



Master of Science in Informatics at Grenoble
Master Mathématiques Informatique - spécialité Informatique
option Artificial Intelligence and the Web

# Abduction and prime implicates, from propositional logic to equational logic

**Sophie Tourret**

Thursday, June 21st 2012

Research project performed at Grenoble Informatics Laboratory in the CAPP group

Under the supervision of:
Dr Mnacho Echenim (Grenoble InP/LIG)
Dr Nicolas Peltier (CNRS/LIG)

Defended before a jury composed of:
Dr Gilles Bisson
Prof. Noel De Palma
Dr Emmanuel Mazer
Prof. Marie-Christine Rousset

June 2012

# Contents

# List of figures



# List of algorithms

# Motivations

The verification of a complex system can often be reduced to testing the satisfiability of a logical formula encoding the properties of the system. If the system is error-free, then the associated formula is unsatisfiable, which can be detected by using an automated theorem prover in the considered logic, for instance a SAT-solver (e.g. zChaff, SATO, MathSAT...) or an SMT-solver (Yices, Z3, OpenSMT...) or a theorem prover for first order logic (E, Vampire, Prover9...). Consequently, if the formula is satisfiable, this means that there is an error somewhere in the system, and then the main objective of the designers of the system is to identify and correct this error. Usually, the detection of errors is done by generating models of the formula and by analysing them. However, this method possesses two major drawbacks: it can be difficult to distinguish the relevant information from the noise; and the identified error may correspond to a single case of a more general error that cannot be easily detected when considering only the models. Accordingly, a more efficient way to identify errors would be to directly find the most general explanations possible why the formula is satisfiable, without relying on models. This kind of reasoning is called *abduction*. It has been extensively studied in propositional logic because of its numerous applications in domains of artificial intelligence like planning [Shanahan, 1988] or truth-maintenance in knowledge bases [de Kleer and Reiter, 1987].

There are different ways to automate abduction. The most commonly used consists in reducing the original problem to one of consequence-finding, by negating the formula. Even though this idea has been exploited a lot in propositional logic, in more complex logics (that allow for the representation of more elaborate systems) there exists, to the best of our knowledge, very few similar approaches to abductive reasoning [Bienvenu, 2007; Mayer and Pirri, 1996]. In [Echenim and Peltier, 2012], a variant of the *superposition calculus* [Bachmair et al., 1994], was devised that is specifically tuned for generating ground consequences of sets of axioms in equational logic. More precisely, the generated explanations can be selected according to their relevance to the problem, by defining a set of interesting symbols, called *abductive constants*, that will be the only ones allowed to appear in the explanations. Given a set of first-order clauses $S$, this calculus is able to generate a set of ground flat clauses $T_\infty(S)$ built over abducible constants, that describes all possible consequences of $S$ that are of interest: if $C$ is an implicate of $S$ built over abducible constants, then $T_\infty(S)$ logically entails $C$. Thus, the formula $\neg T_\infty(S)$ can be viewed as a representation of the set of all

plausible hypotheses ensuring the unsatisfiability of $S$. Each of these hypotheses is a plausible explanation of an error in the original system, expressed as a logical formula that can be easily understood by the designers of the system.

However, returning the set $T_\infty(S)$ to the user is not fully satisfactory, since this set may be very large and contain a lot of redundant information. In practice, it would be more useful to return only the most general clauses that are logical consequences of $T_\infty(S)$, which correspond to the most economical explanations of $S$. Such clauses are the *prime implicates* of $T_\infty(S)$. In [Echenim and Peltier, 2012], the prime implicates of $T_\infty(S)$ are generated by applying the *resolution calculus* [Leitsch, 1997] to the set containing the clauses of $T_\infty(S)$ together with all the ground instances of the equality axioms. It is shown that every prime implicate of $T_\infty(S)$ can be obtained through this method. Yet, applying the obtained algorithm is very inefficient since the potentially huge set $T_\infty(S)$ becomes substantially bigger with the addition of the numerous instantiations of the equality axioms, which will themselves lead to the generation of a lot of potentially uninteresting implicates. For instance, a clause $a \not\simeq b$ will produce, together with the transitivity axiom $a \not\simeq c \vee c \not\simeq b \vee a \simeq b$, a clause $a \not\simeq c \vee c \not\simeq b$ for each constant symbol $c$ in the signature, even if there is no logical connection between $c$ and $a$ or $b$.

To solve this problem, we devise another calculus called $\mathcal{K}$-paramodulation[1], which is used to generate prime implicates more efficiently. This calculus can be viewed as a relaxed form of the superposition calculus, in which equalities are asserted instead of being proved. It is shown that this calculus permits to generate all prime implicates, which is not the case with the superposition calculus: for example, a prime implicate of the clauses $a \not\simeq b$ and $c \simeq d$ is $c \not\simeq a \vee d \not\simeq b$, but there is no way of generating this clause by superposition, even if the ordering conditions are relaxed. The advantage of this calculus is that it is much more restrictive than the resolution calculus. We also present an efficient way to store a set of equational clauses without redundancies thanks to a carefully designed data-structure, the $\mathcal{N}$-clausal tree. This data-structure and the manipulations associated answer the need to efficiently detect and get rid of redundancies during the computation of prime implicates. In addition, it offers a compact way to store potentially huge sets of implicates. By combining the $\mathcal{K}$-paramodulation calculus with the data structure, we obtain an algorithm able to compute the prime implicates of $T_\infty(S)$ (or any other flat ground equational set of clauses) in a new and efficient way.

This report begins by a chapter presenting a study of prime implicate computation in propositional logic, basis of our work in equational logic. In a second chapter, we introduce the data-structure storing sets of equational clauses, and an entailment test specific to ground equational logic that allows for the detection of redundancies. The third chapter is dedicated to the algorithms that manipulate the data structure and the proof of their correctness. The last chapter introduces the $\mathcal{K}$-paramodulation calculus and the proof that it can generate all the implicates of a formula in ground flat equational logic (up to redundancy).

[1] $\mathcal{K}$ stands for $\kappa o \acute{\iota} \mu \eta \sigma \iota \varsigma$ which means assumption in Greek

# Chapter 1

# State of the art for prime implicate generation in propositional logic

In this chapter, we explain the link between abduction and prime implicates before providing a summary and an analysis of the main algorithms that compute prime implicates in propositional logic.

## 1.1 Abduction and prime implicates in propositional logic

The terminology presented in the following section is mostly inspired from [Bittencourt, 2008]

### 1.1.1 Basic definitions

Let $\Sigma_p$ be a *propositional signature*, i.e. a set of *propositional symbols* or *atoms* or *variables*, denoted by $s, t, u...$ together with atoms representing the truth values True and False, which are respectively noted $\top$ and $\bot$. A *propositional logic language* $\mathcal{L}(\Sigma_p)$ is the set of all well-formed formulas constructed from $\Sigma_p$ and logical connectives $(\neg, \wedge, \vee, \Rightarrow, \Leftrightarrow)$ in the usual way. A *literal* $L$ is either an atom $x$ or its negation $\neg x$. The complementary literal of $L$, noted $\bar{L}$ is $\neg x$ if $L = x$ and $x$ if $L = \neg x$.

A *clause* $C$ is a *disjunction* of literals: $C = L_1 \vee \cdots \vee L_n$ and a *term*, or *dual clause*, is a *conjunction* of literals: $D = L_1 \wedge \cdots \wedge L_n$. The symbol $\square$ will denote an empty clause. Clauses and terms can also be seen as multisets of literals (the distinction will be clear from the context). If $C$ is a clause (resp. a term), then $\neg C$ is the term (resp. clause) such that $\neg C = \{\bar{L} | L \in C\}$. Let $C_1$ and $C_2$ be two clauses. It is said that $C_1$ *subsumes* $C_2$ when $C_1 \subseteq C_2$. A *logical*

*formula* $S$ from $\mathcal{L}(\Sigma_p)$ is in *conjunctive normal form* (CNF) if it is a conjunction of clauses: $S_{CNF} = C_1 \wedge \cdots \wedge C_m$. It is in *disjunctive normal form* (DNF) if it is a disjunction of terms: $S_{DNF} = D_1 \vee \cdots \vee D_m$. It is in *negational normal form* (NNF) when all negations are at the atomic level, e.g. $\neg(a \vee b)$ is not in NNF but $\neg a \wedge \neg b$ is.

A *propositional interpretation* is a function from $\Sigma_p$ to $\{\top, \bot\}$.For now, we will refer to propositional interpretations simply as interpretations. To extend the notion of interpretation to formulas, a formula is *evaluated* to $\top$ or $\bot$ in an interpretation $\mathcal{I}$ according to the usual rules:

**Definition 1** The following tables give evaluation rules for the complete system $\{\neg, \vee\}$. Transformation rules are given for the remaining logical symbols, so that all formulas can be evaluated:

- $\begin{array}{|c|}\hline \neg\bot \equiv \top \\ \hline \neg\top \equiv \bot \\ \hline \end{array}$
- $\begin{array}{|c|}\hline x \vee \bot \equiv x \\ \hline x \vee \top \equiv \top \\ \hline \end{array}$
- $x \wedge y \equiv \neg(\neg x \vee \neg y)$
- $x \Rightarrow y \equiv \neg x \vee y$
- $x \Leftrightarrow y \equiv \neg(\neg x \vee \neg y) \vee \neg(x \vee y)$ $\Diamond$

A *model* of a formula $S$ is an interpretation $\mathcal{I}$, such that $S$ is evaluated to $\top$ in $\mathcal{I}$. A *tautology* and a *contradiction* are formulas for which respectively all interpretations are models and no interpretation is a model. A formula is *satisfiable* if it possesses at least one model. Let $S_1$ and $S_2$ be two formulas. We say that $S_1$ *entails* $S_2$ when every model of $S_1$ is a model of $S_2$. If $C_1$ and $C_2$ are non-tautological clauses, then the notions of entailment and subsumption between $C_1$ and $C_2$ are indistinguishable (in propositional logic only). The statement "$S_1$ entails $S_2$" is noted $S_1 \models_0 S_2$, or, in this chapter where there is no ambiguity, simply $S_1 \models S_2$. $S_1$ and $S_2$ are equivalent, noted $S_1 \equiv S_2$, if $S_1 \models S_2$ and $S_2 \models S_1$. Algorithms exist for transforming any formula $S$ into equivalent formulas $S_{CNF}$ and $S_{DNF}$ in CNF and DNF respectively.

**Definition 2** 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 $\forall L \in C$, $S \not\models C - \{L\}$. $\Diamond$

**Definition 3** A term $D$ is an *implicant* of a formula $S$ iff $D \models S$ and $D$ is not a contradiction. It is a *prime implicant* of $S$ iff $D$ is an implicant of $S$ and for every implicate $D'$ of $S$, if $D \models D'$ then $D' \models D$. An equivalent definition is $D \models S$ and $\forall L \in D$, $D - \{L\} \not\models S$. $\Diamond$

**Proposition 4** *A clause $C$ is a prime implicate of $S$ iff $\neg C$ is a prime implicant of $\neg S$.*

These two notions are dual, as the previous proposition shows, and in this way, any algorithm computing prime implicants can be used to compute prime implicates. From this point on, we will only consider prime implicates, even if the algorithms presented were originally designed for prime implicants.

### 1.1.2 Abduction using prime implicates

Before relating abduction to prime implicates, a formal definition is needed. The notions and results of this paragraph are adapted from [Marquis, 1991]. In what follows, a theory is a (finite) set of formulas.

**Definition 5** Let $T$ be a theory, and $S$ be a formula. The formula $H$ is an *abductive explanation* of $S$ with respect to $T$ if $T \wedge H \models S$. $\Diamond$

Of course, not all explanations are relevant, for example any formula $H$ that is unsatisfiable can be viewed as an explanation of $S$ w.r.t. $T$. The properties that make an explanation interesting are defined below:

**Definition 6** Let $T$ be a theory, $S$ be a formula, and $H$ an abductive explanation of $S$ w.r.t. $T$.

- $H$ is *consistent* if $T \wedge H$ is satisfiable.
- $H$ is *minimal* if for all $H'$ abductive explanations of $S$ w.r.t. $T$ such that $H \models H'$, we have also $H' \models H$.

As of now, we will only consider abductive explanations that are both consistent and minimal if they exist. $\Diamond$

What links abduction to prime implicate computation is the following result:

**Theorem 7** *Let $T$ be a theory, and $S$ be a formula. A term $H$ is an abductive explanation of $S$ iff the clause $\neg H$ is a prime implicate of $T \wedge \neg S$.*

Proof. Since $H$ is an abductive explanation of $S$, we have $T \wedge H \models S$ which is equivalent to $T \wedge \neg S \models \neg H$. Moreover $H$ is a minimal explanation, so $\neg H$ is a prime implicate of $T \wedge \neg S$. ∎

This result shows that abduction can be done in a deductive way, by finding all the consequences of the formula containing $T$ and $\neg S$ and by negating those consequences.

## 1.2 Algorithms for prime implicate computation

This section contains a brief survey of existing algorithms for generating prime implicates of propositional formula. We do not formally define every algorithm, but we provide some insights on how they work and discuss the main ideas underlying them, emphasising their common points and differences. We use some simple but interesting examples to illustrate them.

### 1.2.1 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 prime implicates was first introduced in [Quine, 1955]. 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 formulas, 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. For example, 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 [Strzemecki, 1992]), they were never efficient enough to be used in real-life problems.

At the beginning of the seventies, methods directly using the formulas began to appear. They can be roughly divided into two classes: the resolution-based algorithms and the algorithms based on decomposition. In the following of this section some of those algorithms are presented.

### 1.2.2 Resolution-based algorithms

The *resolution calculus* is an inference method made of one rule, called resolution rule:

**Definition 8** Let $C_1 = L \vee \cdots \vee L \vee \alpha$ and $C_2 = \bar{L} \vee \cdots \vee \bar{L} \vee \beta$ be two clauses. 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*. $\Diamond$

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.

When the procedure leaves the resulting formula unchanged, the remaining set of formulas is exactly the set of prime implicates of the original formula. This method is introduced for the first time in [Tison, 1967] and an incremental technique called IPIA inspired from it is presented in [Kean and Tsiknis, 1990]. IPIA and Tison's method define an order on the literals to limit the number of redundant resolution steps, as illustrated in Figure 1.1 (taken from [De Kleer, 1992]). 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. It 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.

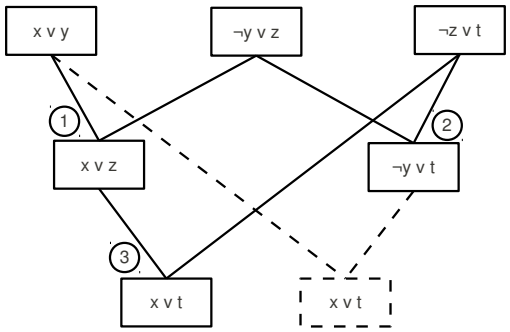



Figure 1.1: Example of redundant resolution avoided by Tison's method

Tison's method was also adapted in [Jackson, 1992] 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 privileges resolution 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 [De Kleer, 1992]. In addition to IPIA's optimisations on the resolution step, CLTMS optimises the subsumption step by using a new data structure to represent the prime implicates generated. This data structure is called a trie or discrimination tree [Fredkin, 1960]. 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:

- Its edges are literals and its leaves are clauses.

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

Figure 1.2 is an example of trie:

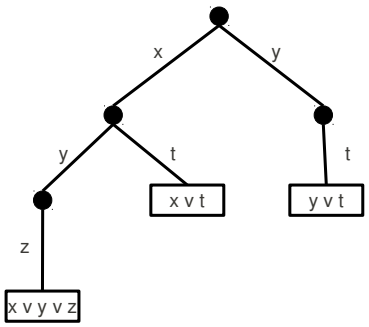



Figure 1.2: Trie representation of the set of clauses $\{x \vee y \vee z; x \vee t; y \vee t\}$ with order $x > y > z > t$

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. This algorithm 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 will be presented here comes from [Simon and Del Val, 2001]. Zres-tison is an algorithm based on a compact representation of formulas called ZBDD (Zero-suppressed Binary Decision Diagram) illustrated in Figure 1.3. 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 labelled by 1. On Figure 1.3, the dashed arrows mean that the literal labelling the parent node is ignored 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.

