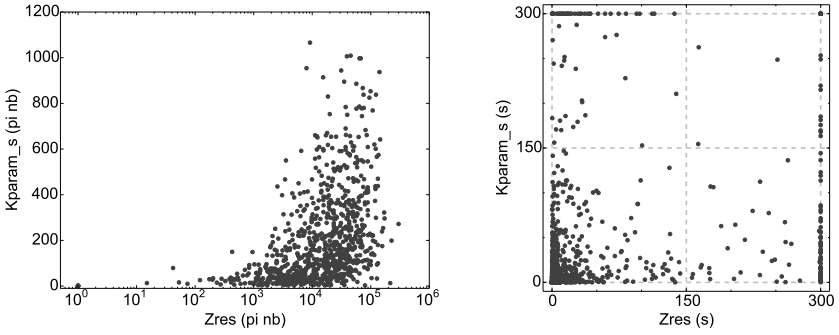
Plot (12c) represents the relative number of implicates (Y axis) and prime implicates (X axis) generated by `Kparam_s`. The results show a quadratic growth of the total number of implicates generated, hence the importance of the redundancy elimination techniques from Part II. This suggests that a lot of time could be gained by constraining the inference rules so as to generate less non-prime implicates (which led to the creation of the c$\mathcal{SP}$ calculus described Part I Chapter 3).

1.2 `Kparam_u` and `Kparam_r`

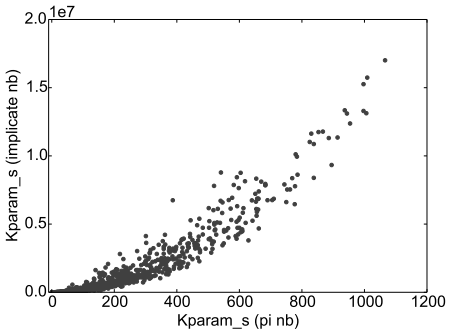
The variants of the $\mathcal{K}$-paramodulation calculus implemented in `Kparam_u` and `Kparam_r` were developed to handle the formulæ with atomic prime implicates more efficiently than `Kparam_s`. To select the best variant, we started with the internal comparison of the options respectively available in `Kparam_u` and `Kparam_r`, then we compared the best of both with each other in order to determine the best setting of parameters (see Table 2 page 134 for a summary of the options).

Plot (13a) shows the execution time of the `-up` (X axis) and `-ur` (Y axis) options. As expected, when the formulæ have no atomic implicates (light gray dots) the results are equivalent, because in this case both programs process inputs the same way. In contrast, when atomic implicates are involved the results vary a lot. Overall, 67% of the formulæ with atomic implicates are handled more efficiently by the `-ur` option than by the `-up` option. Thus `Kparam_u` is more efficient if rewriting is applied eagerly during the unordered paramodulation step, rather than waiting until all unit positive implicates are generated.

Plot (13b) shows that there is next to no difference between the `-o2` and `-o5` options. The expected improvement of the simplification implemented in the `-o5` option is not verified experimentally. This could indicate that the cost of reexamining more clauses (in the waiting set) compensates the benefits of relaxing the collision criterion, but in fact, there are less than ten cases in which the numbers of processed clauses differ using `-o2` and `-o5`. In these cases,



(a) Prime implicates (b) Execution time



(c) `Kparam_s`

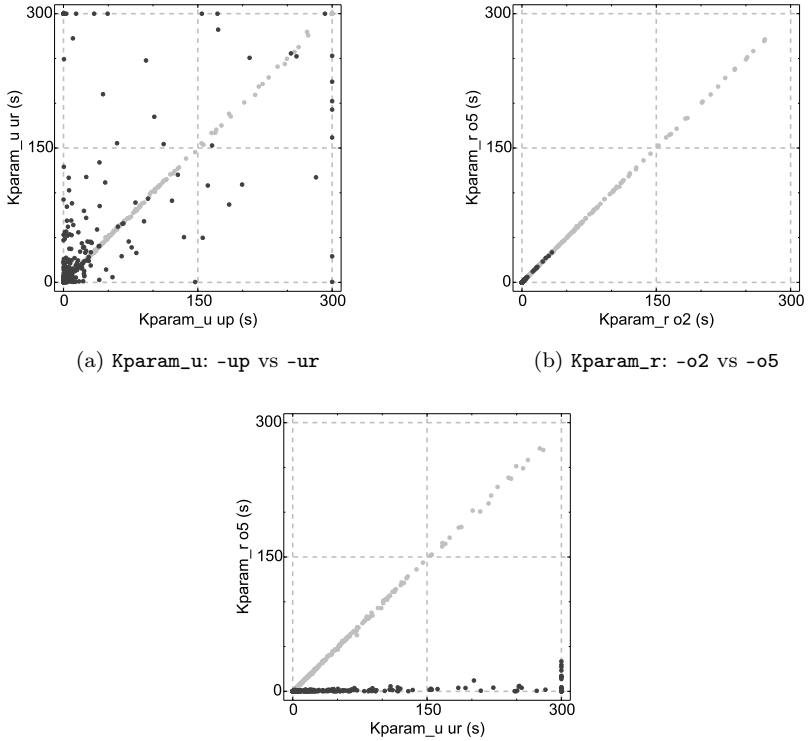
Figure 12 – `Kparam_s` vs `Zres`, random flat benchmark

the difference is of just one clause thus the conclusion is that `-o5` is a good approximation of `-o2`. A first hint of the efficiency of `Kparam_r` is the fact that all the formulæ with atomic implicates (dark gray dots) are processed in less than fifty seconds. Since a finer analysis reveals that `-o5` is more efficient than `-o2` (although not enough to make a real difference) on 68% of the benchmark, we selected this option for the comparison with `Kparam_u`.

The results presented in Plot (13c) confirm the efficiency of `Kparam_r` compared to `Kparam_u`. As with the other plots, the light gray dots represent the formulæ with no atomic implicates, on which `Kparam_u` and `Kparam_r` are roughly equivalent and the formulæ with atomic implicates are represented by dark gray dots. Since `Kparam_r` is clearly more efficient on the latter formulæ than `Kparam_u`, we select this variant for the comparison with `Kparam_s`.

1.3 `Kparam_r` vs `Kparam_s`

Before comparing `Kparam_r` and `Kparam_s`, a first result worth mentioning is that in `Kparam_r` the execution time of the last processing step, i.e. the recovery



(a) `Kparam_u`: `-up` vs `-ur`

(b) `Kparam_r`: `-o2` vs `-o5`

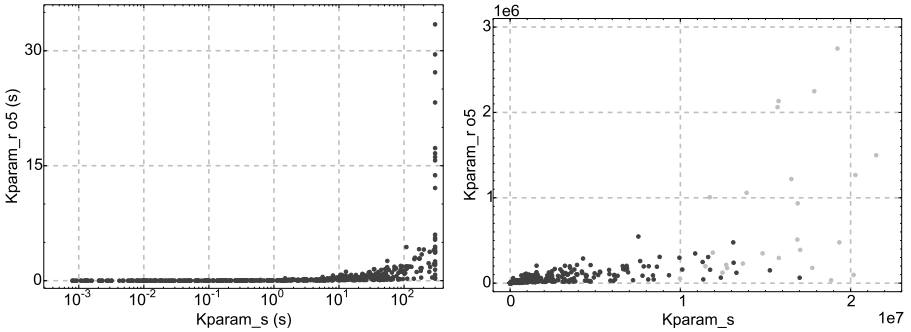
(c) `Kparam_u` (`-ur`) vs `Kparam_r`(`-o5`)

Figure 13 – `Kparam_u` and `Kparam_r` time comparisons, random flat benchmark

of the original solution, is quasi-negligible no matter what the total execution time is: the maximum is less than one second and the mean is 0.04 seconds. In this benchmark, it always represents less that 1 percent of the total execution time [1].

Another interesting indicator of the relative superiority of `Kparam_r` compared to `Kparam_s` is the fact that while 7% of the benchmark reaches timeout with `Kparam_s`, only 4% do so using `Kparam_r`. In our benchmark 49% of the formulæ have no atomic implicates (the light gray dots in Figure 13) and, as expected, `Kparam_s` and `Kparam_r` merely coincide on such problems. Results concerning the remaining 47% of the benchmark are presented in Figure 14. On Plot (14a) the gain of going from `Kparam_s` to `Kparam_r` with regards to the execution time can be observed. A logarithmic scale is used for the X axis to highlight that this graph empirically indicates an exponential gain for our benchmark. The results in Plot (14b) compare the number of implicates generated by

1. Note that this is a characteristic of randomly generated formulæ but it may not always be true.



(a) execution time (b) generated implicates

Figure 14 – `Kparam_r` vs `Kparam_s`, random flat benchmark (formulæ with atomic implicates only)

Number of Atomic implicates	0	1	> 1	> 0	Total
`Kparam_s`	69%	28%	25%	27%	48%
`Kparam_r`	69%	86%	84%	85%	77%

Table 4 – Percentage of Tests Executed Twice Faster than `Zres`

`Kparam_s` and `Kparam_r`. There are two kinds of dots represented on the graph: dark gray and light gray ones, the latter represent tests for which `Kparam_s` reaches the 5 minutes timeout before terminating. The difference of scale between the X and Y axes shows that some formulæ with atomic implicates, that `Kparam_s` cannot solve by computing more than ten million implicates, can be solved by `Kparam_r` with less than two million implicates generated.