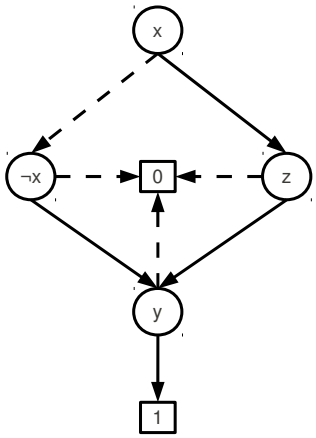



Figure 1.3: ZBDD of $S_{CNF} = (x \vee y \vee z) \wedge (\neg x \vee y)$

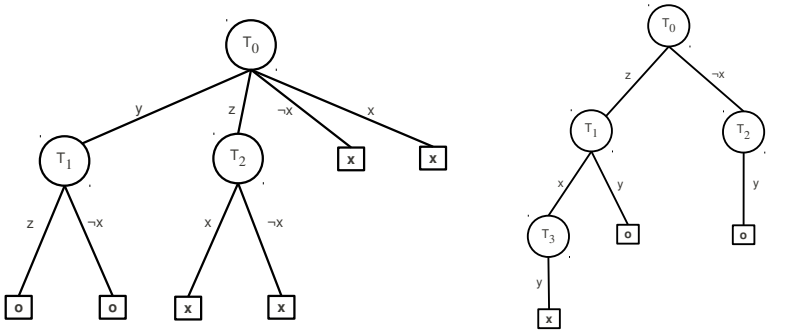
### 1.2.3 Algorithms based on decomposition

The common point of all the algorithms that do not use an inference rule is that they build the prime implicates of formulas by recursively decomposing them into smaller pieces (be it literals, terms or clauses) and merging the results obtained on those pieces. The oldest method of this kind (to the best of our knowledge) was introduced in [Slagle et al., 1970] 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, a leaf being called *failure node* if the term $\top$ belongs to the formula 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 generalisation of the notion of semantic trees (called Set Enumeration trees) is presented in [Rymon, 1994].

In [Jackson and Pais, 1990] TM is compared with a new algorithm of the same type called MM (for Matrix Method). 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 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 9** 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 $S_{CNF}$ previously used to describe minterms at the beginning of Section 1.2.1. 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 1.4. For both graphs, success nodes are the squares containing a 'o' and failure nodes are those with a '×' inside. For TM (Figure 1.4a), the partial order of the literal is based on their number of occurrences, so here we have $y > z, \neg x > x$ and we arbitrarily decide that $z > \neg x$. Starting from $T_0 = \{x \wedge y; \neg x \wedge y; \neg x \wedge z; y \wedge z\}$, deleting all the clauses containing $y$ allows to generate $T_1 = \{\neg x \wedge z\}$ and doing the same with clauses 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 clauses the node is a success node and if a clause is empty after removing literals, the node is a failure node. For MM (Figure 1.4b), the terms themselves are ordered. Here the order used is $\neg x \wedge z > x \wedge y > \neg x \wedge y > y \wedge z$. The extension of paths is done clause by clause in this order, and subsumption checks are done 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 1.4: TM and MM output for $S_{DNF}$

All previous methods need the input formulas to be in CNF or DNF. This can prove to be a major inconvenience for some families of formulas (see [Ngair, 1993; Ramesh et al., 1997]). Several techniques were developed to overcome this problem. In [Ngair, 1993], the incremental algorithm GEN-PI (for GENeration of Prime Implicates) is introduced. It takes a conjunction of DNF formulas 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. Indeed, 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 formulas and a lot more powerful for those being difficult to express in normal form. Another method developed with the same objective is the path-dissolution approach of [Ramesh, 1995; Ramesh et al., 1997]. This extension of MM to NNF formulas has nevertheless 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 [Coudert and Madre, 1992], the algorithm computing prime implicates is in fact building a BDD representing them, using a meta-product representation of the variables of the formula. 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 clause 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 subcases:

- 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).

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 [Manquinho et al., 1998]. This method orders the variables with regard to their number of occurrences in the formula.

In [Manquinho et al., 1998] Coudert and Madre's algorithm is compared with a generally more efficient technique that uses integer linear programming (ILP). This method is introduced in [Errico et al., 1995]. 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 formulas by converting them into ILP problems as shown in the example.

**Example 10** Let us consider $S_{DNF}$ as defined in Example 9. 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 than 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 $\{y_1, z_2\}$ are evaluated to 1 and all others are evaluated to 0. ♣

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 [Manquinho et al., 1998], the search algorithm used is one that is inspired from an improvement of the DPLL algorithm called GRASP [Nieuwenhuis et al., 2006; Marques-Silva and Sakallah, 1999].

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 [Coudert and Madre, 1992]. One of them, in [Henocque, 2002], 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 formulas having more than $10^{20}$ prime implicates. Another recent algorithm based on the same principle but accepting CNF inputs comes from [Matusiewicz et al., 2009], 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 [Matusiewicz et al., 2011] allows it to run with a polynomial memory-space occupation. Experiments have shown that this technique is at least twice faster than those based on resolution.

The last technique presented here is based on [Bittencourt, 2008]. 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 clause it belongs to as seen in the following example:

**Example 11** Lets number the terms of $S_{DNF}$.

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 $\{y^{\{1,2,4\}}, z^{\{3,4\}}\}$.

## 1.3 Summary and conclusions

After presenting the link between abduction and prime implicates, we have seen different algorithms computing these prime implicates in propositional logic (PL), grouped in two families: the resolution-based methods and the decomposition-based methods. In each family, efficient algorithms have been found.

One of the strong points of the decomposition methods is that they can be applied to all kind of formulas (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 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, and this loss can jeopardise criteria that are critical to the smooth running of the algorithms (like termination). Thus, such an extension must be done very carefully.

The subject of the following chapters is generating prime implicates in the context of flat ground equational logic through the adaptation of one of the algorithms presented before. It is the step we chose to come closer to our goal of extending abduction to equational first order logic. Solving this step is enough to answer the needs of performance improvement originating from [Echenim and Peltier, 2012], and it is also a good stepping stone to further extensions of abductive problems involving more complex theories like for example arithmetic.

# Chapter 2

# Reasoning in equational logic

The notions introduced in this chapter and the following ones are an adaptation to flat ground equational logic (EL) of the CLTMS algorithm presented in Section 1.2.2. We chose the algorithm to adapt among resolution-based methods only, because these methods seemed to us to be the most natural to extend into EL. CLTMS was selected because its efficiency and clarity make it a good target for extensions. Another algorithm could have been selected for its efficiency is Zres-Tison, but considering the time limit of a master internship, only one could be studied. Furthermore, Zres-Tison uses a very specific representation of the clause sets that does not directly extend to the equational case, thus in hopes of a more direct adaptation of the resolution calculus, we favoured CLTMS over Zres-Tison.

In this chapter, we first give informal ideas about the work that has to be done to go from PL to EL, and we present the formal notions derived from those ideas.

## 2.1 An informal overview of our contribution

We denote by $\mathbb{E}$ the set of all ground flat clauses in equational logic, which is an extension of PL to equality. Thus when working in $\mathbb{E}$, the literals are equations and disequations and the properties entailed by the equality (denoted by $\simeq$) must be taken into account. These properties are:

- reflexivity ($\forall x,\ x \simeq x$),
- commutativity ($\forall x, y,\ x \simeq y \Leftrightarrow y \simeq x$),
- transitivity ($\forall x, y, z,\ x \simeq y \wedge y \simeq z \Rightarrow x \simeq z$).

To deal with reflexivity, it is enough to replace any literal of the form $a \simeq a$ by $\top$ and any literal of the form $a \not\simeq a$ by $\bot$. Commutativity can be dealt with rather

easily by identifying the atoms $a \simeq b$ and $b \simeq a$ or by establishing an order on constants such that (for example), if we write $a \simeq b$, then it is implied that $a \succeq b$. Both methods are equivalent. The main problem arises from handling the transitivity axiom. For instance $A = a \not\simeq b \vee a \not\simeq c$, $B = a \not\simeq b \vee b \not\simeq c$ and $E = a \not\simeq c \vee b \not\simeq c$ are equivalent. Considering explicitly all these variants leads to a combinatorial explosion and should obviously be avoided in practice. Also, with transitivity, subsumption is no longer equivalent to logical entailment. For instance, $C = a \not\simeq b$ does not subsume $D = a \not\simeq c \vee b \not\simeq c$, but it is clear that $C \models D$. Hence, any subsumption test in CLTMS must be replaced by an entailment test in the algorithm adapted to EL.

To deal with these problems, we define a *normal form* for clauses, so that equivalent clauses (such as $A$, $A'$ and $B$ above) have the same normal form. We also devise a *projection* of one clause in normal form onto another to test logical entailment efficiently, and define an extension of tries (a data-structure commonly used to represent strings [Fredkin, 1960]), called $\mathcal{N}$*-clausal trees*, to optimise storage and entailment tests. As in CLTMS, we focus on the storage and manipulation of the set of implicates generated. There are only three actions to perform (once a new implicate $C$ has been generated):

1. Check if an implicate already found entails $C$; if there is, discard $C$; if not
2. remove all the implicates entailed by $C$ from the set,
3. finally, insert $C$ into the set of implicates.

The last point is not much of a challenge, so we focus, in Chapter 3, on the first two points and respectively design the algorithms isEntailed and pruneEntailed that perform these actions efficiently.

Another major problem when going from PL to EL is that the resolution calculus, used in PL to generate all the implicates of a formula, is unable to do the same in EL without any adaptation. For example, from the formula $S = \{a \simeq b, b \simeq c\}$, the resolution calculus can generate nothing, but by transitivity $a \simeq c$ is an implicate of $S$. A first solution, proposed in [Echenim and Peltier, 2012], is to instantiate the equality axioms and use resolution on the enriched formula. However, as was explained in the introduction, this method leaves room for improvement, especially in terms of efficiency and the number of useless generated clauses. In Chapter 4, we consider a different solution to this problem, namely, replacing resolution by a new calculus inspired from the superposition calculus, the $\mathcal{K}$*-paramodulation calculus*.

The remainder of this chapter is the formalisation of the aforementioned notions as well as others necessary to prove the correctness of our algorithms and calculus.

## 2.2 Clauses in equational logic

Let $\Sigma$ be a *signature*, i.e., in this setting, a set of constants denoted by $a, b, c \ldots$. We assume given a total order $\prec$ on constants. A *literal*, usually denoted by $l$,

is either an equation (or *atom*) $a \simeq b$, or an inequation $a \not\simeq b$ and when written $a \bowtie b$ it can denote either the equation or the inequation between $a$ and $b$. The literal $\bar{l}$ denotes $a \not\simeq b$ (resp. $a \simeq b$) when $l$ denotes $a \simeq b$ (resp. $a \not\simeq b$). A literal of the form $a \not\simeq a$ is called a contradictory literal (or a contradiction) and a literal of the form $a \simeq a$ is a tautological literal (or a tautology).

**Remark 12** *In* all *examples (and only in the examples), the constants will be ordered in the alphabetical order ($a \prec b \ldots$).*

We consider clauses in $\mathbb{E}$ as disjunctions (or multisets) of literals. If $C$ is a clause in $\mathbb{E}$ and $l$ a literal, $C \backslash l$ denotes the clause $C$ where *all* the occurrences of $l$ have been removed and $\neg C$ denotes the conjunction of the literals $\bar{l}$ for $l \in C$. We first define a normal form for clauses, based on the ordering $\prec$ on constant symbols. This form allows us to handle clauses modulo equivalence: two equivalent clauses are reduced to the same syntactic object.

**Definition 13** We define an *equational interpretation* $\mathcal{I}$ as a partition of $\Sigma$. Given two constants $a$ and $b$, we write $a =_{\mathcal{I}} b$ if $a$ and $b$ belong to the same class in $\mathcal{I}$. A positive literal $l = a \simeq b$ is evaluated to $\top$ in $\mathcal{I}$, written $\mathcal{I} \models_{\mathcal{E}} l$, if $a =_{\mathcal{I}} b$; otherwise $l$ is evaluated to $\bot$. A negative literal $l = a \not\simeq b$ is evaluated to $\top$ in $\mathcal{I}$ if $a \neq_{\mathcal{I}} b$, and to $\bot$ otherwise. This evaluation is extended to clauses and sets of clauses in the usual way. A *model* $\mathcal{M}$ of a clause $C$ is an equational interpretation in which $C$ is evaluated to $\top$. A *tautological clause* (or *tautology*) is a clause for which all equational interpretations are models and a *contradiction* is a clause that has no model. $\Diamond$

For technical reasons, we also need to consider interpretations of the clause sets that do not satisfy the equality axioms. Propositional interpretations are extended to formulas as in propositional logic. As in propositional logic, *propositional interpretations* are functions that associate a truth value to equations (and consequently to inequations). We write $\mathcal{I} \models_0 S$ if $\mathcal{I}$ is a propositional interpretation validating $S$ and $S_1 \models_0 S_2$ if every propositional interpretation validating $S_1$ also validates $S_2$. A propositional interpretation is also an equational interpretation iff the axioms of equality are satisfied. For example, by transitivity, if $\mathcal{I} \models_{\mathcal{E}} a \simeq b$ and $\mathcal{I} \models_{\mathcal{E}} a \simeq c$, then $\mathcal{I}$ is an equational interpretation iff[1] $\mathcal{I} \models_{\mathcal{E}} b \simeq c$. From this point on, we will use the term *interpretation* to refer to equational interpretations, and the propositional interpretations will be explicitly referred to as such. In the same way, the symbol $\models$ will now refer to the equational entailment $\models_{\mathcal{E}}$ and not to the propositional entailment $\models_0$.

**Definition 14** Let $C$ be a clause in $\mathbb{E}$, we define for any constant $a$ the *$C$-equivalence class* of $a$ as:

$$[a]_C \quad = \quad \{b \in \Sigma \mid a \not\simeq b \models C\}\,.$$

[1] Note that a propositional interpretation necessarily respects reflexivity ($a \simeq a$ is a tautology) and commutativity ($a \simeq b$ and $b \simeq a$ denote the same literal)

The *C-representatives* of a constant $a$, a literal $l$ and a clause $D$ are respectively defined as:

$$\begin{aligned} a_{\lfloor C} &= \min_{\prec}([a]_C) \\ l_{\lfloor C} &= a_{\lfloor C} \bowtie b_{\lfloor C}, \text{ for } l = a \bowtie b \\ D_{\lfloor C} &= \{l_{\lfloor C} \mid l \in D\} \end{aligned} \qquad \Diamond$$

If the considered clause $C$ is non-tautological, then it is easy to check that the notion of $C$-representative only depends on the set of negative literals occurring in $C$.

**Proposition 15** *The following properties hold:*

1. *Two non-tautological clauses containing the same negative literals generate the same equivalence classes.*
2. *If a non-tautological clause $C$ contains no negative literal, then for any constant $a$, $[a]_C = \{a\}$.*

**Example 16** Let us consider the clause $C = d \not\simeq a \vee d \not\simeq c \vee e \not\simeq b \vee e \simeq a \vee b \simeq c$. Then we have $[c]_C = \{a, c, d\}$, so $a_{\lfloor C} = d_{\lfloor C} = c_{\lfloor C} = a$. Moreover $(e \simeq a)_{\lfloor C} = (b \simeq c)_{\lfloor C} = b \simeq a$ and for $D = d \simeq c \vee e \simeq f$, $D_{\lfloor C} = a \simeq a \vee b \simeq f$. ♣

However this property does not hold for tautological clauses. The reason is that if $C$ is a tautological clause, then any negative literal $u \not\simeq v$ is such that $u \not\simeq v \models C$. Thus all the constants are in the same $C$-equivalence class, regardless of the negative literals in $C$. This explains why in the following properties and definitions, we are only concerned with non-tautological clauses. No loss of generality is entailed, since in practice, tautological clauses will be automatically removed by redundancy detection, as we will see later. To single out a tautological clause $C$, it suffice to compute $C_{\lfloor C}$. Since all the constants are $C$-equivalent and $C$ contains at least one positive literal $l$, the literal $l_{\lfloor C}$ has to be of the form $a \simeq a$ in the clause $C_{\lfloor C}$, which makes it easily recognisable as a tautological clause(notice that a clause containing no positive literal cannot be tautological because it is falsified by any interpretation in which all constant symbols are in the same class).