Our original motivation was to improve `Kparam_s` on formulæ with atomic implicates (denoted by f.a.i. in this paragraph) compared to `Zres`. We summarize these improvements in Table 4. The observation that `Zres` is more efficient on f.a.i. is apparent in the first line of the table, where only 27% of these formulæ are executed at least twice faster with `Kparam_s` than with `Zres`, while on 69% of the rest of the benchmark `Kparam_s` is twice faster than `Zres`. The second line of Table 4 shows an additional 58% of the f.a.i. are executed twice faster using `Kparam_r` than using `Zres`, validating this work as a real improvement over `Kparam_s`. The results also distinguish between formulæ with a single atomic implicate (71% of the f.a.i.) and several ones (29% of the f.a.i.). A slight improvement of the performance is noted for the latter, but not as significant as the gap between none and one atomic implicate.

2 cSP_flat

We now assess the performance of cSP_flat that implements the c$\mathcal{SP}$ calculus on flat clauses defined in Chapter 3, Section 2. We first determine the best tuning of parameters for this calculus (especially concerning the choice of the selection function) and we compare the obtained procedure with Kparam_r.

2.1 Choosing a selection function

In contrast to Kparam, which simply orders the clauses according to their number of literals and selects the shortest one, there are several natural ways of ordering the constrained clauses generated by cSP_flat depending on whether we consider the length of the clause part, of the constraint part, or both, and it is not clear which option is best.

In this experiment, we examine different selection functions that can be used by the saturation loop on the waiting set to select clauses (see page 33), the following options are implemented, where $[C \,|\, X]$ is a constrained clause.

- `-cs1` order based on $|C| + |X|$,
- `-cs2` lexicographic order based on $(|C| + |X|, |C|)$,
- `-cs3` lexicographic order based on $(|C| + |X|, |X|)$,
- `-cs4` order based on $|C|$,
- `-cs5` order based on $|X|$,
- `-cs6` lexicographic order based on $(|C|, |X|)$,
- `-cs7` lexicographic order based on $(|X|, |C|)$,

Based on this description it is clear that `-cs2`, `-cs4` and `-cs6` give a more important role to the clausal part, while `-cs3`, `-cs5` and `-cs7` do the same with the constraint. The first family of option prioritizes the fast generation of any implicate by trying to get rid of the clausal part of c-clauses first. The other family prioritizes the generation of the smallest implicates even if the derivations that generate them are longer. In Table 5 the execution times of cSP_flat with these options is compared. Each cell contains the percentage of the benchmark on which the row option is faster than the column option. The rows and columns are sorted by efficiency (the most efficient option first). This highlights that the clausal part is the most important (on random formulæ) since the best option is `-cs6` followed by `-cs4` and then `-cs2`

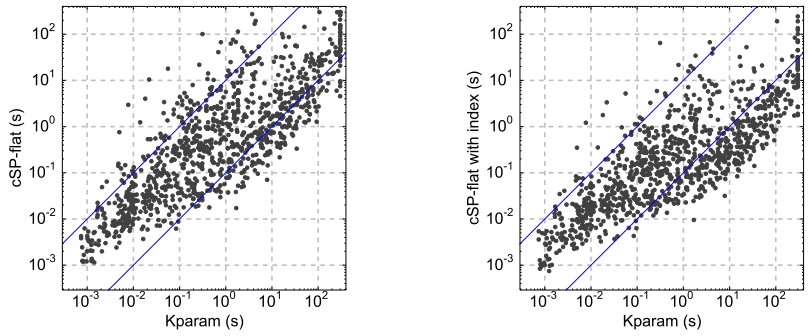
2.2 Impact of the index on cSP_flat

The `-csi` option implements an index to speed up the computation on cSP_flat (see the index description page 130). In fact on 98% of the random flat benchmark the index does indeed speed up the computation of cSP_flat. The remaining 2% are formulæ that are executed by cSP_flat in less than 0.02 seconds. On average the execution time of cSP_flat is divided by two due to the use of an index.

	-cs6	-cs4	-cs2	-cs3	-cs1	-cs5	-cs7
-cs6		68%	80%	88%	88%	89%	90%
-cs4	32%		68%	80%	80%	90%	84%
-cs2	20%	32%		87%	89%	82%	90%
-cs3	12%	20%	13%		57%	63%	83%
-cs1	12%	20%	11%	43%		60%	76%
-cs5	11%	10%	18%	37%	40%		67%
-cs7	10%	16%	10%	17%	24%	33%	

Table 5 – cSP_flat: percentage of the random flat benchmark executed faster using the row option than using the column option

2.3 Comparison of cSP_flat with Kparam



(a) (b)

Figure 15 – time comparison of cSP_flat without and with an index vs Kparam_r on random flat benchmark

Figure 15 presents the execution time of cSP_flat without and with an index relative to Kparam_r (resp. Plot (15a) and Plot (15b)). A logarithmic scale is used on all axes for a better visibility on the fastest results. When comparing the two plots, it is clear that the cloud of dots is lower in the second one, illustrating the results of the previous paragraph. Globally, without an index, cSP_flat is faster than Kparam_r on 54% of the benchmark and with an index on 68% of them. In both plots, the uppermost diagonal line splits the diagram into two parts, above it are the tests for which cSP_flat is at least 10 times slower than Kparam_r. These formulæ represent only 9% of the results on Plot (15a) and 3% on Plot (15b). Below the other diagonal are the tests for which cSP_flat is 10 times faster than Kparam_r. These formulæ represent roughly 17% of the results on Plot (15a) and 36% on Plot (15b). These results are summarized in Table 6. This analysis allows us to conclude that cSP_flat is more efficient on the considered benchmark. Still depending on the chosen time limit, in case of

failure by timeout from `cSP_flat` it is reasonable to try to solve the problem using `Kparam_r` before extending the computation time or giving up [2].

	10 times slower	faster	10 times faster
`cSP_flat`	9%	54%	17%
`cSP_flat` with index	3%	68%	36%

Table 6 – Comparison of `cSP_flat` with/without index vs. `Kparam_r`

3 cSP

In this section, we analyze the performance of `cSP`, an implementation of the c$\mathcal{SP}$ calculus in $\mathbb{E}_1$ introduced Chapter 3, Section 3. In a first experiment we compare `cSP` with `cSP_flat` on the random flat benchmark to observe the impact of the functional term representation (`LogTk`) of `cSP` (see Chapter 8 Section 2.1 for more details about the differences between `cSP` and `cSP_flat`). Then we compare our tools to state-of-the-art solvers (described in Chapter 9, Section 1).

3.1 Impact of handling functional terms

Changing the core of `cSP_flat` into one able to handle functional terms is costly, as Figure 16 shows. On average, `cSP` is ten times slower than `cSP_flat` on the random flat benchmark. This overhead induced by functional terms is

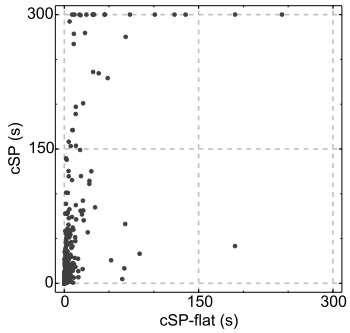


Figure 16 – time comparison of `cSP` vs `cSP_flat`, random flat benchmark

easily explainable: the new term representation (see Chapter 8, Section 2.1) must handle nested terms which prevent the use of a simple integer representation as

2. As future work, it could be interesting to develop a strategy running both `cSP_flat` and `Kparam_r` (successively or in parallel) since their strengths and weaknesses differ, making them most efficient on different kinds of formulæ.

is done in cSP_flat [3]. Moreover, the subsumption method is a bit more involved (see Chapter 7) which no doubts plays a role in slowing down cSP compared to cSP_flat.

3.2 Performance comparisons on random non-flat benchmark

The experiment presented here is a comparison of the prime implicate generation systems Zres and SOLAR presented Chapter 9 Section 1 with cSP_flat and cSP on the random non-flat benchmark described Section 2 of the same chapter. The input formulæ were flattened (see e.g. [11] for a definition) for cSP_flat and Zres, and the substitutivity axiom instantiated when necessary. Furthermore, for Zres, these flat equational formulæ were also converted to propositional ones, by instantiating the transitivity of equality when necessary.

	successes	SOLAR successes			Zres successes			(flat-)cSP			timeouts	
		time(s)	inf.	PIs	time(s)	inf.	PIs	time(s)	inf.	PIs	inf.	PIs?
SOLAR	15%	11.842	663190	506	X	X	X	X	X	X	2452908	28152
Zres	52%	0.695	X	2986	12.474	X	13804	X	X	X	X	X
cSP_flat	63%	6.622	5157	74	2.334	3300	158	14.290	11005	348	68959	X
cSP	76%	0.042	110	21	3.436	1322	47	10.193	1834	79	14714	538

Table 7 – Random non-flat benchmark - test results summary

The results are summarized in Table 7. Each line corresponds to a system. The column labeled 'successes' indicates the percentage of tests that were completed before the 5 minute timeout. The three columns under the label 'SOLAR successes' summarize average results on those tests on which SOLAR terminated before the timeout. The same goes for the columns under 'Zres successes' and '(flat-)cSP successes'. Finally, the 'timeout' columns expose the mean results on all interrupted tests. Columns labeled 'time', 'inf.' and 'PIs' respectively give the mean execution time, mean number of inferences and mean number of prime implicates found for each set of tests. The last column is labeled 'PIs?' because due to the timeout, the implicates found are not guaranteed to be prime. Cells labeled with an 'X' indicate that the corresponding data is not available.

As shown in the 'successes' column, cSP is the obvious winner in terms of the number of tests handled before timeout. It should also be mentioned that cSP solves all the problems that other systems solve, except for two that are solved only by Zres. The 15% of problems solved by SOLAR are the simplest of the random formulæ. The results show that SOLAR's approach is very costly both in terms of time and space, although methods to reduce these costs are being investigated [4]. The high number of prime implicates this tool generates

3. In fact, since cSP handles only closed terms, it should be possible to generate them (statically or dynamically), assign an integer to each and manipulate only the integers during the comparisons. However, the nested forms would still need to be used for the projections and new clause computation so the benefits are unclear.

4. Personal communication of Prof. Nabeshima

compared to those produced by cSP may seem surprising. In fact, SOLAR returns an over-approximation of the result because it does not take into account the equality axioms in its redundancy detection. Thus for example, any literal $t \simeq s$ also appears as $s \simeq t$ and $f(s) \simeq f(t)$ is not detected as redundant w.r.t. $s \simeq t$. Comparatively, the huge number of prime implicates generated by Zres is not surprising at all. It stems directly from the propositional translation of the initial problems and the introduction of new propositional variables. Although Zres is faster than cSP on the problems they both solve, it solves 52% of the problems, while cSP solves 76% of them. The results in the '(flat-)cSP' column are globally higher than those in the 'Zres successes' columns, because the most difficult formulæ are solved only by cSP and to a lesser extend by cSP_flat. Since cSP solves more problems than cSP_flat and does so faster and with fewer clauses processed, cSP is clearly better adapted to dealing with originally non-flat formulæ. Incidentally, note that the overhead of cSP's term handling compared to that of cSP_flat's (observed in the previous experiment Section 3.1) is more than compensated by the direct handling of non-flat formulæ since on the random non-flat benchmark cSP is faster than cSP_flat. The number of inferences and generated non-redundant implicates when the tool times out illustrates the heavy cost of the cSP inferences and redundancy detection mechanism compared to that of SOLAR. It is a price that seems partly unavoidable to eliminate all redundancies, since this requires complex algorithms.

3.3 (Non-)scalability

To have an idea of the scalability of cSP, we ran it on the QF-UF benchmark (see Chapter 9, Section 2), which contains larger formulæ than the other benchmarks. The aim of this experiment is not to compute all the prime implicates of a formula [5] but simply to see how many implicates can be computed before timeout or rather how many of them can be considered as new information, i.e. how many are not subsumed by a clause from the input formula. Table 8 exposes the results of this experiment. The eight formulæ of the QF-UF benchmark are identified by their size in bytes (first line). The following lines present respectively the number of clauses computed by cSP, the number of clauses selected in the waiting set, the number of clauses stored in the processed set at timeout and finally, among these, the ones not subsumed by an input clause. Clearly,

size of the formula (byte)	16K	20K	24K	32K	32K	56K	76K	112K
generated clauses	146169	86909	37901	39571	15141	16478	44592	49667
analyzed clauses	903	479	59	122	1223	105	36	17
potential prime implicates	84	29	3	1	2	1	5	4
non-redundant implicates	21	0	1	0	0	0	1	0

Table 8 – QF-UF benchmark - test results summary

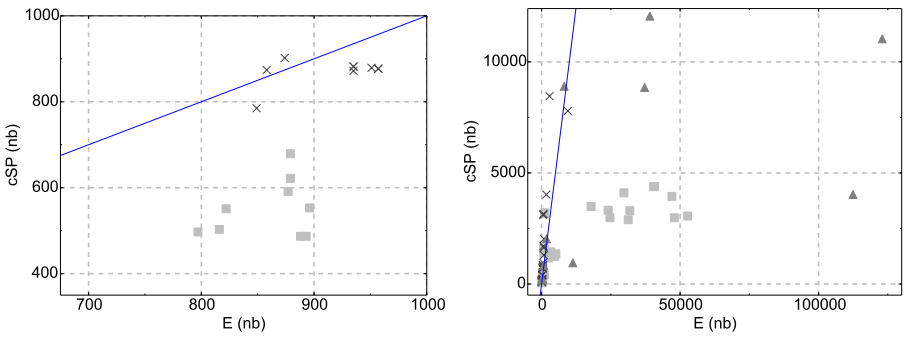
5. That would be extremely ambitious considering that these formulæ were made to push SMT solvers to their limits.

the results show that these formulæ are too complex for cSP to handle, attesting that our program does not scale well, as is the case of all prime implicate generation tools that we know of. To improve on the efficiency of cSP, apart from improving the existing mechanisms (e.g. the inference system, the indexation, etc.), an idea is to consider distributed strategies. For example, the recursive entailment tests could be executed in parallel, and so could the different rules of the c$\mathcal{SP}$ calculus. This direction has not been explored and remains as future work.

4 Tests with filters

4.1 Impact of the normalization

In this experiment, we estimated the impact of the normalization mechanism (see Chapter 1, Section 1.2) on the QF-AX benchmark. As mentioned in Chapter 9, Section 2 this benchmark was used to compare cSP with the E theorem prover. To do so, cSP was run with a filtering option (`-max-size 0`) that blocks the generation of any implicate beside the empty clause, effectively turning cSP into a superposition theorem prover. This way the main differences between cSP and E are the normalization of clauses and the redundancy pruning mechanism. On the one hand, the redundancy pruning algorithm used by cSP is weaker because it does not allow for equational simplification or other n-to-one redundancy pruning rules. On the other hand one-to-one redundancy testing is stronger since its uses logical entailment together with the usual ordering condition instead of subsumption. The comparison of cSP with the E theorem prover on these formulæ shows that the normalization approach can, in some nontrivial cases, reduce the number of processed clauses by an order of magnitude.



(a) storecomm formulæ (b) swap formulæ

Figure 17 – Comparison of the number of processed clauses for E and cSP.

Figure 17 presents the positive results of this experiment, i.e., the results of the storecomm and swap formulæ. Among these, only the formulæ on which

both E and cSP (without Assertion rules) terminate before timeout and without memory overflow were kept. Light gray squares represent the invalid formulæ, i.e. the satisfiable ones, while dark gray crosses mark the unsatisfiable ones. The line $y = x$ occurs in both plots. An interesting observation is that for the largest invalid formulæ, cSP needs to process a smaller number of clauses than E before terminating, even 10 times less in the case of the invalid_swap formulæ. The unsatisfiable swap formulæ were run with a timeout of 10 minutes (the triangles in Plot (17b)) and the corresponding results hint that this phenomenon could also be true for larger unsatisfiable problems. This suggests that the redundancy pruning technique based on normalization and clausal trees could be profitably integrated into state-of-the-art superposition-based theorem-provers, at least for ground equational clause sets. However, it might not always be useful, e.g. Figure 18, plotting the storeinv formulæ where an opposite tendency is observed, although the storeinv formulæ on which neither cSP or E timeout represent only a ten of that of the other two families for which the results are positive.

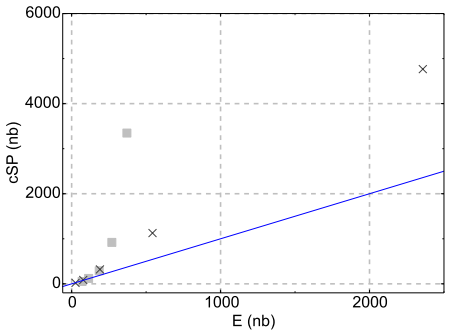


Figure 18 – storeinv formulæ - comparison of the number of processed clauses for E and cSP.

4.2 Prime implicates of size one

The tests with the filter on the prime implicates length less or equal to 1 highlight the limits of cSP, which always reaches the 5 minute timeout (except for one simple storeinv benchmark found unsatisfiable by cSP in less than one second).