**Proposition 17** *Let $a$ be a constant, $l$ be a literal and $C$ and $D$ be two clauses in $\mathbb{E}$, then:*

$$\begin{aligned} \neg C &\models a \simeq a_{\lfloor C}, \\ \neg C &\models l \Leftrightarrow l_{\lfloor C}, \\ \neg C &\models D \Leftrightarrow D_{\lfloor C}. \end{aligned}$$

These two propositions (as well as most of the following ones) emphasise properties that will be useful for proving the correctness of the forthcoming algorithms. But before presenting these algorithms, we need to introduce more theoretical notions like the *normal form* or *projection*, which are used afterwards.

**Definition 18** The total order $<$ on literals is defined as follows:

- the equations are all greater than the inequations;
- for inequations and equations separately, the order $\prec$ on constants is used by first comparing the greatest constants of the (in)equalities with each other, then (if the greatest constants are identical) their other constants.$\Diamond$

**Example 19** The literals $c \simeq b$, $c \not\simeq b$, $b \not\simeq a$ and $c \simeq a$ are ordered in the following way:

$$b \not\simeq a < c \not\simeq b < c \simeq a < c \simeq b$$ ♣

This order allows us to introduce a normal form for clauses, which will be used to handle one of the problems generated by the transitivity axiom.

**Definition 20** A clause $C \in \mathbb{E}$ is in $<$-*normal form* (or simply in *normal form*) if:

1. every literal $a \not\simeq b$ it contains is such that $b = a_{\downarrow C}$;
2. every literal $a \simeq b$ it contains is such that $a = a_{\downarrow C}$ and $b = b_{\downarrow C}$;
3. it contains no literal of the form $a \not\simeq a$ or $a \simeq a$;
4. and the literals it contains occur exactly once in it.

A clause $C \in \mathbb{E}$ is in $<$-*relaxed normal form* (or *relaxed normal form*) if all the literals it contains satisfy conditions 1 and 2 above, and all negative literals satisfy conditions 3 and 4. $\Diamond$

Intuitively, a clause in relaxed normal form may contain tautologies and multiple occurrences of the same literal. This relaxed normal form is introduced because our algorithms use transformations on clauses that preserve the relaxed normal form but not the normal form.

Immediate consequences of this definition are the following propositions:

**Proposition 21** *The only unsatisfiable clause in relaxed normal form is* $\Box$ *and a tautological clause cannot be in normal form.*

**Proposition 22** *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 $b = a_{\downarrow C}$ by definition of a clause in relaxed normal form. But it means that $l$ is a contradiction, which is impossible in a relaxed normal clause. Now assume that there exists an $l \in C$ such that $l$ is of the form $a \not\simeq b$ with $a \succ b$. By definition $b = a_{\downarrow C}$, thus $a \succ a_{\downarrow C}$.■

**Remark 23** *In the following section, we will prove that two equivalent clauses have the same normal form.*

## 2.3 Theoretical basis: projection and consequences

This section introduces the operation of projection that permits to test logical entailment. This projection test provides a purely syntactic (and efficient) criterion allowing one to check whether an implication $C \models D$ holds.

In Proposition 26 and Theorem 27, we work on clauses built over literals involving equivalence classes (in addition to the normal clauses), because it allows for clearer proofs. These literals are defined as follows:

**Definition 24** Given two clauses $C$ and $D$ in $\mathbb{E}$, we denote by $[D]_C$ the clause built over the set of propositional symbols $\{[a]_C \mid a \in \Sigma\}$ and defined by $[D]_C = \{[a]_C \bowtie [b]_C \mid a \bowtie b \in D\}$. $\Diamond$

**Remark 25** *If $C = d \not\simeq a \vee d \not\simeq c \vee e \not\simeq b \vee e \simeq a \vee b \simeq c$ and $D = d \simeq c \vee e \simeq f$ then $[D]_C = \{a, d, c\} \simeq \{a, c, d\} \vee \{e, b\} \simeq \{f\}$. But this can also be written as $[D]_C = [a]_C \simeq [a]_C \vee [b]_C \simeq [f]_C$. The symbols $[a]_C$, $[b]_C$, and $[f]_C$ can be seen as the representatives of their respective classes. However, these classes already have their own representatives, namely $a_{\downarrow C}(= a)$, $b_{\downarrow C}(= b)$ and $f_{\downarrow C}(= f)$. Thus, even though we use the sets for technical reasons in our demonstrations, our results also hold with the sets representatives.*

**Proposition 26** *Assume $C$ is a non-tautological clause in $\mathbb{E}$, and $[a]_C \simeq [b]_C \notin [C]_C$. Then $C \vee a \not\simeq b$ is not a tautology.*

Proof. Let $D = C \vee a \not\simeq b$ and $\mathcal{I}$ be the interpretation constituted of all the $C$-equivalence classes on $\Sigma$ (i.e. $c =_{\mathcal{I}} d$ iff $c \not\simeq d \models C$). We define $\mathcal{J} = \mathcal{I} \setminus \{[a]_C, [b]_C\} \cup \{[a]_C \cup [b]_C\}$. It is clear that $\mathcal{J}$ is an interpretation (i.e. a partition of the constant symbols). Note that in particular, if $a =_{\mathcal{I}} b$, then $\mathcal{I} = \mathcal{J}$. $\mathcal{J} \not\models a \not\simeq b$ since $a =_{\mathcal{J}} b$ and for any negative literal $c \not\simeq d \in C$, by definition $c =_{\mathcal{I}} d$, thus $c =_{\mathcal{J}} d$ and $\mathcal{J} \not\models c \not\simeq d$. For the positive literals of $C$, several cases must be examined. Let $c \simeq d \in C$,

- if $c =_{\mathcal{J}} a$ and $d =_{\mathcal{J}} a$, then $\mathcal{J} \models c \simeq d$, but, as we will see, this case never occurs:
  - if $c =_{\mathcal{I}} a$ and $d =_{\mathcal{I}} a$, then by definition $c \not\simeq a \models C$ and $d \not\simeq a \models C$, hence $c \not\simeq d \models C$, but $c \simeq d \models C$ so $C$ is a tautology which contradicts our hypothesis;
  - if $c =_{\mathcal{I}} a$ and $d =_{\mathcal{I}} b$, then $c \in [a]_C$ and $d \in [b]_C$, hence $[a]_C \simeq [b]_C \in [C]_C$ which also contradicts our hypothesis;
  - the other cases are symmetrical to the ones already developed;
- if $c =_{\mathcal{J}} a$ and $d \neq_{\mathcal{J}} a$ then $c \neq_{\mathcal{J}} d$, thus $\mathcal{J} \not\models c \simeq d$ (the same holds when $c \neq_{\mathcal{J}} a$ and $d =_{\mathcal{J}} a$);
- if $c \neq_{\mathcal{J}} a$ and $d \neq_{\mathcal{J}} a$, then $c \neq_{\mathcal{I}} d$ since $\mathcal{I} \not\models c \simeq d$. Indeed we would otherwise have $c \not\simeq d \models C$ hence $C$ would be a tautology. Thus by definition of $\mathcal{J}$, $c \neq_{\mathcal{J}} d$ and $\mathcal{J} \not\models c \simeq d$. ∎

**Theorem 27** *Let $C$ and $D$ be two non-tautological clauses in $\mathbb{E}$. $D \models C$ iff every negative literal in $[D]_C$ is a contradiction and every positive literal in $[D]_C$ is also in $[C]_C$.*

Proof. First assume that $D \models C$. Consider a negative literal $a \not\simeq b \in D$, since $D \models C$, we must have $b \in [a]_C$, i.e., $[a]_C = [b]_C$, and the corresponding literal in $[D]_C$ is a contradiction. Now consider a positive literal $a \simeq b \in D$ and assume $[a]_C \simeq [b]_C \notin [C]_C$. Then $C \vee a \not\simeq b$ is not a tautology by Proposition 26. But $D \vee a \not\simeq b \models C \vee a \not\simeq b$, and $D \vee a \not\simeq b$ is a tautology, which yields a contradiction.

For the converse implication, we prove that every literal in $D$ entails $C$. Let $a \not\simeq b \in D$, then by hypothesis we have $[a]_C = [b]_C$, and by definition $a \not\simeq b \models C$. Let $a \simeq b \in D$, then by hypothesis $[a]_C \simeq [b]_C \in [C]_C$. By construction there is a literal $c \simeq d \in C$ such that $c \in [a]_C$ and $d \in [b]_C$. By definition $\neg C \models a \simeq c \wedge b \simeq d$, hence $\{a \simeq b\} \cup \neg C \models c \simeq d$. But $c \not\simeq d \in \neg C$ therefore $\{a \simeq b\} \cup \neg C$ is unsatisfiable. Thus $a \simeq b \models C$. ∎

By replacing equivalence classes by their representatives, we obtain the following corollary:

**Corollary 28** *Let $C$ and $D$ be two non-tautological clauses in $\mathbb{E}$. $D \models C$ iff every negative literal in $D_{\downarrow C}$ is a contradiction and every positive literal in $D_{\downarrow C}$ is also in $C_{\downarrow C}$.*

**Example 29** Let $C = a \not\simeq b \vee b \not\simeq c \vee e \simeq a \vee b \simeq b$ and $D = a \not\simeq c \vee e \simeq c$. As in the other examples, the constants are ordered alphabetically. By projection, we have $D_{\downarrow C} = a \not\simeq a \vee e \simeq a$, because $a_{\downarrow C} = \{a, b, c\}$. The literal $a \not\simeq a$ is a contradiction and $e \simeq a = e \simeq a_{\downarrow C} \in C_{\downarrow C}$, thus $D \models C$. Conversely, when projecting $C$ on $D$, the result is $C_{\downarrow D} = a \not\simeq b \vee b \not\simeq a \vee e \simeq a \vee f \simeq b$, thus $C \not\models D$. Indeed, $a \not\simeq b$ is not a contradiction, and $f \simeq b \neq e \simeq a = e \simeq b_{\downarrow D}$. ♣

**Remark 30** *From now on in this document, testing the projection of a clause $D$ on a clause $C$ will refer to the act of verifying whether every negative literal in $D_{\downarrow C}$ is a contradiction and every positive literal in $D_{\downarrow C}$ is also in $C_{\downarrow C}$.*

The main idea behind the algorithms presented in Sections 3.1.1 and 3.2.1 that test whether a clause $D$ entails a clause $C$ is based on Corollary 28. Informally, it consists in testing the projection of $D$ on $C$ by using the representatives of $C$ to rewrite $D$. The projection succeeds when the $C$-equivalence classes of $D$ are subsets of the $C$-equivalence classes of $C$ ("every negative literal in $D_{\downarrow C}$ is a contradiction") and the equalities in $D$ are $C$-equivalent to those in $C$ ("every positive literal in $D_{\downarrow C}$ is also in $C_{\downarrow C}$").

Corollary 28 is also useful in the proof of the uniqueness of the normal form of a clause, which also relies on the following lemma:

**Lemma 31** *Let $C$ and $D$ be two non-tautological clauses in normal form. If $C \neq D$ then $C \not\equiv D$.*

PROOF. Assume that $C \neq D$ and $C \equiv D$. Let $l = a \not\simeq b \in C$ where $a \succ b$, and assume that $l \notin D$. Then $b = a_{\downarrow C}$, and $b \neq a_{\downarrow D}$ since $a \not\simeq b$ does not occur in $D$ (and $D$ is in normal form), hence $[a]_C \neq [a]_D$. But since $C \equiv D$, we have $[a]_C = [a]_D$ for every $a$, which raises a contradiction. By symmetry, we can assume that all negative literals of $D$ occur in $C$, thus the negative part of $C$ is equal to that of $D$. By Proposition 15(1), we deduce that $[a]_C = [a]_D$, for every constant symbol $a$. Now let $l = a \simeq b \in C$, and assume that $l \notin D$. Since $C$ is in normal form and $C \equiv D$, we have $l = l_{\downarrow C} = l_{\downarrow D}$. By hypothesis $l \models C$, hence $l \models D$ and by Corollary 28 $l_{\downarrow D} \in D_{\downarrow D}$. Since $D$ is in normal form, we conclude that $l \in D$ which contradicts our hypothesis. The case where $D$ contains a literal not occurring in $C$ is symmetric. ∎

**Theorem 32** *For any non-tautological clause $C$ in $\mathbb{E}$, there exists a unique clause $C_\downarrow$ in normal form equivalent to $C$.*

PROOF. Let $C$ be a clause. We show that the set (not a multiset) $C_\downarrow = \{a \not\simeq a_{\downarrow C} \mid a \text{ occurs in } C \wedge a \neq a_{\downarrow C}\} \cup \{a_{\downarrow C} \simeq b_{\downarrow C} \mid a \simeq b \in C\}$ is the unique clause in normal form that is equivalent to $C$. If $\neg C$ and $C_\downarrow$ are both true in a given interpretation $\mathcal{I}$, then at least one literal $l \in C_\downarrow$ must be true in $\mathcal{I}$. If $l$ is a negative literal $l = a \not\simeq a_{\downarrow C}$, then by Proposition 17, $a \simeq a_{\downarrow C}$ must also be true in $\mathcal{I}$ since $\neg C \models a \simeq a_{\downarrow C}$, a contradiction. If $l = a_{\downarrow C} \simeq b_{\downarrow C}$ with $a \simeq b \in C$, then by Proposition 17, since $\neg C \models a \simeq b \Leftrightarrow a_{\downarrow C} \simeq b_{\downarrow C}$, necessarily $a \simeq b$ is also true in $\mathcal{I}$, and this interpretation cannot satisfy $\neg C$. Thus $C_\downarrow \models C$. Now assume $\neg C_\downarrow$ and $C$ are true in an interpretation $\mathcal{I}$' and let $l \in C$ be a literal evaluated to true in $\mathcal{I}$'. If $l$ is a negative literal $a \not\simeq b$, then $a_{\downarrow C} = b_{\downarrow C}$ and $\neg C_\downarrow \models a \simeq a_{\downarrow C} \wedge b \simeq b_{\downarrow C}$ by definition, so $a \simeq b$ is true in $\mathcal{I}$' and this is impossible. If $l$ is a positive literal $a \simeq b$, then $a_{\downarrow C} \simeq b_{\downarrow C}$ is also true in $\mathcal{I}$', since $a \simeq a_{\downarrow C}$ and $b \simeq b_{\downarrow C}$ are both true in $\mathcal{I}$', which is in contradiction with $\neg C_\downarrow$. Thus $C \equiv C_\downarrow$ and $C_\downarrow$ is unique by Lemma 31. ∎

**Example 33** Let $C_1 = b \not\simeq a \vee c \not\simeq b \vee e \not\simeq d \vee d \simeq b \vee e \simeq a$ and $C_2 = c \not\simeq a \vee c \not\simeq b \vee e \not\simeq d \vee e \simeq c \vee e \simeq b$. These two clauses have the same normal form: $C_{1\downarrow} = C_{2\downarrow} = b \not\simeq a \vee c \not\simeq a \vee e \not\simeq d \vee a \simeq d$, thus they are equivalent. ♣

The construction provided in the proof of this theorem has given us an easy way to compute the normal form of any clause when it exists. This will be useful in the implementation of our algorithms.