As shown in Figure 19 on the storeinv formulæ of the QF-AX benchmark, the number of clauses that cSP can process in 5 minutes is highly dependent on the size of the input formulæ (the size of the array modeled), i.e. on the number of clauses and of different symbols it contains. This tendency is verified also with the other families of formulæ. The more complex the formula, the more generated clauses for each processed clause. With a 5 minute timeout, the system cannot generate more than approximately 20000 clauses, which seems rather small compared to the hundred million of inferences performed by SOLAR

(not included in Figure 19). This rift is also observed in the implicates recovered at timeout that are measured in tens for `cSP` and in hundreds for `SOLAR`. Putting aside the aforementioned reserve about the redundant nature of these implicates (see Section 3.2 of this chapter), hypotheses that could explain this difference are:

— the maturity of `SOLAR` compared to `cSP`,
— the light cost of the tableau calculus compared to our method where the many necessary comparisons and projections (see Chapter 7) may significantly slow down the computations on large formulæ.

A proper investigation of this phenomena is a subject of future work.

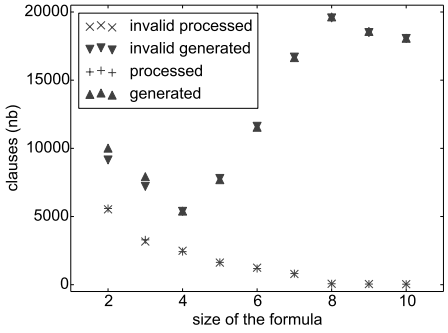


Figure 19 – Number of processed and generated clauses for `storeinv` formulæ on `cSP` with filter (size ≤ 1) after 5 minutes.

5 Summary

The different experiments presented in this chapter allow us to draw the following picture of our prime implicate generation algorithms:

1. It is always better to have a method tailored to the input formulæ rather than to preprocess them by flattening and/or conversion to propositional logic.
2. Our tools compare favorably with other state-of-the-art prime implicate generation softwares, but do not scale better than them.
3. The best option to use for each tool is:
   — for `Kparam`, the rewriting on the fly with the relaxed collision criterion (`-o5` option);
   — for `cSP_flat`, the lexicographic selection function that considers the clausal part of c-clauses first, plus the use of an index to speed up the calculus (`-csi` option);
   — for `cSP`, the same as for `cSP_flat` (implemented by default).

4. On flat formulæ, `cSP_flat` is the best choice over `Kparam` and `cSP`, while on non-flat ones `cSP` gives the best results as expected.
5. The normalization method used in our algorithms significantly reduces the number of generated clauses and thus could be profitably integrated in state-of-the-art equational theorem provers.

Conclusion

The work on prime implicate generation in ground equational logic exposed in this thesis comprises four main contributions:

- Two calculi that generate implicates are proposed.
  - The $\mathcal{K}$-paramodulation calculus, restricted to flat formulæ, is a relaxation of the unordered paramodulation calculus in which hypotheses are added directly to the generated clauses to allow for superposition inferences upon different constants.
  - The c$\mathcal{SP}$ calculus augments the classical superposition calculus (full with its ordering constraints) with two rules that allow the addition of conditions to the generated clauses. It can handle non-flat formulæ.
- A clause storage data structure is defined, the normal clausal tree. It significantly reduces the memory consumption of the algorithms and allows the efficient detection and handling of redundant clauses.
- Several prototypes of prime implicate generation tools implement the calculi, storage method and their variants.

The main results associated with these contributions are the following. On the theoretical side, we proved the deductive-completeness of both calculi and their variants and the termination and correctness of the clausal tree manipulation algorithms. On the practical side we realized an experimental evaluation of our prototypes. It show that they are on par with state-of-the-art prime implicate generation tools. Precisely, our best prototype in flat ground logic (`cSP_flat`) is faster than the reference tool (`Zres`) in 68% of the corresponding benchmark and in non-flat ground logic, the best prototype (`cSP`) is able to handle 60% more formulæ of the corresponding benchmark than the reference tool (`SOLAR`) before timeout. We also created a clause-handling method that in some cases, compared to the state-of-the-art `E` theorem prover, reduces significantly the number of clauses to manipulate to check the satisfiability of a formula. The limitations of our methods include their restriction to equational logic and the cost of our clause-handling algorithms that render them non-scalable as is.

In a short term perspective, this work leaves several questions unanswered that could lead to improvements of the performance of the algorithms. In Remark 3.12 page 81, a relaxation of c-subsumption, i.e. the definition of redundancy for constrained clauses, is suggested that would allow the detection of more redundancies than the current definition. However, its implementation is non-trivial and thus the gain in the search space might be mitigated by the

added cost to the redundancy detection algorithms of cSP. Another lead to explore is the inversion of clauses and constraints in constrained clausal trees, as is proposed in Remark 7.4 page 117. The simpler inclusion test of the constraints would be performed before the costly redundancy detection used for clauses, which could globally improve the execution time of the redundancy detection algorithms of cSP. As mentioned in Chapter 10 at the end of Section 3.3, the parallelization of some specific parts of our algorithms is also an option. For example, the application of the rules of the calculus on different clauses or the recursive manipulations of a clausal tree on different subtrees seem well-suited to parallel executions. Other directions in which this work can be pursued include the improvement of the filtering mechanisms of cSP so as to block the generation of undesired clauses instead of removing them once they are generated. On the experimental side, a replacement of Zres, a tool used as a reference in our experiments, with the more recent primer [6] should be considered if new experiments are performed. Finaly, an investigation of the cause behind the small number of clauses generated compared to SOLAR in the experiment presented in Section 3.3 of Chapter 10 could also provide insights on how to improve the cSP prototype.

In a longer term perspective, the most promising direction in which to pursue the work presented in this thesis would be to implement a version of cSP based on a state-of-the-art equational theorem prover such as the E theorem prover so as to benefit from all the clever improvements it contains. Once the scalability issue has been solved (or at least significantly improved) the extension of the c$\mathcal{SP}$ calculus to more expressive logics should be considered, e.g., from the simplest extension to the most complex, logics including boolean terms, many-sorted terms and variables. Once our algorithms are extended to handle variables, and a way to go around the semi-decidability of prime implicate generation, e.g. the systematic use of filters, has been devised, it could be interesting to compute the prime implicates of SMT formulæ, e.g. formulæ in Presburger arithmetic or the theory of arrays. Two approaches can be considered. One would simply consist in giving the theory along with the input formula. The axioms would be used in the calculus only with clauses from the input formula or their consequences, and never with each other. Any direct consequence of the axioms would be deleted from the set of implicates. The other approach applies to the best known theories only, like Presburger arithmetic, bit-vectors, array with extensivity, etc. It would consists in using an external SMT solver (e.g. Z3) to suggest simplifications for the terms of the considered theory (for example, in arithmetic, $x + 0$ can be simplified into x) and to assess the possible equivalence of these terms so as to know where and under what conditions to apply the rules of an extended c$\mathcal{SP}$ calculus.

6. Both tools are mentioned in Chapter i.

Appendices

Appendix A

Mini-sudoku formalization

For a better readability, the formalization of the mini-sudoku problem presented in the introduction is presented in the TPTP fof style that uses disjunctions '|' and conjunctions '&'. To shorten this appendix, some similar formulæ are ommited and replaced by '...'. The complete formalization of the problem is included with the archive downloadable at http://lig-membres.imag.fr/tourret/index.php?&tab=3&slt=tools.

```
% problem

fof(ax1,axiom,(ssA(n1,n1)=n1)).
fof(ax2,axiom,(ssA(n4,n1)=n2)).
fof(ax3,axiom,(ssA(n3,n2)=n1)).
fof(ax4,axiom,(ssA(n2,n3)=n2)).
fof(ax5,axiom,(ssA(n4,n3)=n1)).
fof(ax6,axiom,(ssA(n3,n1)=n4)).

% constraints

%----Lower cardinality bound
fof(ax1,axiom,
    ( n1 != n2
    & n1 != n3
    & n1 != n4
    & n2 != n3
    & n2 != n4
    & n3 != n4 )).

%----Row constraints
fof(ax2,axiom,
    ( ssA(n1,n1) = n1
    | ssA(n1,n2) = n1
```

```
    | ssA(n1,n3) = n1
    | ssA(n1,n4) = n1 )).

...

fof(ax5,axiom,
    ( ssA(n1,n1) = n4
    | ssA(n1,n2) = n4
    | ssA(n1,n3) = n4
    | ssA(n1,n4) = n4 )).

fof(ax11,axiom,
    ( ssA(n1,n1) != ssA(n1,n2)
    & ssA(n1,n1) != ssA(n1,n3)
    & ssA(n1,n1) != ssA(n1,n4)
    & ssA(n1,n2) != ssA(n1,n3)
    & ssA(n1,n2) != ssA(n1,n4)
    & ssA(n1,n3) != ssA(n1,n4) )).


fof(ax12,axiom,
    ( ssA(n2,n1) = n1
    | ssA(n2,n2) = n1
    | ssA(n2,n3) = n1
    | ssA(n2,n4) = n1 )).

...

fof(ax15,axiom,
    ( ssA(n2,n1) = n4
    | ssA(n2,n2) = n4
    | ssA(n2,n3) = n4
    | ssA(n2,n4) = n4 )).

fof(ax21,axiom,
    ( ssA(n2,n1) != ssA(n2,n2)
    & ssA(n2,n1) != ssA(n2,n3)
    & ssA(n2,n1) != ssA(n2,n4)
    & ssA(n2,n2) != ssA(n2,n3)
    & ssA(n2,n2) != ssA(n2,n4)
    & ssA(n2,n3) != ssA(n2,n4) )).

fof(ax22,axiom,
    ( ssA(n3,n1) = n1
    | ssA(n3,n2) = n1
    | ssA(n3,n3) = n1
```

```
    | ssA(n3,n4) = n1 )).

...

fof(ax25,axiom,
    ( ssA(n3,n1) = n4
    | ssA(n3,n2) = n4
    | ssA(n3,n3) = n4
    | ssA(n3,n4) = n4 )).

fof(ax31,axiom,
    ( ssA(n3,n1) != ssA(n3,n2)
    & ssA(n3,n1) != ssA(n3,n3)
    & ssA(n3,n1) != ssA(n3,n4)
    & ssA(n3,n2) != ssA(n3,n3)
    & ssA(n3,n2) != ssA(n3,n4)
    & ssA(n3,n3) != ssA(n3,n4) )).

fof(ax32,axiom,
    ( ssA(n4,n1) = n1
    | ssA(n4,n2) = n1
    | ssA(n4,n3) = n1
    | ssA(n4,n4) = n1 )).

...

fof(ax35,axiom,
    ( ssA(n4,n1) = n4
    | ssA(n4,n2) = n4
    | ssA(n4,n3) = n4
    | ssA(n4,n4) = n4 )).


fof(ax41,axiom,
    ( ssA(n4,n1) != ssA(n4,n2)
    & ssA(n4,n1) != ssA(n4,n3)
    & ssA(n4,n1) != ssA(n4,n4)
    & ssA(n4,n2) != ssA(n4,n3)
    & ssA(n4,n2) != ssA(n4,n4)
    & ssA(n4,n3) != ssA(n4,n4) )).

%----column constraints
fof(ax92,axiom,
    ( ssA(n1,n1) = n1
    | ssA(n2,n1) = n1
    | ssA(n3,n1) = n1
```

```
    | ssA(n4,n1) = n1 )).

...

fof(ax95,axiom,
    ( ssA(n1,n1) = n4
    | ssA(n2,n1) = n4
    | ssA(n3,n1) = n4
    | ssA(n4,n1) = n4 )).

fof(ax101,axiom,
    ( ssA(n1,n1) != ssA(n2,n1)
    & ssA(n1,n1) != ssA(n3,n1)
    & ssA(n1,n1) != ssA(n4,n1)
    & ssA(n2,n1) != ssA(n3,n1)
    & ssA(n2,n1) != ssA(n4,n1)
    & ssA(n3,n1) != ssA(n4,n1) )).

fof(ax102,axiom,
    ( ssA(n1,n2) = n1
    | ssA(n2,n2) = n1
    | ssA(n3,n2) = n1
    | ssA(n4,n2) = n1 )).

...

fof(ax105,axiom,
    ( ssA(n1,n2) = n4
    | ssA(n2,n2) = n4
    | ssA(n3,n2) = n4
    | ssA(n4,n2) = n4 )).

fof(ax111,axiom,
    ( ssA(n1,n2) != ssA(n2,n2)
    & ssA(n1,n2) != ssA(n3,n2)
    & ssA(n1,n2) != ssA(n4,n2)
    & ssA(n2,n2) != ssA(n3,n2)
    & ssA(n2,n2) != ssA(n4,n2)
    & ssA(n3,n2) != ssA(n4,n2) )).

fof(ax112,axiom,
    ( ssA(n1,n3) = n1
    | ssA(n2,n3) = n1
    | ssA(n3,n3) = n1
    | ssA(n4,n3) = n1 )).
```

```
...

fof(ax115,axiom,
    ( ssA(n1,n3) = n4
    | ssA(n2,n3) = n4
    | ssA(n3,n3) = n4
    | ssA(n4,n3) = n4 )).

fof(ax121,axiom,
    ( ssA(n1,n3) != ssA(n2,n3)
    & ssA(n1,n3) != ssA(n3,n3)
    & ssA(n1,n3) != ssA(n4,n3)
    & ssA(n2,n3) != ssA(n3,n3)
    & ssA(n2,n3) != ssA(n4,n3)
    & ssA(n3,n3) != ssA(n4,n3) )).

fof(ax122,axiom,
    ( ssA(n1,n4) = n1
    | ssA(n2,n4) = n1
    | ssA(n3,n4) = n1
    | ssA(n4,n4) = n1 )).

...

fof(ax125,axiom,
    ( ssA(n1,n4) = n4
    | ssA(n2,n4) = n4
    | ssA(n3,n4) = n4
    | ssA(n4,n4) = n4 )).

fof(ax131,axiom,
    ( ssA(n1,n4) != ssA(n2,n4)
    & ssA(n1,n4) != ssA(n3,n4)
    & ssA(n1,n4) != ssA(n4,n4)
    & ssA(n2,n4) != ssA(n3,n4)
    & ssA(n2,n4) != ssA(n4,n4)
    & ssA(n3,n4) != ssA(n4,n4) )).

%----Subsquare constraints
fof(ax182,axiom,
    ( ssA(n1,n1) = n1
    | ssA(n1,n2) = n1
    | ssA(n2,n1) = n1
    | ssA(n2,n2) = n1 )).

...
```

```
fof(ax185,axiom,
    ( ssA(n1,n1) = n4
    | ssA(n1,n2) = n4
    | ssA(n2,n1) = n4
    | ssA(n2,n2) = n4 )).

fof(ax191,axiom,
    ( ssA(n1,n1) != ssA(n1,n2)
    & ssA(n1,n1) != ssA(n2,n1)
    & ssA(n1,n1) != ssA(n2,n2)
    & ssA(n1,n2) != ssA(n2,n1)
    & ssA(n1,n2) != ssA(n2,n2)
    & ssA(n2,n1) != ssA(n2,n2) )).

fof(ax182,axiom,
    ( ssA(n1,n3) = n1
    | ssA(n1,n4) = n1
    | ssA(n2,n3) = n1
    | ssA(n2,n4) = n1 )).

...

fof(ax185,axiom,
    ( ssA(n1,n3) = n4
    | ssA(n1,n4) = n4
    | ssA(n2,n3) = n4
    | ssA(n2,n4) = n4 )).

fof(ax191,axiom,
    ( ssA(n1,n3) != ssA(n1,n4)
    & ssA(n1,n3) != ssA(n2,n3)
    & ssA(n1,n3) != ssA(n2,n4)
    & ssA(n1,n4) != ssA(n2,n3)
    & ssA(n1,n4) != ssA(n2,n4)
    & ssA(n2,n3) != ssA(n2,n4) )).

fof(ax182,axiom,
    ( ssA(n3,n1) = n1
    | ssA(n3,n2) = n1
    | ssA(n4,n1) = n1
    | ssA(n4,n2) = n1 )).

...

fof(ax185,axiom,
```

```
    ( ssA(n3,n1) = n4
    | ssA(n3,n2) = n4
    | ssA(n4,n1) = n4
    | ssA(n4,n2) = n4 )).

fof(ax191,axiom,
    ( ssA(n3,n1) != ssA(n3,n2)
    & ssA(n3,n1) != ssA(n4,n1)
    & ssA(n3,n1) != ssA(n4,n2)
    & ssA(n3,n2) != ssA(n4,n1)
    & ssA(n3,n2) != ssA(n4,n2)
    & ssA(n4,n1) != ssA(n4,n2) )).

fof(ax182,axiom,
    ( ssA(n3,n3) = n1
    | ssA(n3,n4) = n1
    | ssA(n4,n3) = n1
    | ssA(n4,n4) = n1 )).

...

fof(ax185,axiom,
    ( ssA(n3,n3) = n4
    | ssA(n3,n4) = n4
    | ssA(n4,n3) = n4
    | ssA(n4,n4) = n4 )).

fof(ax191,axiom,
    ( ssA(n3,n3) != ssA(n3,n4)
    & ssA(n3,n3) != ssA(n4,n3)
    & ssA(n3,n3) != ssA(n4,n4)
    & ssA(n3,n4) != ssA(n4,n3)
    & ssA(n3,n4) != ssA(n4,n4)
    & ssA(n4,n3) != ssA(n4,n4) )).

%----Codomain
fof(ax272,axiom,
    ( ssA(n1,n1) = n1
    | ssA(n1,n1) = n2
    | ssA(n1,n1) = n3
    | ssA(n1,n1) = n4 )).

...

fof(ax275,axiom,
    ( ssA(n1,n4) = n1
```

```
    | ssA(n1,n4) = n2
    | ssA(n1,n4) = n3
    | ssA(n1,n4) = n4 )).

fof(ax281,axiom,
    ( ssA(n2,n1) = n1
    | ssA(n2,n1) = n2
    | ssA(n2,n1) = n3
    | ssA(n2,n1) = n4 )).

...

fof(ax284,axiom,
    ( ssA(n2,n4) = n1
    | ssA(n2,n4) = n2
    | ssA(n2,n4) = n3
    | ssA(n2,n4) = n4 )).

fof(ax290,axiom,
    ( ssA(n3,n1) = n1
    | ssA(n3,n1) = n2
    | ssA(n3,n1) = n3
    | ssA(n3,n1) = n4 )).

...

fof(ax293,axiom,
    ( ssA(n3,n4) = n1
    | ssA(n3,n4) = n2
    | ssA(n3,n4) = n3
    | ssA(n3,n4) = n4 )).

fof(ax299,axiom,
    ( ssA(n4,n1) = n1
    | ssA(n4,n1) = n2
    | ssA(n4,n1) = n3
    | ssA(n4,n1) = n4 )).

...

fof(ax302,axiom,
    ( ssA(n4,n4) = n1
    | ssA(n4,n4) = n2
    | ssA(n4,n4) = n3
    | ssA(n4,n4) = n4 )).
```

```
%------------------------------------------------------------------
```