**Example 34** Let us consider, as in Example 16 the clause $C = d \not\simeq a \vee d \not\simeq c \vee e \not\simeq b \vee e \simeq a \vee c \simeq b$. Then $C_\downarrow = d \not\simeq a \vee c \not\simeq a \vee e \not\simeq b \vee b \simeq a$ is the normal form of $C$, and $C_\downarrow \vee b \simeq a$ is a relaxed normal form of $C$. ♣

**Notation 35** From this point on, we will refer to the set of ground flat equational clauses in normal form as $\mathbb{E}_\downarrow$. Note that this set contains no tautological clause. ◊

## 2.4 Data structures to represent sets of clauses

We introduce the notion of a clausal tree as a tree that permits a compact representation of a set of clauses in $\mathbb{E}$. The edges of a clausal tree are labelled with the literals of the represented clauses, and each leaf represents the clause composed of the literals appearing in the path from the root to this leaf. A clausal tree also allows an efficient detection of logical entailment between a clause and the clausal tree. The formal definition follows.

**Definition 36** A *clausal tree* is either $\square$, or a set of pairs $(l, T')$, with $l$ a literal and $T'$ a clausal tree. The set of clauses represented by a clausal tree $T$ is denoted by $\mathcal{C}(T)$ and defined inductively as follows:

- $\mathcal{C}(T) = \{\square\}$ if $T = \square$,
- $\mathcal{C}(T) = \bigcup_{(l,T')\in T} \left( \bigcup_{D\in\mathcal{C}(T')} l \vee D \right)$ otherwise. $\Diamond$

**Notation 37** Let $T$ be a clausal tree distinct from $\square$. $\mathcal{B}_r(T)$ denotes the set of literals $\mathcal{B}_r(T) = \{l \,|\, (l, T') \in T\}$ and $\mathcal{B}(T)$ is the set of literals inductively defined by $\mathcal{B}(\square) = \emptyset$ and $\mathcal{B}(T) = \mathcal{B}_r(T) \cup \bigcup_{(l,T')\in T} \mathcal{B}(T')$ otherwise. $\Diamond$

**Remark 38** *A pair $(l, \emptyset)$ represents no clause at all. Such pairs can be viewed as* failure nodes*, and can always be discarded from clausal trees. We assume that no such pair appears in our clausal trees, except when they are explicitly introduced.*

**Example 39** Let $T$ be the clausal tree in Figure 2.1.

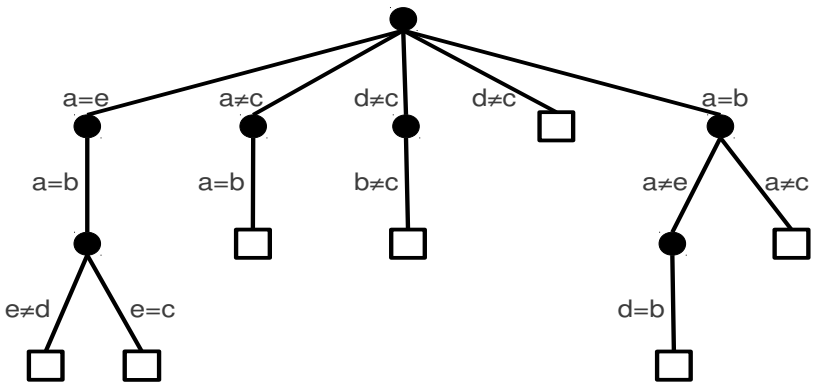



Figure 2.1: A clausal tree

Then $\mathcal{C}(T) = \{a \simeq e \vee a \simeq b \vee d \not\simeq e, e \simeq a \vee a \simeq b \vee c \simeq e, a \not\simeq c \vee a \simeq b, d \not\simeq c \vee b \not\simeq c, d \not\simeq c, a \simeq b \vee a \not\simeq e \vee d \simeq b, a \simeq b \vee a \not\simeq c\}$. This data structure is not really storage-efficient. For example, nothing prevents a clause to be represented twice, like the clause $a \not\simeq c \vee a \simeq b$. An other problem is that redundant clauses can be represented, like $d \not\simeq c \vee b \not\simeq c$ which is subsumed by $d \not\simeq c$. Fortunately, these problems can be sidestepped by adding constraints to the definition of clausal trees, which will be done later in this document. ♣

**Notation 40** Let $l$ be a literal and $T$ be a clausal tree. We denote by $T|_l^{\uparrow}$ the clausal tree such that $T|_l^{\uparrow} = \{(l, T)\}$. ◊

The following property holds:

**Proposition 41** *If $T$ is a clausal tree not reduced to $\Box$, then:*

$$\mathcal{C}(T) = \bigcup_{(l,T') \in T} \mathcal{C}(T'|_l^{\uparrow})$$

Similarly to the clauses in (relaxed) normal form that we defined previously, we introduce two categories of clausal trees whose aim it will be to represent such clauses.

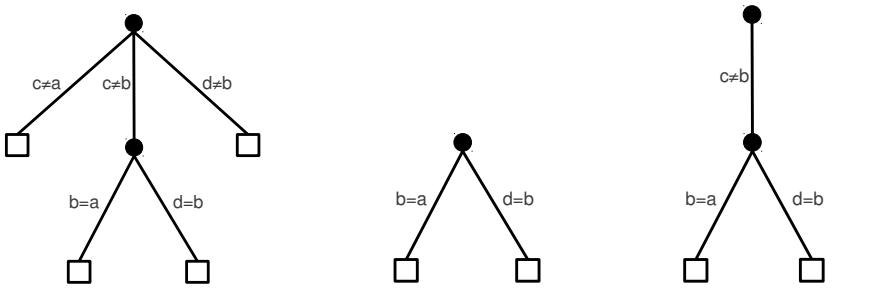
**Definition 42** A *relaxed normal clausal tree* (or *$\mathcal{RN}$-clausal tree*) $T$ is a clausal tree such that:

- for any pair $(l, T') \in T$, if $l$ is a negative literal then:
  - $l$ is not of the form $a \not\simeq a$;
  - $\forall l' \in \mathcal{B}(T')$, $l < l'$;
  - if $l = a \not\simeq b$, with $a \succ b$, then for all $l' \in \mathcal{B}(T')$, the constant $a$ does not occur in $l'$;
- for any pair $(l, T') \in T$, $T'$ is an $\mathcal{RN}$-clausal tree. ◊

**Definition 43** A *normal clausal tree* (or *$\mathcal{N}$-clausal tree*) $T$ is an $\mathcal{RN}$-clausal tree such that:

- for any pair $(l, T') \in T$, there is no $T'' \neq T'$ such that $(l, T'') \in T$;
- for any pair $(l, T') \in T$, if $l$ is a positive literal then:
  - $l$ is not of the form $a \simeq a$;
  - for all $l' \in \mathcal{B}(T')$, we have $l < l'$;
- for any pair $(l, T') \in T$, $T'$ is an $\mathcal{N}$-clausal tree. ◊

**Example 44** Figure 2.2 illustrates the different notations defined previously on an $\mathcal{N}$-clausal tree. Figure 2.2a represents the $\mathcal{N}$-clausal tree $T$ such that $\mathcal{C}(T) = \{c \not\simeq a,\ c \not\simeq b \vee b \simeq a,\ c \not\simeq b \vee d \simeq b,\ d \not\simeq b\}$. In this $\mathcal{N}$-clausal tree, $\mathcal{B}(T) = \{c \not\simeq a,\ c \not\simeq b,\ b \simeq a,\ d \simeq b,\ d \not\simeq b\}$ and $\mathcal{B}_r(T) = \{c \not\simeq a,\ c \not\simeq b,\ d \not\simeq b\}$. If $l = c \not\simeq b$, Figure 2.2b represents $T'$ a subtree of $T$ and Figure 2.2c represents $T'|_l^{\uparrow}$ a tree included in $T$. ♣




(a) $\mathcal{N}$-clausal tree $T$ (b) $\mathcal{N}$-clausal tree $T'$ (c) $\mathcal{N}$-clausal tree $T'|_l^{\uparrow}$

Figure 2.2: $\mathcal{N}$-clausal trees

**Proposition 45** *If $T$ is an $\mathcal{N}$-clausal tree (resp. an $\mathcal{RN}$-clausal tree) then $\mathcal{C}(T)$ is a set of clauses in normal form (resp. a set of clauses in relaxed normal form).*

As with clauses in relaxed normal form, $\mathcal{RN}$-clausal trees are introduced only to be used inside algorithms. Outside the algorithms, the implicates, being represented by clauses in normal form, will naturally be stored in $\mathcal{N}$-clausal trees. It is straightforward to use Definition 42 to verify the following result:

**Proposition 46** *The two following properties hold.*

- *Let $(l, T)$ be an $\mathcal{RN}$-clausal tree. If $l$ is a positive literal, then all the literals of $\mathcal{B}(T)$ are also positive.*
- *If $T_1, \ldots, T_n$ are $\mathcal{RN}$-clausal trees that are not leaves, then $\bigcup_{i=1}^{n} T_i$ is also an $\mathcal{RN}$-clausal tree.*

Proof. The first item of this proposition is justified by the fact that if $l$ and $l'$ are literals such that $l' < l$ and $l$ is a positive literal, then $l'$ is also a positive literal by definition of $<$. The second item is a direct consequence of the definition of a $\mathcal{RN}$-clausal tree. ■

## 2.5 Rewriting constant symbols

The rewriting of a constant into another is at the core of projections. We introduce some notations for denoting the replacement of a constant symbol inside a clause or a clausal tree and we establish some basic properties of this operation.

**Notation 47** Let $a$ and $b$ be two constants such that $a \succ b$, $C$ be a clause in $\mathbb{E}$ and $T$ be a clausal tree. We note $C[a := b]$ (resp. $T[a := b]$) the clause $C$ (resp. the tree $T$) where all the occurrences of $a$ have been replaced $b$. $\Diamond$

Note that if $C$ is not tautological, $C[a := b]$ and $C_{\downharpoonright\{a \not\simeq b\}}$ can be identified.

**Remark 48** *Obviously, this operation does not necessarily need to be explicitly realised, especially when the transformation is not meant to be definitive, as is the case in our algorithms. Instead, it is preferable to store the set of rewrite rules that have been applied on a considered clausal tree, so that they can be applied on demand, in a lazy way.*

**Proposition 49** *For any clausal tree $T$, $\mathcal{C}(T[a := b]) = \mathcal{C}(T)[a := b]$*

**Proposition 50** *Let $C$ be a clause in $\mathbb{E}$ and $l = a \not\simeq b$ with $a \succ b$ be a negative literal in $C$, then $C[a := b] \equiv C \backslash l$*

Proof. By definition, $C \equiv (l \vee C\backslash l)$ and by Proposition 17, $a \simeq b \models C \Leftrightarrow C_{\downharpoonright\{a \not\simeq b\}}$, and so $C \equiv a \not\simeq b \vee C[a := b]$, hence the result. ∎

**Proposition 51** *Let $C$ and $D$ be two clauses in $\mathbb{E}$, let $a$ and $b$ be two constants such that $a \succ b$. 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 coincide with $\mathcal{M}$ on all constants except on $a$ which is such that $a =_{\mathcal{I}} b$. Since $a$ does not appear in $D[a := b]$, $\mathcal{I}$ is also a model of $D[a := b]$. Moreover $\mathcal{I} \models a \simeq b$, hence by Proposition 17, $\mathcal{I} \models D$, and by hypothesis $\mathcal{I} \models C$. Then, again by Proposition 17, $\mathcal{I} \models C[a := b]$ and finally $\mathcal{M} \models C[a := b]$ because $a$ does not appear in $C[a := b]$. ∎

**Proposition 52** *Let $C$ and $D$ be two clauses in $\mathbb{E}$, such that $D[a := b] \models C$, then $D \models C \vee a \not\simeq b$.*

Proof. Let $\mathcal{M}$ be a model of $D$. If $a \neq_{\mathcal{M}} b$, then $\mathcal{M} \models a \not\simeq b$, so $\mathcal{M} \models C \vee a \not\simeq b$. If $a =_{\mathcal{M}} b$, then $\mathcal{M} \models D[a := b]$, so $\mathcal{M} \models C$ and finally $\mathcal{M} \models C \vee a \not\simeq b$. Hence the result. ∎

All the notions necessary to formalise our work have been introduced. The prominent ones are the projection test defined in Corollary 28 and the normal form exposed in Definition 20. These key notions are used extensively in the next two chapters.

# Chapter 3

# Manipulating $\mathcal{N}$-clausal trees

In this chapter, we present the algorithms isEntailed and pruneEntailed, that manipulate $\mathcal{N}$-clausal trees. The algorithm isEntailed tests efficiently if a clause is entailed by one of the clauses contained in a set of clauses. The algorithm pruneEntailed does the contrary test (does a clause entail one of the clauses contained in a set of clauses) and removes at the same time from the set of clauses all the ones that are entailed by the given clause. Each algorithm is described informally and in pseudo-code. Proofs of termination and correctness are presented for both algorithms.

## 3.1 Entailment by an $\mathcal{N}$-clausal tree

### 3.1.1 Algorithm description

In this section, we introduce an algorithm that tests whether a given clause is logically entailed by a clause stored in a given $\mathcal{N}$-clausal tree, even though in practice, the use of projections inside the algorithm recursive calls compels us to always consider an $\mathcal{RN}$-clausal tree instead.

Let $C$ be a clause in normal form and $T$ be an $\mathcal{RN}$-clausal tree. The algorithm isEntailed called on $C$ and $T$ returns $\top$ if and only if there exists a clause $D$ in $\mathcal{C}(T)$ such that $D$ entails $C$. To test this entailment, the algorithm runs through clause $C$, and at the same time performs a depth-first traversal of $T$, attempting to project every encountered literal on $C$. If a literal in $\mathcal{B}_r(T)$ cannot be projected, the exploration of the subtree associated to this literal is useless, so the algorithm switches to the following literal in $\mathcal{B}_r(T)$. As soon as a clause entailing $C$ is found, the exploration halts and $\top$ is returned. Note that in an actual implementation of isEntailed, the projections do not have to be explicitly computed (cf Remark 48).

**Algorithm 1** isEntailed$(C, T)$

**Require:** $C$ is a clause in normal form and $T$ is an $\mathcal{RN}$-clausal tree.
**Ensure:** isEntailed$(C, T) = \top \Leftrightarrow \exists D \in \mathcal{C}(T),\ D \models C$

```
if T = □ then
    return ⊤
end if
if C = □ then
    return ⊥
end if
l_1 ← min_< {l ∈ C}
for all (l, T') ∈ T such that l ≥ l_1 do
    if l_1 = a ≉ b, with a ≻ b then
        if l = l_1 then
            if isEntailed(C\l_1, T') then
                return ⊤
            end if
        else if ¬(l = a ≉ c), with a ≻ c then
            if isEntailed(C\l_1, (T'|_l^↑)[a := b]) then
                return ⊤
            end if
        end if
    else if l ∈ C then
        if isEntailed(C, T') then
            return ⊤
        end if
    end if
end for
return ⊥
```

Termination of the algorithm is proved by induction on $|C| + |T|$, where $|C|$ is the number of literals in $C$, and $|T|$ represents the height of $T$ (note that $|T|$ can also be viewed as the size of the biggest clause in $\mathcal{C}(T)$).

**Theorem 53** isEntailed$(C, T)$ *terminates for any normal clause $C$ and $\mathcal{RN}$-clausal tree $T$.*

Proof. If there are no recursive calls, the algorithm terminates in all cases. In the recursive calls, either $|C|$ or $|T|$ is decremented (and neither $|C|$ nor $|T|$ can increase), so by induction on $|C| + |T|$ the algorithm terminates. ∎

### 3.1.2 Soundness proof

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 an $\mathcal{RN}$-clausal tree. For the clauses, this

is obvious because no rewriting operation is ever performed, and literals are deleted from the smallest to the greatest. This ensures that the resulting clause remains in normal form. For the clausal trees, the proof is more involved:

**Lemma 54** *Let $C$ be a clause and $T$ an $\mathcal{RN}$-clausal tree. All the trees appearing in the recursive calls of* isEntailed$(C, T)$ *are also $\mathcal{RN}$-clausal trees.*

Proof. For any $(l, T') \in T$, $T'$ is an $\mathcal{RN}$-clausal tree by definition. It is also straightforward to see that $T'|_l^{\uparrow}$ is an $\mathcal{RN}$-clausal tree. The only case to consider is when $l_1$ is of the form $a \not\simeq b$ with $a \succ b$, $l > l_1$ and $l$ is not of the form $a \not\simeq c$ with $a \succ c$. We then show by induction that $(T'|_l^{\uparrow})[a := b]$, the argument of the recursive call, is an $\mathcal{RN}$-clausal tree.