Appendix B

BNF syntax of the inputs

The BNF syntax below is extracted from the TPTP BNF syntax. This syntax is that of `Kparam` and `cSP`.The parts highlighted in red are specific to `cSP`. Original comments of the TPTP syntax are preceded by `%----` and comments specific to our syntax begin with `%----*`.

```
%----*Syntax for CNF on ground equational clauses for Kparam and cSP
%----*extracted from :
%----v5.4.0.0 (TPTP version.internal development number)
%--------------------------------------------------------------------------
%----README ... this header provides important meta- and usage information
%----
%----Intended uses of the various parts of the TPTP syntax are explained
%----in the TPTP technical manual, linked from www.tptp.org.
%----
%----Four kinds of separators are used, to indicate different types of rules:
%----  ::= is used for regular grammar rules, for syntactic parsing.
%----  :== is used for semantic grammar rules. These define specific values
%----      that make semantic sense when more general syntactic rules apply.
%----  ::- is used for rules that produce tokens.
%----  ::: is used for rules that define character classes used in the
%----       construction of tokens.
%----
%----White space may occur between any two tokens. White space is not specified
%----in the grammar, but there are some restrictions to ensure that the grammar
%----is compatible with standard Prolog: a <TPTP_file> should be readable with
%----read/1.
%----
%----The syntax of comments is defined by the <comment> rule. Comments may
%----occur between any two tokens, but do not act as white space. Comments
%----will normally be discarded at the lexical level, but may be processed
%----by systems that understand them (e.g., if the system comment convention
%----is followed).
%----
%--------------------------------------------------------------------------
```

```
%----Files. Empty file is OK.
<TPTP_file>          ::= <TPTP_input>*
<TPTP_input>         ::= <annotated_formula>

%----*Formula records (restricted to cnf, without anotations)
<annotated_formula>  ::= <cnf_annotated>
<cnf_annotated>      ::= cnf(<name>,<formula_role>,<cnf_formula><annotations>).
<annotations>        ::= <null>

%----Types for problems.
<formula_role>       ::= <lower_word>
<formula_role>       :== axiom | hypothesis | definition | assumption |
                         lemma | theorem | conjecture | negated_conjecture |
                         plain | unknown
%----*The different formula roles are kept for information but have no impact
%----*during execution
%--------------------------------------------------------------------------
%----CNF formulae (variables implicitly universally quantified)
%----*restricted to ground equational literals
<cnf_formula>        ::= (<disjunction>) | <disjunction>
<disjunction>        ::= <literal> | <disjunction> <vline> <literal>
<literal>            ::= <atomic_formula> | <fol_infix_unary>
%--------------------------------------------------------------------------
%----Special formulae
<fol_infix_unary>    ::= <term> <infix_inequality> <term>

%----First order atoms
<atomic_formula>     ::= <defined_plain_formula> | <defined_atomic_formula>
<defined_plain_formula> ::= <defined_plain_term>
<defined_plain_formula> :== <defined_prop>
<defined_prop>       :== <atomic_defined_word>
<defined_prop>       :== $false
%----Pure CNF should use $false only at the roots of a refutation.

<defined_atomic_formula> ::= <defined_infix_formula>
<defined_infix_formula>  ::= <term> <defined_infix_pred> <term>
<defined_infix_pred> ::= <infix_equality>
<infix_equality>     ::= =
<infix_inequality>   ::= !=

%----First order terms
<term>               ::= <function_term>
<function_term>      ::= <plain_term>
<plain_term>         ::= <constant> | <functor>(<arguments>)
<constant>           ::= <functor>
<functor>            ::= <atomic_word>
%----Defined terms have TPTP specific interpretations
<defined_plain_term> ::= <defined_constant>
<defined_constant>   ::= <defined_functor>
<defined_functor>    ::= <atomic_defined_word>
```

```
%----Arguments recurse back up to terms (this is the FOF world here)
<arguments>          ::= <term> | <term>,<arguments>

%------------------------------------------------------------------------------

%----General purpose
<name>               ::= <atomic_word> | <integer>
%----Integer names are expected to be unsigned
<atomic_word>        ::= <lower_word>
<atomic_defined_word> ::= <dollar_word>
<null>               ::=
%------------------------------------------------------------------------------
%----Rules from here on down are for defining tokens (terminal symbols) of the
%----grammar, assuming they will be recognized by a lexical scanner.
%----A ::- rule defines a token, a ::: rule defines a macro that is not a
%----token. Usual regexp notation is used. Single characters are always placed
%----in []s to disable any special meanings (for uniformity this is done to
%----all characters, not only those with special meanings).

%----These are tokens that appear in the syntax rules above. No rules
%----defined here because they appear explicitly in the syntax rules,
%----except that <vline>, <star>, <plus> denote "|", "*", "+", respectively.
%----Keywords:    fof cnf thf tff include
%----Punctuation: ( ) , . [ ] :
%----Operators:   ! ? ~ & | <=> => <= <~> ~| ~& * +
%----Predicates:  = != $true $false

%----For lex/yacc there cannot be spaces on either side of the | here
<comment>            ::- <comment_line>
<comment_line>       ::- [%]<printable_char>*

<dollar_word>        ::- <dollar><lower_word>
<lower_word>         ::- <lower_alpha><alpha_numeric>*

%----Tokens used in syntax, and cannot be character classes
<vline>              ::- [|]

%----Numbers. Signs are made part of the same token here.
<integer>            ::- (<signed_integer>|<unsigned_integer>)
<signed_integer>     ::- <sign><unsigned_integer>
<unsigned_integer>   ::- <decimal>
<decimal>            ::- (<zero_numeric>|<positive_decimal>)
<positive_decimal>   ::- <non_zero_numeric><numeric>*

%---Space and visible characters upto ~, except ' and  <sign>               ::: [+-]
<zero_numeric>       ::: [0]
<non_zero_numeric>   ::: [1-9]
<numeric>            ::: [0-9]
<lower_alpha>        ::: [a-z]
<upper_alpha>        ::: [A-Z]
```

```
<alpha_numeric>      ::: (<lower_alpha>|<upper_alpha>|<numeric>|[_])
<dollar>             ::: [$]
<printable_char>     ::: .
%----<printable_char> is any printable ASCII character, codes 32 (space) to 126
%----(tilde). <printable_char> does not include tabs, newlines, bells, etc. The
%----use of . does not not exclude tab, so this is a bit loose.
%-------------------------------------------------------------------------------
```

Bibliography

[1] Alessandro Armando, Maria Paola Bonacina, Silvio Ranise, and Stephan Schulz. New results on rewrite-based satisfiability procedures. *ACM Trans. Comput. Logic*, 10(1):1–51, 2009. (Cited on pages xiv and 138.)

[2] Franz Baader and Tobias Nipkow. *Term Rewriting and All That.* Cambridge University Press, 1998. (Cited on pages xiii, 29, 41 and 131.)

[3] L. Bachmair and H. Ganzinger. Rewrite-based Equational Theorem Proving with Selection and Simplification. *Journal of Logic and Computation*, 3(4):217–247, 1994. (Cited on pages 24, 31 and 32.)

[4] Leo Bachmair and Harald Ganzinger. Resolution Theorem Proving. In *Handbook of Automated Reasoning*, pages 19–99. 2001. (Cited on pages ii, iii, 8, 12 and 14.)

[5] Clark Barrett, Pascal Fontaine, and Cesare Tinelli. The SMT-LIB Standard: Version 2.5. Technical report, Department of Computer Science, The University of Iowa, 2015. Available at ttwww.SMT-LIB.org. (Cited on pages xiv and 138.)

[6] Clark Barrett, Roberto Sebastiani, Sanjit Seshia, and Cesare Tinelli. Satisfiability Modulo Theories. In Armin Biere, Marijn J. H. Heule, Hans van Maaren, and Toby Walsh, editors, *Handbook of Satisfiability*, volume 185 of *Frontiers in Artificial Intelligence and Applications*, pages 825–885. IOS Press, 2009. (Cited on page 26.)

[7] Dan Benanav. Simultaneous paramodulation. In Mark E. Stickel, editor, *10th International Conference on Automated Deduction*, number 449 in Lecture Notes in Computer Science, pages 442–455. Springer Berlin Heidelberg, July 1990. (Cited on page 48.)

[8] M. Bienvenu. Prime implicates and prime implicants in modal logic. In *Proceedings of the National Conference on Artificial Intelligence*, volume 22, page 379. Menlo Park, CA; Cambridge, MA; London; AAAI Press; MIT Press; 1999, 2007. (Cited on pages ii, 22 and 24.)

[9] G. Bittencourt. Combining syntax and semantics through prime form representation. *Journal of Logic and Computation*, 18(1):13, 2008. (Cited on pages ii and 20.)

[10] Patrick Blackburn, Johan Van Benthem, and Frank Wolter. *Handbook of Modal Logic*. Studies in logic and practical reasoning - ISSN 1570-2464 ; 3. Elsevier, 2007. (Cited on page 24.)

[11] Maria Paola Bonacina and Mnacho Echenim. Theory decision by decomposition. *Journal of Symbolic Computation*, 45(2):229–260, 2010. (Cited on pages xiii, 132 and 147.)

[12] D. Brand. Proving Theorems with the modification method. *SIAM Journal of Computing*, 4:412–430, 1975. (Cited on page 22.)

[13] Ricardo Caferra. *Logic for computer science and artificial intelligence*. John Wiley & Sons, 2013. (Cited on page 13.)

[14] Sylvain Conchon and Jean-Christophe Filliâtre. A Persistent Union-find Data Structure. In *Proceedings of the 2007 Workshop on Workshop on ML*, ML '07, pages 37–46. ACM, 2007. (Cited on pages xii and 128.)

[15] O. Coudert and JC Madre. A new method to compute prime and essential prime implicants of boolean functions. In *Proc. of MIT VLSI Conference*, 1992. (Cited on pages ii, 19 and 20.)

[16] Simon Cruanes. Logtk: A Logic ToolKit for Automated Reasoning and its Implementation. In *4th Workshop on Practical Aspects of Automated Reasoning.*, 2014. (Cited on pages xiii and 130.)

[17] J. De Kleer. An improved incremental algorithm for generating prime implicates. In *Proceedings of the National Conference on Artificial Intelligence*, pages 780–780. John Wiley & Sons ltd, 1992. (Cited on pages ii, 12, 15, 25, 90 and 91.)

[18] A. Degtyarev and A. Voronkov. Equality Elimination for the Tableau Method. In *Proceeding of DISCO-96 (Design and Implementation of Symbolic Computation Systems. International Symposium)*, pages 46–60. Springer LNCS 1128, 1996. (Cited on page 23.)

[19] Isil Dillig and Thomas Dillig. Explain: a tool for performing abductive inference. In *Computer Aided Verification*, pages 684–689. Springer, 2013. (Cited on pages 24 and 135.)

[20] Isil Dillig, Thomas Dillig, and Alex Aiken. Small Formulas for Large Programs: On-Line Constraint Simplification in Scalable Static Analysis. In Radhia Cousot and Matthieu Martel, editors, *Static Analysis*, number 6337 in Lecture Notes in Computer Science, pages 236–252. Springer Berlin Heidelberg, 2010. (Cited on page 87.)

[21] Isil Dillig, Thomas Dillig, and Alex Aiken. Automated Error Diagnosis Using Abductive Inference. In *Proceedings of the 33rd ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '12, pages 181–192, New York, NY, USA, 2012. ACM. (Cited on page 24.)

[22] Isil Dillig, Thomas Dillig, Kenneth L. McMillan, and Alex Aiken. Minimum Satisfying Assignments for SMT. In P. Madhusudan and Sanjit A. Seshia, editors, *Computer Aided Verification*, number 7358 in Lecture Notes in

Computer Science, pages 394–409. Springer, 2012. (Cited on pages ii, xiii, 22 and 24.)

[23] Mnacho Echenim and Nicolas Peltier. A Calculus for Generating Ground Explanations. In Bernhard Gramlich, Dale Miller, and Uli Sattler, editors, *Automated Reasoning*, number 7364 in Lecture Notes in Computer Science, pages 194–209. Springer Berlin Heidelberg, June 2012. (Cited on pages i, 9 and 11.)

[24] Mnacho Echenim, Nicolas Peltier, and Sophie Tourret. A Superposition Strategy for Abductive Reasoning in Ground Equational Logic. In Maria Paola Bonacina and Maribel Fern andez, editors, *IWS 2012 - International Workshop on Strategies in Rewriting, Proving and Programming (IJCAR 2012 workshop)*, pages 4–11, Manchester, United Kingdom, July 2012. http://www.dcs.kcl.ac.uk/staff/maribel/IWS2012/IWS2012.html. (Cited on pages i, 11 and 12.)

[25] Mnacho Echenim, Nicolas Peltier, and Sophie Tourret. An Approach to Abductive Reasoning in Equational Logic. In Francesca Rossi, editor, *IJCAI 2013 - International Joint Conference on Artificial Intelligence*, pages 531–537, Beijing, China, August 2013. AAAI Press. (Cited on pages i, 11, 12 and 24.)

[26] Mnacho Echenim, Nicolas Peltier, and Sophie Tourret. A Deductive-Complete Constrained Superposition Calculus for Ground Flat Equational Clauses. In *4th Workshop on Practical Aspects of Automated Reasoning.*, 2014. (Cited on pages i, 11 and 12.)

[27] Mnacho Echenim, Nicolas Peltier, and Sophie Tourret. A Rewriting Strategy to Generate Prime Implicates in Equational Logic. In Stéphane Demri, Deepak Kapur, and Christoph Weidenbach, editors, *Automated Reasoning*, Lecture Notes in Computer Science 8562, pages 137–151. Springer International Publishing, July 2014. (Cited on pages i, 11 and 12.)

[28] Mnacho Echenim, Nicolas Peltier, and Sophie Tourret. A Superposition-Based Approach to Abductive Reasoning in Equational Clausal Logic. In *ADDCT (IJCAR'14 workshop)*, page 1, 2014. (Cited on pages i and 11.)

[29] Mnacho Echenim, Nicolas Peltier, and Sophie Tourret. Quantifier-Free Equational Logic and Prime Implicate Generation. In *CADE-25*, pages 311–325. Springer, 2015. (Cited on pages i, 11 and 12.)

[30] B. Errico, F. Pirri, and C. Pizzuti. Finding Prime Implicants by Minimizing Integer Programming Problems. In *AI-CONFERENCE*, pages 355–362. World Scientific Publishing, 1995. (Cited on pages ii and 19.)

[31] Edward Fredkin. Trie memory. *Commun. ACM*, 3(9):490–499, 1960. (Cited on page 15.)

[32] R. Hähnle. Tableaux and Related Methods. In A. Robinson and A. Voronkov, editors, *Handbook of Automated Reasoning*, volume I, pages 100–178. Elsevier Science, 2001. (Cited on pages xiii and 22.)

[33] L. Henocque. The prime normal form of boolean formulas. *Technical report at http://www.Isis.org/fiche.php*, 2002. (Cited on pages ii and 20.)

[34] Thomas Hillenbrand, D. Topic, and Christoph Weidenbach. Sudokus as Logical Puzzles. In *Third Workshop on Disproving*, pages 2–12. W., Baumgartner, P., de Nivelle, H., 2006. (Cited on page 10.)

[35] Douglas R. Hofstadter. Gödel, Escher, Bach: an eternal golden braid. *Harmondsworth: Penguin, 1979*, 1, 1979. (Cited on page 8.)

[36] Katsumi Inoue. Consequence-finding Based on Ordered Linear Resolution. In *Proceedings of the 12th International Joint Conference on Artificial Intelligence - Volume 1*, IJCAI'91, pages 158–164, San Francisco, CA, USA, 1991. Morgan Kaufmann Publishers Inc. (Cited on pages ii and 22.)

[37] Koji Iwanuma, Hidetomo Nabeshima, and Katsumi Inoue. Toward an Efficient Equality Computation in Connection Tableaux: A Modification Method without Symmetry Transformation1—A Preliminary Report—. *First-order Theorem Proving*, page 19, 2009. (Cited on pages 22 and 136.)

[38] Said Jabbour, Joao Marques-Silva, Lakhdar Sais, and Yakoub Salhi. Enumerating Prime Implicants of Propositional Formulae in Conjunctive Normal Form. In *Logics in Artificial Intelligence*, pages 152–165. Springer, 2014. (Cited on pages ii and 21.)