We suppose that: $\forall (l', T'') \in T'$, $(T''|_{l'}^{\uparrow})[a := b]$ is an $\mathcal{RN}$-clausal tree. By Proposition 46, $T'[a := b]$ is an $\mathcal{RN}$-clausal tree and if $l$ is a positive literal, so is any $l' \in \mathcal{B}(T')$ thus $(T'|_l^{\uparrow})[a := b]$ is also an $\mathcal{RN}$-clausal tree. If $l$ is of the form $u \not\simeq v$ with $u \succ v$, then necessarily $u \succ a$, because $l > a \not\simeq b$ and $l \neq a \simeq c$ with $a \succ c$, thus $l[a := b]$ is not a contradiction. Furthermore, for all $l' \in \mathcal{B}_r(T')$, if $l'$ is positive, by definition of the order on literals, $l[a := b] < l'[a := b]$. If $l'$ is negative, then $l' = s \not\simeq t$ with $s \succ u$ (and $s \succ t$), so $s \succ a$, hence $l[a := b] < l'[a := b]$. In addition, since $u$ does not appear in $T'$, it also does not appear in $T'[a := b]$ (because $u \neq b$). Since all the properties are verified, we conclude that $(T'|_l^{\uparrow})[a := b]$ is an $\mathcal{RN}$-clausal tree. ∎

Lemma 54 proves that the requirements of the isEntailed algorithm are met at every recursive call. This validates the use of induction in the algorithm's proof of soundness. Note that, unlike $\mathcal{RN}$-clausal trees, $\mathcal{N}$-clausal trees do not verify Lemma 54, their properties are not preserved by the projection operation, which justifies the introduction of a formal definition for $\mathcal{RN}$-clausal trees to reason by induction.

**Theorem 55** isEntailed$(C, T) = \top$ *iff* $\exists D \in \mathcal{C}(T),\ D \models C$

Proof. We first assume that isEntailed$(C, T) = \top$ and show by induction that there exists a clause $D \in \mathcal{C}(T)$ such that $D \models C$. We examine all the cases in which isEntailed$(C, T)$ returns $\top$ in their order of appearance in the algorithm.

- If $T = \square$ then it represents the empty clause and since $\square \models C$, the property holds.
- Assume $l_1 = \min_{<} \{l_i \in C\}$ is of the form $a \not\simeq b$ with $a \succ b$.
  - If there exists a $(l_1, T') \in T$ such that isEntailed$(C \backslash l_1, T')$ is true, then by induction, there exists a $D \in \mathcal{C}(T')$ such that $D \models C \backslash l_1$. Therefore $l_1 \vee D \models C$ and since $l_1 \vee D \in \mathcal{C}(T)$, we have the result.
  - Assume there exists a $(l, T') \in T$ such that $l$ is not of the form $a \not\simeq c$ with $a \succ c$ and isEntailed$(C \backslash l_1,\ T'|_l^{\uparrow}[a := b]) = \top$. Then by

induction there exists a $D' \in \mathcal{C}(T'|_l^{\uparrow}[a := b])$ such that $D' \models C \setminus l_1$. By Proposition 49, there exists a $D \in \mathcal{C}(T'|_l^{\uparrow})$ such that $D[a := b] \models C \setminus l_1$. Thus, $D \models C$ by Proposition 52 and since $\mathcal{C}(T'|_l^{\uparrow}) \subseteq \mathcal{C}(T)$, the property is verified.

- Now assume that $l_1 = a \simeq b$ with $a \succ b$, that there exists a pair $(l, T') \in T$ with $l \geq l_1$ such that $l \in C$ and isEntailed$(C, T')$ is true. By induction there exists a $D \in T'$ such that $D \models C$. Hence $l \vee D \models l \vee C$, and since $l \in C$, $l \vee D \models C$ so the property is verified.

To prove the converse implication, we suppose that isEntailed$(C,\ T) = \bot$ and prove that there is no $D \in \mathcal{C}(T)$ such that $D \models C$. The first case where isEntailed$(C, T)$ returns $\bot$ is when $T \neq \square$ and $C = \square$. All the clauses in $T$ are satisfiable by Proposition 21, so for all $D \in \mathcal{C}(T)$, $D \not\models \square$, i.e. the property is not verified. The other case is when $T \neq \square$ and $C \neq \square$. Then, since $\mathcal{C}(T) = \bigcup_{(l,T') \in T} \mathcal{C}(T'|_l^{\uparrow})$ by Proposition 41, showing that the property does not hold for $T$ is equivalent to showing that is does not hold for all $T'|_l^{\uparrow}$ where $(l, T') \in T$. Let $(l, T') \in T$, we distinguish several cases depending on the natures of $l$ and $l_1 = \min_< \{l \in C\}$, and their relative order. The first case prevents the loop from being entered; the second one prevents the first two recursive calls from being made; the third one prevents the last recursive call from being made, and the fourth one corresponds to the case where there is a recursive call. More formally:

1. $l < l_1$;
2. $l \geq l_1$, $l_1$ is of the form $a \not\simeq b$ and $l$ is of the form $a \not\simeq c$ with $a \succ c \succ b$;
3. $l \geq l_1$, $l_1$ is of the form $a \simeq b$, and $l \notin C$;
4. none of the conditions above hold.

We prove each point separately:

1. Assume $l < l_1$, we distinguish two cases depending on whether $l_1$ is positive or negative.
   - If $l_1 = a \not\simeq b$ with $a \succ b$, then $l$ must be a negative literal and is therefore of the form $l = u \not\simeq v$ where $u \succ v$. By definition of an $\mathcal{RN}$-clausal tree, $l$ cannot be a contradiction, and since $l_i > l$ for all literals $l_i$ in $C$, $l_{\downharpoonright C}$ is not a contradiction. Indeed, since $l < l_i$, $v$ cannot be the maximal constant of a negative literal in $C$, thus by Proposition 22, $v = v_{\downharpoonright C}$; furthermore if $u_{\downharpoonright C} = v$, then $C$ would contain the literal $u \not\simeq v$ which contradicts our hypothesis. Since every clause $D \in T'|_l^{\uparrow}$ contains $l$ and $l_{\downharpoonright C}$ is not a contradiction, by Corollary 28 we conclude that: $\forall D \in T'|_l^{\uparrow}$, $D \not\models C$.
   - If $l_1 = a \simeq b$,then $C$ can contain no negative literal, thus for every literal $l'$, we have $l'_{\downharpoonright C} = l'$ by Proposition 15(2). If $l = u \not\simeq v$, then

$l_{\lfloor C}$ is not a contradiction since $l$ is not, and if $l = u \simeq v$, then for all $l_i \in C$, $l_{\lfloor C} < (l_i)_{\lfloor C}$ so $l_{\lfloor C} \notin C$. Again by Corollary 28, we conclude that $\forall D \in T'|_l^{\uparrow}$, $D \not\models C$.

2. If $l_1 = a \not\simeq b$ and $l = a \not\simeq c$ with $a \succ c \succ b$, then $a_{\lfloor C} = b$ because $C$ is in normal form and $c_{\lfloor C} = c$ by Proposition 22. Therefore $l_{\lfloor C} = b \not\simeq c$ is not a contradiction since $c \succ b$ and we obtain the result by Corollary 28.

3. Assume that , $l_1$ is of the form $a \simeq b$, and $l \notin C$. Since $l \geq l_1$, necessarily $l$ is of the form $l = c \simeq d$ and since $C$ contains no negative literal, $l_{\lfloor C} = l$ by Proposition 15(2). If there exists a clause $D \in T'$ such that $l \vee D \models C$, then by Corollary 28, $l_{\lfloor C} \in C_{\lfloor C}$. But $C_{\lfloor C} = C$ by the definition of a clause in relaxed normal form, so $l \in C$, which is impossible.

4. In all remaining cases, a recursive call is necessarily performed and we distinguish the call that is made.

   (a) Assume that $l_1 = a \not\simeq b$ and $l = l_1$, and let $C_1 = C \backslash l_1$. Then the call to isEntailed$(C_1, T')$ must have returned $\bot$. Suppose that there exists a clause $D' \in \mathcal{C}(T')$ such that $l \vee D' \models C$. By Corollary 28, for all $l' \in D'$, if $l'$ is negative then ${l'}_{\lfloor C}$ is a contradiction and otherwise ${l'}_{\lfloor C} \in C_{\lfloor C}$. But by definition of an $\mathcal{RN}$-clausal tree, constant $a$ does not appear in any $l'$, hence, for all $l' \in D'$, ${l'}_{\lfloor C} = {l'}_{\lfloor C_1}$. Thus, for all $l' \in D'$, if $l'$ is negative then ${l'}_{\lfloor C_1}$ is a contradiction. In the case where $l'$ is positive, since $C$ is in normal form by hypothesis, ${l'}_{\lfloor C} \in C$ and ${l'}_{\lfloor C} = l'$. By assumption, $l_1$ is negative, thus $l' \in C_1$ and ${l'}_{\lfloor C_1} \in {C_1}_{\lfloor C_1}$. By Corollary 28, we deduce that $D' \models C_1$, which by induction, implies that isEntailed$(C_1, T')$ returns $\top$ and this contradicts our hypothesis.

   (b) If $l_1 = a \not\simeq b$ with $a \succ b$, $l > l_1$ and $l$ is not of the form $a \not\simeq c$ with $a \succ c$, then the recursive call that is made is isEntailed$(C\backslash l_1, T'|_l^{\uparrow}[a := b])$ and since it returns $\bot$, by induction, $\forall D \in \mathcal{C}(T'|_l^{\uparrow}[a := b])$, $D \not\models C\backslash l_1$. Now assume that there exists a clause $D' \in \mathcal{C}(T'|_l^{\uparrow})$ such that $D' \models C$. Then, $D'[a := b] \models C[a := b]$ by Proposition 51, and since $C[a := b] \equiv C\backslash l_1$ by Proposition 50, we have also $D'[a := b] \models C\backslash l_1$. But $D'[a := b] \in \mathcal{C}(T'|_l^{\uparrow}[a := b])$ which raises a contradiction.

   (c) Assume $l_1$ is a positive literal, $l \geq l_1$ and $l \in C$. Then the call to isEntailed$(C_1, T')$ must have returned $\bot$ . The proof is identical to case 4a: we assume that there exists a clause $D' \in \mathcal{C}(T')$ such that $l \vee D' \models C$ and derive a contradiction.

Since all trees $T'|_l^{\uparrow}$ contained in $T$ verify one of these conditions, it is clear that $T$ represents no clause entailing $C$, which prove the result. ∎

We have proven that the algorithm isEntailed is sound. We now provide a short informal complexity analysis.

Assume that the rewriting of the constant $a$ with the constant $b$ performed in the recursive call isEntailed$(C\backslash l_1, T'|_l^{\uparrow}[a := b])$ is not carried out by going through the whole tree $T'|_l^{\uparrow}$, but simply taken into account in the following recursive calls (with a constant cost[1]). We can 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, in the worst of all cases, the complexity of isEntailed$(C, T)$ is in $\mathcal{O}(|\{l \in D \mid D \in \mathcal{C}(T)\}| + |C| \times |\mathcal{C}(T)|)$. Obviously, this upper bound should not be reached very often since the algorithm is built to explore as little of $T$ as is necessary. Furthermore, $T$ itself should have in general less edges than there are literals in $\mathcal{C}(T)$, since a literal appearing in several clauses can be represented by a unique edge.

## 3.2 Pruning an $\mathcal{N}$-clausal tree

In this section, we present an algorithm that permits to delete from a set of clauses represented by an $\mathcal{N}$-clausal tree all those that are entailed by a given clause. This algorithm is similar in its structure to the previous one, but with the roles of the clause and the tree reversed, thus, this time, $C$ will be in relaxed normal form and $T$ will be an $\mathcal{N}$-clausal tree. Moreover, this time the tree is pruned by the algorithm.

### 3.2.1 Algorithm description

Let $C$ be a clause in relaxed normal form and $T$ be an $\mathcal{N}$-clausal tree. The algorithm pruneEntailed called on $C$ and $T$ removes from $T$ all clauses entailed by $C$. To do so, it performs a depth-first traversal of $T$ and attempts to project $C$ on every clause in $\mathcal{C}(T)$, deleting those on which such a projection succeeds. As soon as a projection is identified as impossible, the exploration of the associated subtree associated halts and the algorithm moves to the next clause. When every literal in $C$ has been projected, all the clauses represented in the corresponding subtree are entailed by $C$, thus they are removed.

[1] because necessarily $b = b_{\downarrow C}$, thus each $a$ is rewritten at most only once; and assuming a constant access to each rewritting, stored for example in a hashtable

**Algorithm 2** pruneEntailed$(C, T)$

**Require:** $C$ is a clause in relaxed normal form, $T$ is an $\mathcal{N}$-clausal tree and isEntailed$(C, T) = \bot$
**Ensure:** $\forall D \in \mathcal{C}(T)$, $C \not\models D$
  **if** $C = \square$ **then**
    $T \leftarrow \emptyset$
    **exit**
  **end if**
  **if** $T = \square$ **then**
    **exit**
  **end if**
  $l_1 \leftarrow \min_{<} \{l_i \in C\}$
  **for all** $(l, T') \in T$ such that $l \leq l_1$ **do**
    **if** $l_1 = l$ **then**
      pruneEntailed$(C \backslash l_1, T')$
    **else**
      **if** $l = a \simeq b$ **then**
        pruneEntailed$(C, T')$
      **else if** $l = a \not\simeq b$, with $a \succ b$ **and** $\nexists c$, $l_1 = a \not\simeq c$, with $a \succ c$ **then**
        pruneEntailed$(C[a := b], T')$
      **end if**
    **end if**
  **end for**

**Remark 56** *If this is not done by a built-in mechanism of the data structure, the pairs $(l, \emptyset)$ introduced in $T$ during the execution of the algorithm can be directly removed by adding the following loop at the end of the algorithm:*

***for all*** $(l, T') \in T$ ***do***
  ***if*** $T' = \emptyset$ ***then***
    $T \leftarrow T \backslash \{(l, T')\}$
  ***end if***
***end for***

*The recursive nature of the algorithm ensures that the removals will be done in an incremental and efficient way*

The termination proof of pruneEntailed is similar to that of isEntailed by induction on $|T|$.

### 3.2.2 Soundness proof

As with the previous algorithm, before proving the soundness, we must ensure that the requirements of the algorithm are met by all the recursive calls.

**Notation 57** Let $C$ be a clause in relaxed normal form and $T$ an $\mathcal{N}$-clausal tree. We will denote by $T_{out}$ the $\mathcal{N}$-clausal tree $T$ after the call of

PRUNEENTAILED$(C, T)$ (and $T$ will denote the $\mathcal{N}$-clausal tree before the call). It will always be clear from the context which call of PRUNEENTAILED generated $T_{out}$ (i.e. the main call or one of the recursive calls). $\Diamond$

**Proposition 58** *Let $C$ be a clause in relaxed normal form and $T$ an $\mathcal{N}$-clausal tree. If* PRUNEENTAILED$(C, T)$ *is called, then $T_{out}$ is an $\mathcal{N}$-clausal tree.*

PROOF. It is clear that no element is added to $T$ and no literal in $\mathcal{B}_r(T)$ is modified during the execution of PRUNEENTAILED$(C, T)$. The only operations allowed by the algorithm are replacing subtrees with empty sets and removing elements from $T$. Thus, the order constraints required by an $\mathcal{N}$-clausal tree are preserved from $T$ to $T_{out}$. ∎

**Lemma 59** *Let $C$ be a clause in relaxed normal form and $T$ an $\mathcal{N}$-clausal tree. All the clauses arguments of a recursive call in* PRUNEENTAILED$(C, T)$ *are in relaxed normal form.*

PROOF. Since $C$ is in relaxed normal form, clearly for any literal $l_i \in C$, $C \backslash l_1$ is also in relaxed normal form. The only case that must be detailed is when $l < l_1$ (with $(l, T') \in T$ and $l_1 = \min_< \{l_i \in C\}$), $l = a \not\succ b$, with $a \succ b$ and $\neg(l_1 = a \not\succ c$, with $a \succ c)$, where $C[a := b]$ is an argument of a recursive call of PRUNEENTAILED.

If $l_1$ is a positive literal, then all the literals of $C$ and $C[a := b]$ are positive. Thus by Proposition 15(2), for all $l_i \in C[a := b]$, $l_i = l_{i \downarrow C[a:=b]}$, and so $C[a := b]$ is in relaxed normal form. In the case where $l_1$ is a negative literal (of the form $u \not\succ v$ with $u \succ v$), we prove by induction on the size of clauses that if $C$ is in relaxed normal form, so is $C[a := b]$. By definition, $C \backslash l_1$ is in relaxed normal form and by induction $(C \backslash l_1)[a := b]$ is too. The literal $l_1[a := b]$ (denoted by $l'_1$ in the rest of the proof) verifies the following properties:

- $l'_1 = u' \not\succ v'$ is not a contradiction. Since $l_1 = u \not\succ v$ is not a contradiction, and $u \succ a$ by hypothesis, $l'_1$ cannot be a contradiction. Indeed, $u \neq a$, thus $u' = u$ and $v' \leq v < u$ hence $u' \neq v'$.
- $l'_1$ is unique in $C[a := b]$. The literal $l'_1$ is smaller than all the positive literals in $C[a := b]$. Moreover, for any negative literal $l'_2 \in C \backslash l_1[a := b]$, the corresponding literal $l_2 \in C \backslash l_1$ is of the form $s \not\succ t$ (where $s \succ t$), with $s \succ u$, so $s \succ a$ and $l'_1 < l'_2$ (since $u = u'$).
- $l'_1 = u' \not\succ v'$ with ${u'}_{\downarrow C[a:=b]} = v'$. To respect the minimality of $l'_1$ in $C[a := b]$ (proven in the previous point), necessarily ${v'}_{\downarrow C[a:=b]} = v'$. Since ${u'}_{\downarrow C[a:=b]} = {v'}_{\downarrow C[a:=b]}$, the property holds.

In addition, since $u$ does not appear in any literal in $C \backslash l_1$ and since $b \neq u$, for all literal $l_i \in C \backslash l_1$, $l_{i \downarrow C[a:=b]} = l_{i \downarrow (C \backslash l_1)[a:=b]}$. Thus, the properties verified by induction by the literals of $(C \backslash l_1)[a := b]$ are also verified in $C[a := b]$. ∎

As with the previous algorithm, the use of induction in the soundness proof is valid, because Proposition 58 and Lemma 59 ensure that the requirements of the algorithm are respected in the recursive calls; and as with $\mathcal{RN}$-clausal trees in the soundness proof of isEntailed, the definition of a relaxed normal form for clauses is necessary to preserve the invariant of the induction.

**Theorem 60** *Let $C$ be a relaxed normal clause and $T$ an $\mathcal{N}$-clausal tree such that* $\text{isEntailed}(C, T) = \bot$. *The following result holds:* $\mathcal{C}(T_{out}) = \{D \in \mathcal{C}(T) \mid C \not\models D\}$

Proof. If $D \notin \mathcal{C}(T)$, then $D \notin \mathcal{C}(T_{out})$, because $T_{out}$ is merely the tree $T$ with less branches. Thus, this proof will focus only on clauses in $\mathcal{C}(T)$. If $C = \Box$, 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 $T_{out} = \emptyset$. Now assume that $C \neq \Box$. We prove that $C(T_{out}) = \{D \in \mathcal{C}(T) \mid C \models D\}$ by proving the two inclusions successively:

Let $D \in \mathcal{C}(T)$ such that $C \models D$. We show by induction that $D \notin \mathcal{C}(T_{out})$. If $D = \Box$ (i.e. $T = \Box$), then $C$ is a contradiction and since $C$ is in relaxed normal form, $C = \Box$. Thus, $T_{out} = \emptyset$ and $D \notin \mathcal{C}(T_{out})$. From now on, we assume that $C \neq \Box$ and $D \neq \Box$. We write $l_1 = \min_<\{l_i \in C\}$, $l = \min_<\{l' \in D\}$ and $D' = D \backslash l$. Since $T$ is an $\mathcal{N}$-clausal tree and $D \in \mathcal{C}(T)$, by definition there exists a unique $\mathcal{N}$-clausal tree $T'$ such that $(l, T') \in T$ and $D' \in \mathcal{C}(T')$. There are several cases to consider:

1. $l > l_1$, in which case no recursive call is done,
2. $l = a \not\simeq b$ and $l_1 = l$, in which case $\text{pruneEntailed}(C \backslash l_1, T')$ is called,
3. $l = a \not\simeq b$ and $l_1 = a \not\simeq c$, with $a \succ c \succ b$, in which case no recursive call is done,
4. $l = a \not\simeq b$ and $l_1 > l$ with $l_1$ not of the form $a \not\simeq c$ with $a \succ c \succ b$, in which case $\text{pruneEntailed}(C[a := b], T')$ is called,
5. $l = a \simeq b$ and $l_1 = l$, in which case $\text{pruneEntailed}(C \backslash l_1, T')$ is called,
6. $l = a \simeq b$ and $l_1 > l$, in which case $\text{pruneEntailed}(C, T')$ is called.

As can be seen, those cases cover all the possible relations between $l$ and $l_1$.

1. Assume $l > l_1$. Since $C \models D$, the two following implications hold:
   - If $l_1 = c \not\simeq d$ with $c \succ d$, then $l_{1\restriction D}$ is a contradiction by Corollary 28, so $c_{\restriction D} = d_{\restriction D}$. But for all $l' \in D$, $l' > l_1$, thus by Proposition 22, $d_{\restriction D} = d$ (or else $d \not\simeq d_{\restriction D} \in D$ which is impossible because $d \not\simeq d_{\restriction D} < l_1$ and $l$, greater than $l_1$, is the minimal literal of $D$, by definition of an $\mathcal{N}$-clausal tree). Therefore $c_{\restriction D} = d$ and by definition of a clause in normal form, $c \not\simeq d \in D$ which contradicts our hypothesis.

- If $l_1$ is positive, then $l_{1\restriction D} \in D_{\restriction D}$ by Corollary 28, and because $D$ is in normal form, $l_{1\restriction D} \in D$. But since for all $l' \in D$, the inequation $l' > l_1$ holds, $D$ has only positive literals. Hence by Proposition 15(2), $l_{1\restriction D} = l_1$, thus $l_1 \in D$ which contradicts our hypothesis.

2. Assume $l = a \not\simeq b$ with $a \succ b$ and $l_1 = l$. In this case, pruneEntailed$(C\backslash l_1, T')$ is called. Since $C \models D$, by Corollary 28, for any literal $l' \in C$, with $l' \neq l_1$:

   - Either $l'$ is negative and ${l'}_{\restriction D}$ is a contradiction. By definition of a clause in relaxed normal form, the constant $a$ cannot appear in any literal of $C$ other than $l_1$, hence $C\backslash {l_1}_{\restriction D} = C\backslash {l_1}_{\restriction D'}$. Thus ${l'}_{\restriction D'}$ is also a contradiction.
   - Or $l'$ is positive and ${l'}_{\restriction D} \in D_{\restriction D}$. But by definition of an $\mathcal{N}$-clausal tree, the positive literals of $D_{\restriction D}$ are the same as those of $D$, $D'$ and ${D'}_{\restriction D'}$. Hence ${l'}_{\restriction D'} \in {D'}_{\restriction D'}$.

   By Corollary 28, $C\backslash l_1 \models D'$ and by induction $D' \notin \mathcal{C}(T'_{out})$, thus $D \notin \mathcal{C}(T_{out})$.

3. If $l = a \not\simeq b$ and $l_1 = a \not\simeq c$, with $a \succ c \succ b$, then $l_{1\restriction D} = b \not\simeq c$ is not a contradiction. Thus by Corollary 28, $C \not\models D$, which contradicts our hypothesis.

4. If $l = a \not\simeq b$ with $a \succ b$ and $l_1 > l$ where $l_1$ is not of the form $a \not\simeq c$ with $a \succ c \succ b$, then pruneEntailed$(C[a := b], T')$ is called. By Proposition 51, $C[a := b] \models D[a := b]$, and $D[a := b] \equiv D'$ by Proposition 50, hence $C[a := b] \models D'$. By induction $D' \notin \mathcal{C}(T'_{out})$, so $D \notin \mathcal{C}(T_{out})$.

5. Assume $l = a \simeq b$ and $l_1 = l$. In this case, both $C$ and $D$ contain only positive literals, thus for any $l' \in C$ such that $l' \neq l_1$, by Proposition 15(2), ${l'}_{\restriction D} = {l'}_{\restriction D'} = l'$ and $D_{\restriction D} = D$. Furthermore, by Corollary 28, $l' \in D_{\restriction D}$, hence $l' \in D'$. Again by Corollary 28, $C\backslash l_1 \models D'$, so by induction $D' \notin \mathcal{C}(T'_{out})$ and finally $D \notin \mathcal{C}(T_{out})$.

6. If $l = a \simeq b$ and $l_1 > l$, the same reasoning as for the previous point holds for any $l' \in C$, including $l_1$, thus $C \models D'$ and $D' \notin \mathcal{C}(T'_{out})$ by induction.

Now assume that $D \in \mathcal{C}(T)$ such that $C \not\models D$ (note that necessarily $C \neq \Box$). We show by induction that $D \in \mathcal{C}(T_{out})$. If $D = \Box$, then $T = \Box$ and $T_{out} = T$, so $D \in \mathcal{C}(T_{out})$. As before we write $l_1 = \min_<\{l_i \in C\}$, $l = \min_<\{l_i \in D\}$, $D' = D\backslash l$ and we consider the unique couple $(l, T') \in T$. By Corollary 28, there must be a literal $l' \in C$ that cannot be projected on $D$. We must consider the same cases as before:

1. If $l_1 < l$, no recursive call is done on $T'$, and $T' \neq \emptyset$ because $D' \in \mathcal{C}(T')$, thus $D \in \mathcal{C}(T_{out})$.

2. Assume $l = a \not\simeq b$ with $a \succ b$ and $l_1 = l$. Clearly, $l' \neq l_1$ since $l_1 \in D$, thus $l' \in C \backslash l_1$ and $l'$ also cannot be projected on $D'$, because ${D'}_{\downarrow D'} = (D_{\downarrow D}) \backslash (l_{\downarrow D})$. Hence $C \backslash l_1 \not\models D'$ and by induction $D' \in \mathcal{C}(T'_{out})$. By definition, $D \in \mathcal{C}(T_{out})$.

3. If $l = a \not\simeq b$ and $l_1 = a \not\simeq c$, with $a \succ c \succ b$, as seen before, $C \not\models D$. No recursive call is done, so $T'_{out} = T'$, thus $D \in \mathcal{C}(T_{out})$.

4. Assume $l = a \not\simeq b$ with $a \succ b$ and $l_1 > l$ where $l_1$ is not of the form $a \not\simeq c$ with $a \succ c \succ b$. The constant $a$ does not occur in $l'$ by Proposition 15(2), thus $l'[a := b] = l'$ and ${l'}_{\downarrow D} = {l'}_{\downarrow D'}$. This allows us to conclude that $l' \in C[a := b]$ and $l'$ cannot be projected on $D'$ because ${D'}_{\downarrow D'} = (D_{\downarrow D}) \backslash (l_{\downarrow D})$, hence by Corollary 28, $C[a := b] \not\models D'$. By induction $D' \in \mathcal{C}(T'_{out})$ and so $D \in \mathcal{C}(T_{out})$.

5. If $l = a \simeq b$ and $l_1 = l$, then as in Point 2, $l' \neq l_1$, hence $l' \in C \backslash l_1$. Furthermore, all the literals in $D$ are positive, thus ${D'}_{\downarrow D'} = D \backslash l$. It implies that $l'$ cannot be projected on $D'$ and by Corollary 28, $C \backslash l_1$ does not entail $D'$. By induction $D' \in \mathcal{C}(T'_{out})$ and so $D \in \mathcal{C}(T_{out})$.

6. If $l = a \simeq b$ and $l_1 > l$, the proof is the same as in Point 5, except that it is possible that $l' = l_1$, thus it is $C$ that does not entail $D'$ instead of $C \backslash l_1$. ∎

Like with the algorithm isEntailed, we have proven the soundness of pruneEntailed and are now concerned with the efficiency of this algorithm, that we will consider through a short complexity analysis. The two algorithms isEntailed and pruneEntailed have a similar structure in terms of recursive calls, hence they also have a similar complexity. However, as mentioned in the termination proof of pruneEntailed for which the induction is based solely on $|T|$, even in the worst case the recursive calls of pruneEntailed always reduce the tree, which is not the case in isEntailed. Thus these recursive calls are not influenced by the number of literals in $C$, which ensure a slightly better theoretical complexity for pruneEntailed than for isEntailed (it is in $\mathcal{O}(|\{l \in D \mid D \in \mathcal{C}(T)\}|)$ in the worst case). In practice, it is very likely that isEntailed will be more efficient in general, because it is more constrained than pruneEntailed by the number of literals in $C$, which is likely to be small compared to that of $T$. To verify this hypothesis, the most practical way is to run the algorithms on a benchmark and analyse the results (which will be done if this master leads to a PhD).

We have presented efficient storage data-structures and algorithms to test the entailment of equational clauses. In the context of the generation of prime implicates adapted from CLTMS, these two algorithms are completed by a third one, which takes care of inserting a new clause into an $\mathcal{N}$-clausal tree. The three algorithms put together permit an update of the implicates stored in $T$:

- isEntailed$(C, T)$ tests whether an implicate $C$ is entailed by a set of implicates $\mathcal{C}(T)$; if it is not,

- PRUNEENTAILED$(C, T)$ removes from $\mathcal{C}(T)$ the implicates that are entailed by $C$; and finally,
- the last algorithm inserts $C$ into $\mathcal{C}(T)$.

These three steps allow for an efficient storage and manipulation of the implicates of a formula, until only the prime implicates remain.

The remaining problem to solve is that of the generation of $C$ and the other implicates to be (or not to be, depending on the result of ISENTAILED) inserted in $\mathcal{C}(T)$. It is the subject of the next chapter.

# Chapter 4

# Generating implicates in equational logic

In this chapter, we introduce a new calculus inspired from the superposition calculus and prove that it generates all the implicates of a given formula.

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

**Remark 61** *In the previous chapters, the notions of propositional and equational entailment where never intertwined, it was always clear from the context which one was being referred to. It is not so in this chapter, hence the introduction of the following notations to clarify the different uses of the symbol* $\models$.

**Notation 62** Let $\mathcal{I}$ be an interpretation and $S$ be a formula. When we write $\mathcal{I} \models S$, we mean that $\mathcal{I}$ is a model of $S$, and the context is sufficient for us to know whether $\mathcal{I}$ is a propositional or an equational interpretation. We will now distinguish explicitly these two kinds of interpretations by always writing $\mathcal{I} \models_0 S$ when $\mathcal{I}$ is a propositional interpretation, and $\mathcal{I} \models_{\mathcal{E}} S$ when $\mathcal{I}$ is an equational interpretation. Similarly, if $S$ and $S'$ are two formulas, then $S \models_0 S'$ means that every propositional model of $S$ is also a propositional model of $S'$, and $S \models_{\mathcal{E}} S'$ means that every equational model of $S$ is also an equational model of $S'$. $\Diamond$

Note that if $S \models_0 S'$, then $S \models_{\mathcal{E}} S'$, since the equational models of $S$ are a subset of the propositional models of $S$. For example, if $S = \{a \simeq b; b \simeq c\}$ and $S' = \{a \simeq b\}$, then $S \models_0 S'$, thus $S \models_{\mathcal{E}} S'$. In this case, $\mathcal{I}$ that evaluates $a \simeq b$, $b \simeq c$ and $a \not\simeq c$ to $\top$ is a propositional model of both $S$ and $S'$. $\mathcal{I}$ is not an equational interpretation because it does not respect transitivity between $a$, $b$ and $c$. On the contrary, an equational interpretation $\mathcal{J}$ that puts $a$, $b$ and $c$ in the same equivalence class is an equational model of $S$ and $S'$ but also a propositional model of these formulas that evaluates $a \simeq b$, $b \simeq c$ and $a \simeq c$ to $\top$. The

converse (if $S \models_{\mathcal{E}} S'$, then $S \models_0 S'$) is not true. For example, with $C = e \simeq f$ and $D = a \not\simeq b \vee b \not\simeq c \vee a \simeq c$, it is clear that $C \models_{\mathcal{E}} D$ because $D$ is an instantiation of the axiom of transitivity, thus is always true in equational logic; but $C \not\models_0 D$ since it is possible to define a propositional interpretation that evaluates $e \simeq f$, $a \simeq b$, $b \simeq c$ and $a \not\simeq c$ to $\top$.