[39] P. Jackson. Computing prime implicates incrementally. *Automated Deduction CADE-11*, pages 253–267, 1992. (Cited on pages ii and 15.)

[40] Peter Jackson and John Pais. Computing Prime Implicants. In Mark E. Stickel, editor, *10th International Conference on Automated Deduction, Kaiserslautern, FRG, July 24-27, 1990, Proceedings*, volume 449 of *Lecture Notes in Computer Science*, pages 543–557. Springer, 1990. (Cited on pages ii and 17.)

[41] A. Kean and G. Tsiknis. An incremental method for generating prime implicants/implicates. *Journal of Symbolic Computation*, 9(2):185–206, 1990. (Cited on pages ii and 15.)

[42] E. Knill, P. T. Cox, and T. Pietrzykowski. Equality and abductive residua for Horn clauses. *Theoretical Computer Science*, 120(1):1–44, November 1993. (Cited on pages 22 and 24.)

[43] V.M. Manquinho, A.L. Oliveira, and J. Marques-Silva. Models and Algorithms for Computing Minimum-Size Prime Implicants. In *Proceedings of the International Workshop on Boolean Problems*, 1998. (Cited on pages ii, 19 and 20.)

[44] J.P. Marques-Silva and K.A. Sakallah. GRASP: A search algorithm for propositional satisfiability. *Computers, IEEE Transactions on*, 48(5):506–521, 1999. (Cited on page 20.)

[45] Pierre Marquis. Extending abduction from propositional to first-order logic. In Philippe Jorrand and Jozef Kelemen, editors, *Fundamentals of Artificial Intelligence Research*, number 535 in Lecture Notes in Computer Science, pages 141–155. Springer, 1991. (Cited on pages 22 and 24.)

[46] Pierre Marquis. Consequence finding algorithms. In *Handbook of Defeasible Reasoning and Uncertainty Management Systems*, pages 41–145. Springer, 2000. (Cited on page 13.)

[47] A. Matusiewicz, N. Murray, and E. Rosenthal. Prime implicate tries. *Automated Reasoning with Analytic Tableaux and Related Methods*, pages 250–264, 2009. (Cited on pages ii, xiii, 20 and 135.)

[48] A. Matusiewicz, N. Murray, and E. Rosenthal. Tri-based set operations and selective computation of prime implicates. *Foundations of Intelligent Systems*, pages 203–213, 2011. (Cited on pages ii and 20.)

[49] Marta Cialdea Mayer and Fiora Pirri. First order abduction via tableau and sequent calculi. *Logic Journal of the IGPL*, 1(1):99–117, 1993. (Cited on pages 22 and 24.)

[50] Hidetomo Nabeshima, Koji Iwanuma, Katsumi Inoue, and Oliver Ray. SOLAR: An automated deduction system for consequence finding. *AI Communications*, 23(2):183–203, January 2010. (Cited on pages xiii and 22.)

[51] Greg Nelson and Derek C. Oppen. Fast Decision Procedures Based on Congruence Closure. *J. ACM*, 27(2):356–364, April 1980. (Cited on page 131.)

[52] T.H. Ngair. A new algorithm for incremental prime implicate generation. In *Proceedings of the 13th international joint conference on Artificial intelligence-Volume 1*, pages 46–51. Morgan Kaufmann Publishers Inc., 1993. (Cited on pages ii and 19.)

[53] R. Nieuwenhuis, A. Oliveras, and C. Tinelli. Solving SAT and SAT modulo theories: From an abstract Davis-Putnam-Logemann-Loveland procedure to DPLL (T). *Journal of the ACM*, 53(6):937–977, 2006. (Cited on page 20.)

[54] Robert Nieuwenhuis and Albert Rubio. Paramodulation-Based Theorem Proving. In *Handbook of Automated Reasoning*, pages 371–443. 2001. (Cited on pages 9 and 33.)

[55] Luigi Palopoli, Fiora Pirri, and Clara Pizzuti. Algorithms for selective enumeration of prime implicants. *Artificial Intelligence*, 111(1–2):41–72, July 1999. (Cited on pages ii and 21.)

[56] A. Previti, A. Ignatiev, A. Morgado, and J. Marques-Silva. Prime compilation of non-clausal formulae. In *Proceedings of the 24th International Conference on Artificial Intelligence*, pages 1980–1987. AAAI Press, 2015. (Cited on pages ii, xiii, 21, 22 and 135.)

[57] WV Quine. A way to simplify truth functions. *The American Mathematical Monthly*, 62(9):627–631, 1955. (Cited on pages ii and 14.)

[58] A. Ramesh, G. Becker, and N.V. Murray. CNF and DNF considered harmful for computing prime implicants/implicates. *Journal of Automated Reasoning*, 18(3):337–356, 1997. (Cited on pages ii and 19.)

[59] A.G. Ramesh. *Some applications of non clausal deduction.* PhD thesis, State University of New York at Albany, 1995. (Cited on pages ii and 19.)

[60] Manoj K. Raut. An Incremental Knowledge Compilation in First Order Logic. *arXiv:1110.6738 [cs]*, October 2011. (Cited on page 22.)

[61] Manoj K. Raut. An Incremental Algorithm for Computing Prime Implicates in Modal Logic. In *Theory and Applications of Models of Computation*, pages 188–202. Springer, 2014. (Cited on pages 22 and 24.)

[62] Manoj K. Raut and Arindama Singh. Prime implicants of first order formulas via transversal clauses. *International Journal of Computer Mathematics*, 81(2):157–167, 2004. (Cited on page 22.)

[63] Manoj K. Raut and Arindama Singh. Prime Implicates of First Order Formulas. *IJCSA*, 1(1):1–11, 2004. (Cited on page 22.)

[64] Manoj K. Raut and Arindama Singh. A survey on computing prime implicants and implicates in classical and non-classical logics. *COMPUTER SYSTEMS SCIENCE AND ENGINEERING*, 29(5):327–340, 2014. (Cited on page 14.)

[65] Alexandre Riazanov and Andrei Voronkov. Limited resource strategy in resolution theorem proving. *Journal of Symbolic Computation*, 36(1–2):101–115, July 2003. (Cited on page 34.)

[66] G. Robinson and L. Wos. Paramodulation and theorem-proving in first-order theories with equality. In D. Michie and R. Meltzer, editors, *Machine Intelligence*, volume 4, pages 135–150. Edinburg U. Press, 1969. (Cited on page 57.)

[67] R. Rymon. An se-tree-based prime implicant generation algorithm. *Annals of Mathematics and Artificial Intelligence*, 11(1):351–365, 1994. (Cited on pages ii and 17.)

[68] Stephan Schulz. E - a brainiac theorem prover. *AI Commun.*, 15(2-3):111–126, 2002. (Cited on pages iii and 34.)

[69] Stephan Schulz. System Description: E 1.8. In Ken McMillan, Aart Middeldorp, and Andrei Voronkov, editors, *Proc.of the 19th LPAR, Stellenbosch*, volume 8312 of *LNCS*. Springer, 2013. (Cited on pages xv and 138.)

[70] L. Simon and A. Del Val. Efficient consequence finding. In *International Joint Conference on Artificial Intelligence*, volume 17, pages 359–370. Lawrence Erlbaum Associates ltd, 2001. (Cited on pages ii, xiii and 16.)

[71] J.R. Slagle, C.L. Chang, and R.C.T. Lee. A new algorithm for generating prime implicants. *Computers, IEEE Transactions on*, 100(4):304–310, 1970. (Cited on pages ii and 17.)

[72] Viorica Sofronie-Stokkermans. Hierarchical Reasoning for the Verification of Parametric Systems. In *IJCAR*, pages 171–187, 2010. (Cited on page 24.)

[73] Viorica Sofronie-Stokkermans. Hierarchical Reasoning and Model Generation for the Verification of Parametric Hybrid Systems. In *Automated Deduction - CADE-24 - 24th International Conference on Automated Deduction, Lake Placid, NY, USA, June 9-14, 2013. Proceedings*, pages 360–376, 2013. (Cited on page 24.)

[74] T. Strzemecki. Polynomial-time algorithms for generation of prime implicants. *Journal of Complexity*, 8(1):37–63, 1992. (Cited on page 14.)

[75] G. Sutcliffe. The TPTP Problem Library and Associated Infrastructure: The FOF and CNF Parts, v3.5.0. *Journal of Automated Reasoning*, 43(4):337–362, 2009. (Cited on pages xii and 127.)

[76] Pierre Tison. Generalization of Consensus Theory and Application to the Minimization of Boolean Functions. *IEEE Transactions on Electronic Computers*, EC-16(4):446–456, 1967. (Cited on pages ii and 15.)

[77] Duc-Khanh Tran, Christophe Ringeissen, Silvio Ranise, and Hélène Kirchner. Combination of convex theories: Modularity, deduction completeness, and explanation. *Journal of Symbolic Computation*, 45(2):261–286, February 2010. (Cited on pages 22, 24 and 58.)

Index