### 4.1.1 Intuition of the calculus

Our objective is to create a calculus operating on clauses in $\mathbb{E}$ (the set of all ground flat equational clauses) that computes all the implicates of a formula. To achieve this, we define the *$\mathcal{K}$-paramodulation calculus* in $\mathbb{E}$.

The principle underlying this calculus is to assert equalities rather than proving them contrary 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 this formula since no constant occurs in both clauses. On the contrary, 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 superposition of $a$ into $c$ under the assumption that $a$ and $c$ are equal. To do so, we must allow the superposition of constants that are not known to be equal, like $a$ and $c$ in $S$, by adding into the generated clause a hypothesis justifying this superposition, here, the literal $a \not\simeq c$.

### 4.1.2 Formal definition

**Definition 63** The $\mathcal{K}$-paramodulation is defined in $\mathbb{E}$ by three inference rules:

- $\mathcal{K}$-paramodulation[1]:
$$\frac{a \simeq b \vee C \quad a' \bowtie c \vee D}{a \not\simeq a' \vee b \bowtie c \vee C \vee D}$$
- $\mathcal{K}$-equational factorisation:
$$\frac{a \simeq b \vee a' \simeq b' \vee C}{a \not\simeq a' \vee b \not\simeq b' \vee a \simeq b \vee C}$$
- Reflexion:
$$\frac{a \not\simeq a \vee C}{C}$$

$\Diamond$

**Remark 64** *Intuitively, the rules of $\mathcal{K}$-paramodulation can be interpreted as:*

- *$\mathcal{K}$-paramodulation: Under the assumption that the constants $a$ and $a'$ are equal, a superposition can be performed between $a \simeq b \vee C$ and $a' \bowtie c \vee D$.*
- *$\mathcal{K}$-equational factorisation: Under the assumption that $a \simeq b$ and $a' \simeq b'$ are equal, one of these two literals can be removed from the clause.*

[1] recall that $\bowtie$ stands for $\simeq$ or $\not\simeq$

- *Reflexion: This rule is used to remove from clauses the contradictory literals (literals of the form $a \not\simeq a$). Notice that the Reflexion rule is actually not needed for completeness. However, it is useful to simplify the generated clauses*

**Lemma 65** *The $\mathcal{K}$-paramodulation calculus is sound, i.e., the clauses generated by this calculus are logical consequences of their premises.*

**Example 66** By $\mathcal{K}$-paramodulation, we have $a \simeq b, c \simeq d \vdash_{\mathcal{K}} a \not\simeq c \vee b \simeq d$ as we wanted. Note also that $\mathcal{K}$-equational factorisation associated with Reflexion simulates the standard factorisation (of the superposition calculus), and can thus be used to get rid of duplicate literals. For example, we have $a \simeq b \vee a \simeq b \vdash_{\mathcal{K}} a \not\simeq a \vee b \not\simeq b \vee a \simeq b$ by $\mathcal{K}$-equational factorisation and $a \not\simeq a \vee b \not\simeq b \vee a \simeq b \vdash_{\mathcal{K}} a \simeq b$ by Reflexion. ♣

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

Since the $\mathcal{K}$-paramodulation calculus simulates the superposition calculus, it is *refutationaly complete*: if a set of clauses $S$ is closed[2] and does not contain the empty clause, then $S$ is satisfiable. But refutational completeness is not enough to ensure that this calculus can generate all implicates of a set of clauses. This is why we prove the following stronger result: if $S$ is closed, $C$ is not a tautology and $S \models_{\mathcal{E}} C$, then there exists a clause $D \in S$ such that $D \models_{\mathcal{E}} C$. The proof proceeds as follows. We assume that $S$ is closed, $S \models_{\mathcal{E}} C$ and $S$ contains no clause $D$ such that $D \models_{\mathcal{E}} C$ and we construct an equational interpretation satisfying $S \cup \neg C$ – thus contradicting the fact that $S \models_{\mathcal{E}} C$. This construction is done in two steps. First, in Section 4.2.1, we associate each pair $(C, S)$ with a propositional interpretation $\mathcal{I}(C, S)$, constructed by induction on a suitably chosen ordering $<_C$. Then, in Section 4.2.2, we show that $\mathcal{I}(C, S)$ is actually an equational model of $S \cup \neg C$ if $S$ is saturated and contains no implicant of $C$.

### 4.2.1 Construction of a fitting order and interpretation

What we want to prove is that every clause entailed by a formula can be generated by $\mathcal{K}$-paramodulation. To do so, we first define a new ordering $<_C$ on literals that allows us to distinguish the literals entailing a given clause $C$ (by ensuring that these literals are smaller than the other ones), and to order all the literals according to their projection on $C$.

**Definition 67** Let $C$ be a non-tautological clause in $\mathbb{E}$. The partial order on literals $<_C$ is defined as follows: let $l_1, l_2$ be two literals, then $l_1 <_C l_2$ iff one of the conditions below holds:

- $l_1 \models_{\mathcal{E}} C$ and $l_2 \not\models_{\mathcal{E}} C$;

[2] this concept will be defined formally in Definition 72

- or else $l_{1\restriction C} = a_1 \bowtie b_1$, with $a_1 \succeq b_1$; $l_{2\restriction C} = a_2 \bowtie b_2$, with $a_2 \succeq b_2$ and
  - $a_1 \prec a_2$, or
  - $a_1 = a_2$ and $b_1 \prec b_2$. $\Diamond$

This order is extended to clauses in the usual way for multisets.

**Proposition 68** *$<_C$ is transitive, hence is a partial order.*

**Example 69** Let $C = b \not\simeq c \vee b \simeq d$. The inequality $c \simeq d <_C a \simeq b$ holds because $c \simeq d \models_{\mathcal{E}} C$ and $a \simeq b \not\models_{\mathcal{E}} C$. Moreover, since $a \not\simeq b \not\models_{\mathcal{E}} C$, the literals $a \simeq b$ and $a \not\simeq b$ cannot be compared, which can also be written as $a \simeq b \not<_C a \not\simeq b$ and $a \not\simeq b \not<_C a \simeq b$. $\clubsuit$

From now on, all atoms are considered modulo commutativity. For example $a \simeq b$ and $b \simeq a$ are assumed to represent the same atom.

**Definition 70** Let $C$ be a clause in $\mathbb{E}$ and $S$ be a set of clauses in $\mathbb{E}$. We assume w.l.o.g. the set of constants to be finite, thus the set of literals is also finite. We create the finite enumeration of atoms $p_1, \ldots, p_n$ such that:

- for any atom $l$, there exists a unique $i \in \{1 \ldots n\}$ such that $l_{\restriction C} = p_i$,
- for any $i, j \in \{1 \ldots n\}$, if $i < j$, then $p_i <_C p_j$.

We define inductively the following *propositional* interpretations. For all $i \in \{1 \ldots n\}$, let $\mathcal{I}_i(C, S)$ be the propositional interpretation such that:

1. for any atom $l$ such that $l_{\restriction C} = p_j$ with $j < i$, $\mathcal{I}_i(C, S) \models_0 l$ iff $\mathcal{I}_{i-1}(C, S) \models_0 l$,
2. for any atom $l$ such that $l_{\restriction C} = p_j$ with $j > i$, $\mathcal{I}_i(C, S) \not\models_0 l$,
3. for any atom $l$ such that $l_{\restriction C} = p_i$, $\mathcal{I}_i(C, S) \models_0 l$ iff
   (a) either $p_i$ is of the form $a \simeq a$
   (b) or $p_i$ does not occur in the normal form of $C$ ($p_i \notin C_{\downarrow}$) and there exists a clause $D \vee l' \in S$ such that:
   - $l'_{\restriction C} = p_i$,
   - $D <_C l'$,
   - $\mathcal{I}_{i-1}(C, S) \not\models_0 D$.

We note $\mathcal{I}(C, S)$ the interpretation $\mathcal{I}_n(C, S)$. For all $i \in 1 \ldots n$, $\mathcal{I}(C, S)$ coincides with $\mathcal{I}_i(C, S)$ for all atoms $l$ such that $l_{\restriction C} = p_j$ with $j \leq i$. $\Diamond$

**Remark 71** *By definition, for two atoms $l$ and $l'$ such that $l_{\restriction C} = l'_{\restriction C}$, necessarily, either $\mathcal{I}(C, S) \models_0 l$ and $\mathcal{I}(C, S) \models_0 l'$, or $\mathcal{I}(C, S) \not\models_0 l$ and $\mathcal{I}(C, S) \not\models_0 l'$.*

The proof of the completeness theorem consists in showing that this propositional interpretation is actually an equational interpretation, that satisfies the formula $S \cup \neg C$ when $S$ is closed, entails $C$ and contains no clause $D$ such that $D \models_{\mathcal{E}} C$. This is done in the next section.

### 4.2.2 Validating the interpretation $\mathcal{I}(C, S)$

**Definition 72** Let $S$ be a set of clauses in $\mathbb{E}$. $S$ is *closed* iff every clause $C$ deducible from $S$ by the $\mathcal{K}$-paramodulation calculus is either a tautology or an element of $S$. $\Diamond$

The following theorem expresses the fact the $\mathcal{K}$-paramodulation calculus is complete in terms of implicate generation.

**Theorem 73** *Let $S$ be a closed set of clauses in $\mathbb{E}$ and $C \in \mathbb{E}$ be a non-tautological clause. If $S \models_{\mathcal{E}} C$, then there exists a clause $D \in S$ such that $D \models_{\mathcal{E}} C$.*

Proof. Let $S$ be a closed set of clauses in $\mathbb{E}$. Assume $C \in \mathbb{E}$ is a non-tautological clause such that $S \models_{\mathcal{E}} C$ and for all clauses $D \in S$, $D \not\models_{\mathcal{E}} C$. We prove that this hypothesis cannot hold by exhibiting an interpretation $\mathcal{I}$ such that $\mathcal{I} \models_{\mathcal{E}} S$ and $\mathcal{I} \not\models_{\mathcal{E}} C$. Let $\mathcal{I} = \mathcal{I}(C, S)$ as defined in Definition 70. Note that $\mathcal{I}$ was constructed as a propositional interpretation. We prove first that $\mathcal{I} \not\models_0 C$, which, if $\mathcal{I}$ is an equational interpretation, is the same as proving that $\mathcal{I} \not\models_{\mathcal{E}} C$. Then, we prove by induction that $\mathcal{I}$ is an equational interpretation such that $\mathcal{I} \models_{\mathcal{E}} S$.

Assume that $\mathcal{I} \models_0 C$, in this case there is a literal $l \in C$ such that $\mathcal{I} \models_0 l$.

- If $l$ is a positive literal, then there exists an $i$ in $\{1 \dots n\}$ such that $l_{\downharpoonright C} = p_i$. By definition of $\mathcal{I}$, either $p_i$ is of the form $a \simeq a$, thus $C_{\downarrow}$ and $C$ are tautologies which is impossible; or $p_i \notin C_{\downarrow}$ hence $l \not\models_{\mathcal{E}} C$, which contradicts the hypothesis $l \in C$. In both cases, this situation cannot happen.

- Otherwise $l$ is a negative literal and there exists an $i$ in $\{1 \dots n\}$ such that $l_{\downharpoonright C} = \bar{p_i}$. In this case, $p_i$ is not a tautology by Condition 3a of the definition of $\mathcal{I}$. This is impossible by Corollary 28, indeed $l$ is of the form $a \not\simeq b$ with $a \succ b$ and $l \models_{\mathcal{E}} C$, thus $a_{\downharpoonright C} = b_{\downharpoonright C}$ and consequently $p_i = a_{\downharpoonright C} = a_{\downharpoonright C}$. This proves that $\mathcal{I} \not\models_0 C$.

To prove that $\mathcal{I}$ is an equational interpretation and that $\mathcal{I} \models_{\mathcal{E}} S$, we go back to the definition of $\mathcal{I}$ (see Definition 70). By induction, we assume that $\mathcal{I}_{i-1}(C, S)$ is an equational interpretation when restricted to the literals $l$ such that $p_{i-1} \not<_C l$, and that for all clauses $D \in S$ whose literals verify this relation with $p_i$, we have $\mathcal{I}_{i-1}(C, S) \models_{\mathcal{E}} D$. We show that the same condition holds for $\mathcal{I}_i(C, S)$ and $p_i$.

We first show that $\mathcal{I}_i(C, S) \models D$, with $D$ such that every literal $l \in D$ is not greater than $p_i$ ($\forall l \in D$, $p_i \not<_C l$). If $D$ contains only literals $l$ such that $l <_C p_i$ then the proof follows from the induction hypothesis, since by definition $\mathcal{I}_i(C, S)$ coincide with $\mathcal{I}_{i-1}(C, S)$ on such atoms. Thus we assume that $D$ contains literals such that their projection on $C$ is either $p_i$ or $\bar{p_i}$. Several cases must be distinguished.

1. In the case where there are two literals $l$ and $l'$ in $D$ such that $l_{\downharpoonright C} = p_i$ and ${l'}_{\downharpoonright C} = \bar{p_i}$, by definition, if $\mathcal{I}_i(C, S) \not\models_0 l$, then $\mathcal{I}_i(C, S) \models_0 l'$ and if $\mathcal{I}_i(C, S) \not\models_0 l'$, then $\mathcal{I}_i(C, S) \models_0 l$. Thus in both cases $\mathcal{I}_i(C, S) \models_0 D$

2. In the case where there is no literal in $D$ that can be projected to $p_i$ on $C$, there are two possibilities to consider:

   - If there is only one literal $l \in D$ such that $l_{\downarrow C} = \bar{p_i}$, then we write $D = l \vee D'$ with $D' <_C l$. If $\mathcal{I}_i(C,S) \models_0 \bar{p_i}$, then the result is direct, thus we assume $\mathcal{I}_i(C,S) \models_0 p_i$ and by definition, there are two cases to examine:
     - If $p_i$ is a tautology, then $l_{\downarrow C}$ is a contradiction, and by Corollary 28, $l \models_{\mathcal{E}} C$. Then by definition of $<_C$, we have $l' \models_{\mathcal{E}} C$ for every $l' \in D'$, hence $D \models_{\mathcal{E}} C$, which contradicts our hypothesis.
     - If $p_i$ is not a tautology, then $p_i \notin C_{\downarrow}$ and there exists a clause $D'' \vee l' \in S$, where ${l'}_{\downarrow C} = p_i$, $D'' <_C l'$ and $\mathcal{I} \not\models_0 D''$. Let $l = a \simeq b$ and $l' = a' \simeq b'$ with $a_{\downarrow C} = {a'}_{\downarrow C}$ and $b_{\downarrow C} = {b'}_{\downarrow C}$. We write $E = a \not\simeq a' \vee b \not\simeq b' \vee D' \vee D''$, note that $E <_C p_i$ because $a \not\simeq a', b \not\simeq b' \models_{\mathcal{E}} C$ and $D', D'' <_C p_i$. By $\mathcal{K}$-paramodulation, $l \vee D', l' \vee D'' \vdash_{\mathcal{K}} E$, and since $S$ is closed, $E \in S$ or $E$ is a tautology. If $E$ is a tautology, then $\mathcal{I}_i(C,S) \models_0 E$, and if $E \in S$, we reach the same conclusion by induction, because $E <_C p_i$. Moreover, since $\mathcal{I}_i(C,S) \not\models_0 a \not\simeq a', b \not\simeq b', D', D''$ by definition, we have $\mathcal{I}_i(C,S) \not\models_0 E$, which raises a contradiction.
   - If there are several literals that can be projected to $\bar{p_i}$ in $D$, the same disjunction of cases can be applied as with only one literal projected on $\bar{p_i}$, with the difference that in the second case, the $\mathcal{K}$-paramodulation is applied several times until a clause $E$ of the form $l_1 \vee \cdots \vee l_k \vee D' \vee \cdots \vee D' \vee D'' \vee \cdots \vee D''$ with $l_1, \ldots, l_k$ negative literals equationally entailing $C$. The same contradiction can then be raised on $E$.

3. In the case where there is no literal in $D$ that can be projected to $\bar{p_i}$ on $C$, there are again two possibilities to consider:

   - If $l$ is the only atom in $D$ such that $l_{\downarrow C} = p_i$, then we write $D = D' \vee l$ with $D' <_C p_i$. If $\mathcal{I}_{i-1}(C,S) \models_0 D'$, then by definition $\mathcal{I}_i(C,S) \models_0 D$. In the other case, $\mathcal{I}_{i-1}(C,S) \not\models_0 D'$ and if $p_i \in C_{\downarrow}$, then $l \models_{\mathcal{E}} C$ by Corollary 28. Hence by definition of $<_C$, for all literals $l' \in D'$, necessarily $l' \models_{\mathcal{E}} C$, meaning that $D \models_{\mathcal{E}} C$ which contradicts our hypothesis. Thus $p_i \notin C$ and $l$ verifies Condition 3b of Definition 70, hence $\mathcal{I}_i(C,S) \models_0 l$.
   - Assume there are two maximal atoms in $D$ that are projected to $p_i$ on $C$, then we write $D = l \vee l' \vee D'$ with $D' <_C l, l'$. The proof is similar if there are more than two maximal atoms (by applying several time the $\mathcal{K}$-equational factorisation rule instead of just once). Let $l$ be of the form $a \simeq b$, $l'$ be of the form $a' \simeq b'$, with $a_{\downarrow C} = {a'}_{\downarrow C}$ and $b_{\downarrow C} = {b'}_{\downarrow C}$. Let $E = a \not\simeq a' \vee b \not\simeq b' \vee l \vee D'$. By $\mathcal{K}$-equational factorisation, $D \vdash_{\mathcal{K}} E$. Since $S$ is closed, $E$ is a tautology or $E \in S$. If $E$ is a tautology, then $\mathcal{I}_i(C,S) \models_0 E$, and since $\mathcal{I}_i(C,S) \not\models_0 a \not\simeq a'$, $\mathcal{I}_i(C,S) \not\models_0 b \not\simeq b'$ and

$\mathcal{I}_i(C,S)\not\models_0 D'$, necessarily $\mathcal{I}_i(C,S)\models_0 l$, thus $\mathcal{I}_i(C,S)\models_0 D$. If $E \in S$, we go back to the previous case to obtain the result.

Finally, we prove that the restriction of $\mathcal{I}_i(C,S)$ to the atoms lower or equal to $p_i$ (with regard to $<_C$) is an equational interpretation By induction, it is already true for the literals that are strictly lower than $p_i$, since $\mathcal{I}_i(C,S)$ coincide with $\mathcal{I}_{i-1}(C,S)$ on those literals. Let $l$ be a literal such that $l_{\downarrow C} = p_i$.

- Reflexivity: if $l$ is a tautology, then $p_i$ is one too, so $\mathcal{I}_i(C,S)\models_0 l$.
- Commutativity: since we identify $a \simeq b$ and $b \simeq a$, it is naturally respected by $\mathcal{I}_i(C,S)$.
- Transitivity: if $l = a \simeq b$, $\mathcal{I}_i(C,S)\models_0 l$ and $\mathcal{I}_i(C,S)\models_0 a \simeq c$ with $c \prec b$, then we prove $\mathcal{I}_i(C,S)\models_0 b \simeq c$.

  1. If $(a \simeq b)_{\downarrow C}$ and $(a \simeq c)_{\downarrow C}$ are both of the form $a_{\downarrow C} \simeq a_{\downarrow C}$, then $(b \simeq c)_{\downarrow C}$ is too, thus $\mathcal{I}_i(C,S)\models_0 b \simeq c$.
  2. Assume $(a \simeq b)_{\downarrow C}$ is of the form $a_{\downarrow C} \simeq a_{\downarrow C}$, $(a \simeq c)_{\downarrow C} \notin C$ and there exists a clause $D \vee a' \simeq c' \in S$, such that $(a \simeq c)_{\downarrow C} = (a' \simeq c')_{\downarrow C}$, with $D <_C (a' \simeq c')$ and $\mathcal{I}\not\models_0 D$. Since $b_{\downarrow C} = a_{\downarrow C}$, the inequality $D <_C (b \simeq c)$ holds, and $(b \simeq c)_{\downarrow C} = (a' \simeq c')_{\downarrow C}$, thus $\mathcal{I}_i(C,S)\models_0 b \simeq c$.
  3. The same reasoning applies if $(a \simeq c)_{\downarrow C}$ is of the form $a_{\downarrow C} \simeq a_{\downarrow C}$ and $(a \simeq b)_{\downarrow C} \notin C$.
  4. Assume $(a \simeq b)_{\downarrow C} \notin C$, $(a \simeq c)_{\downarrow C} \notin C$, and there exist two clauses $D \vee d \simeq e$ and $D' \vee d' \simeq f$ both in $S$ such that $(a \simeq b)_{\downarrow C} = (d \simeq e)_{\downarrow C}$, $(a \simeq c)_{\downarrow C} = (d' \simeq f)_{\downarrow C}$ (w.l.o.g. $a_{\downarrow C} = d_{\downarrow C} = d'_{\downarrow C}$, $b_{\downarrow C} = e_{\downarrow C}$ and $c_{\downarrow C} = f_{\downarrow C}$), $D <_C (a \simeq b)$, $D' <_C (a \simeq c)$, $\mathcal{I}_{i-1}(C,S)\not\models_0 D$ and $\mathcal{I}_{i-1}(C,S)\not\models_0 D'$. We denote $E = d \not\simeq d' \vee e \simeq f \vee D \vee D'$; in this case, $S \vdash_{\mathcal{K}} E$ and $E$ is a tautology or $E \in S$ by saturation. If $E$ is a tautology, then $\mathcal{I}_i(C,S)\models_0 E$; and if $E \in S$:
     - if $e \simeq f <_C l$ then $E <_C l$ and by induction $\mathcal{I}_i(C,S)\models_0 E$,
     - if $(e \simeq f)_{\downarrow C} = l_{\downarrow C}$, then $f_{\downarrow C} = a_{\downarrow C}$, thus $c_{\downarrow C} = a_{\downarrow C}$ and we are not in Case 4 but in Case 3 so the result holds,
     - if $l <_C e \simeq f$, then we are outside of the bounds of what we want to prove, so this case can be ignored.

     Hence, in the case considered here, we always have $\mathcal{I}_i(C,S)\models_0 E$. Moreover, $\mathcal{I}_i(C,S)\not\models_0 D, D', d \not\simeq d'$ by definition, thus, necessarily $\mathcal{I}_i(C,S)\models_0 e \simeq f$. Since $(e \simeq f)_{\downarrow C} = (b \simeq c)_{\downarrow C}$, the two literals $e \simeq f$ and $b \simeq c$ both verify the conditions for which $\mathcal{I}_i(C,S)$ is a model of $e \simeq f$, hence $\mathcal{I}_i(C,S)\models_0 b \simeq c$.
  5. The last case necessary to prove transitivity has different hypotheses. It corresponds to one of the cases that were rejected as out of bound in the previous point. In this cases, the two atoms $a \simeq c$ and $b \simeq c$ are such that $\mathcal{I}_i(C,S)\models_0 a \simeq c$, $\mathcal{I}_i(C,S)\models_0 b \simeq c$ and $a \simeq c, b \simeq c <_C p_i$

with $p_i = (a \simeq b)_{\downarrow C}$. We show that $\mathcal{I}_i(C,S) \models_0 a \simeq b$. We consider only the case where $(a \simeq c)_{\downarrow C} \notin C$ and $(b \simeq c)_{\downarrow C} \notin C$, because the others are similar to the points 1, 2 and 3 of this proof. By hypothesis, there exist two clauses $D \vee d \simeq f$ and $D' \vee e \simeq f'$ with $d_{\downarrow C} = a_{\downarrow C}$, $e_{\downarrow C} = b_{\downarrow C}$ and $f_{\downarrow C} = {f'}_{\downarrow C} = c_{\downarrow C}$, such that $D <_C d \simeq f$, $D' <_C e \simeq f'$, $\mathcal{I}_i(C,S) \not\models_0 D$ and $\mathcal{I}_i(C,S) \not\models_0 D'$. By saturation, the clause $E = f \not\simeq f' \vee d \simeq e \vee D \vee D'$ is a tautology or $E \in S$. Moreover, $\max <_C E = p_i$, thus, in the case where $E \in S$, as proved previously in this demonstration, $\mathcal{I}_i(C,S) \models_0 E$. Obviously, if $E$ is a tautology, then $\mathcal{I}_i(C,S) \models_0 E$ is also true. By hypothesis $\mathcal{I}_i(C,S) \not\models_0 D$, $\mathcal{I}_i(C,S) \not\models_0 D'$ and by definition $\mathcal{I}_i(C,S) \not\models_0 f \not\simeq f'$, thus $\mathcal{I}_i(C,S) \models_0 d \simeq e$. As in Point 4, this allows us to conclude that $\mathcal{I}_i(C,S) \models_0 a \simeq b$. ∎

# Conclusion

From a study of prime implicates generation in propositional logic, we have selected an algorithm and adapted it into a method for prime implicate generation in ground flat equational logic. This method is composed of two main parts. One is the storage of ground flat equational clauses modulo equivalence, through the definition of a new and relatively compact data structure, the $\mathcal{N}$-clausal tree. To manipulate an $\mathcal{N}$-clausal tree, we have also provided algorithms that allow to update this data structure and get rid of the redundancies it may contain, and we proved the soundness and termination of these algorithms. The other part of the method deals with the generation of the implicates that will be stored in the $\mathcal{N}$-clausal tree. We have devised a variation of the superposition calculus, the $\mathcal{K}$-paramodulation calculus, that can generate all the implicates of a formula in ground flat equational logic, and we proved that it is complete, namely that any implicate $C$ of a formula $S$ is entailed by a clause $D$ generated from $S$ by the calculus.

There are still many interesting problems to solve and we are currently working on some of them. A natural way to improve the calculus would be to handle normal clauses instead of standard ones. Since equivalent clauses all have the same normal form, this would significantly reduce the number of clauses to consider. Another interesting improvement would be to define a redundancy criterion for clauses preserving the completeness of the calculus. Handling normal clauses instead of standard ones will require a modification of the completeness proof, and for the time being, a suitable redundancy criterion has proved difficult to find: the ones being examined are either not restrictive enough or do not preserve the completeness of the calculus. However we believe that these problems will be solved fairly quickly. It can also be noted that the current version of the $\mathcal{K}$-paramodulation calculus has no ordering restrictions whatsoever. Even though this lack of order constraints is in some ways unavoidable to ensure the generation of all implicates, it is clear that enforcing some partial ordering conditions that preserve the completeness of the $\mathcal{K}$-paramodulation calculus would also improve its efficiency. We intend to investigate in detail what kind of ordering conditions can be imposed.

This theoretical work will be followed by an implementation of the whole algorithm, integrating both an efficient generation of implicates and their storage, which will allow us to ascertain the efficiency of our method and verify how it scales up to concrete problems.

There are many interesting ways to extend the research in this area in the long term. The first two directions we intend to explore are the following: One is to adapt other algorithms from propositional logic, and more specifically those based on the decomposition method, to which one is the more efficient. Another is to investigate the possible extensions of the method described here to more expressive logics, first by considering ground abducible terms and not simply abducible constants thus working with non flat clauses and afterwards considering non-ground abducible terms and clauses.

# Bibliography


L. Bachmair, H. Ganzinger, and U. Waldmann. Refutational theorem proving for hierarchic first-order theories. *Applicable Algebra in Engineering, Communication and Computing*, 5(3):193–212, 1994.

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.

G. Bittencourt. Combining syntax and semantics through prime form representation. *Journal of Logic and Computation*, 18(1):13, 2008.

O. Coudert and J. Madre. A new method to compute prime and essential prime implicants of boolean functions. In *Proc. of MIT VLSI Conference*, 1992.

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.

J. de Kleer and R. Reiter. Foundations for assumption-based truth maintenance systems: Preliminary report. In *Proc. American Assoc. for Artificial Intelligence Nat. Conf*, pages 183–188, 1987.

M. Echenim and N. Peltier. A calculus for generating ground explanations (technical report). *Arxiv preprint arXiv:1201.5954*, 2012.

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.

E. Fredkin. Trie memory. *Commun. ACM*, 3(9):490–499, 1960. ISSN 0001-0782. doi: http://doi.acm.org/10.1145/367390.367400.

L. Henocque. The prime normal form of boolean formulas. *Technical report at http://www.Isis.org/fiche.php*, 2002.

P. Jackson. Computing prime implicates incrementally. *Automated Deduction CADE-11*, pages 253–267, 1992.

P. Jackson and J. Pais. Computing prime implicants. In *10th International Conference on Automated Deduction*, pages 543–557. Springer, 1990.

A. Kean and G. Tsiknis. An incremental method for generating prime implicants/implicates. *Journal of Symbolic Computation*, 9(2):185–206, 1990.

A. Leitsch. *The resolution calculus*. Springer. Texts in Theoretical Computer Science, 1997.

V. Manquinho, A. Oliveira, and J. Marques-Silva. Models and algorithms for computing minimum-size prime implicants. In *Proceedings of the International Workshop on Boolean Problems*, 1998.

J. Marques-Silva and K. Sakallah. Grasp: A search algorithm for propositional satisfiability. *Computers, IEEE Transactions on*, 48(5):506–521, 1999.

P. Marquis. Extending abduction from propositional to first-order logic. In *Fundamentals of artificial intelligence research*, pages 141–155. Springer, 1991.

A. Matusiewicz, N. Murray, and E. Rosenthal. Prime implicate tries. *Automated Reasoning with Analytic Tableaux and Related Methods*, pages 250–264, 2009.

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.

M. Mayer and F. Pirri. Abduction is not deduction-in-reverse. *Logic Journal of IGPL*, 4(1):95–108, 1996.

T. 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.

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.

W. Quine. A way to simplify truth functions. *The American Mathematical Monthly*, 62(9):627–631, 1955.

A. Ramesh. *Some applications of non clausal deduction*. PhD thesis, 1995.

A. Ramesh, G. Becker, and N. Murray. Cnf and dnf considered harmful for computing prime implicants/implicates. *Journal of Automated Reasoning*, 18 (3):337–356, 1997.

R. Rymon. An se-tree-based prime implicant generation algorithm. *Annals of Mathematics and Artificial Intelligence*, 11(1):351–365, 1994.

M. Shanahan. Prediction is deduction but explanation is abduction. In *IJCAI 89*, pages 1055–1060, 1988.

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.

J. Slagle, C. Chang, and R. Lee. A new algorithm for generating prime implicants. *Computers, IEEE Transactions on*, 100(4):304–310, 1970.

T. Strzemecki. Polynomial-time algorithms for generation of prime implicants. *Journal of Complexity*, 8(1):37–63, 1992.

P. Tison. Generalization of consensus theory and application to the minimization of boolean functions. *Electronic Computers, IEEE Transactions on*, (4): 446–456, 1967.