Imperial College of Science,

Technology and Medicine

(University of London)

Department of Computing

Theorem Proving Using NAND Expressions

by

Marta M. F. Chaves

Submitted in partial fulfilment

of the requirements for the MSc

Degree in Computing Science of the

University of London and for the

Diploma of Imperial College of

Science, Technology and Medicine.

September 2006

Abstract

The NAND system is a theorem prover derived from an original idea by Sandqvist Tor. It can be applied to propositional and first order logic. The major difference from other proof systems resides on its syntactic normal form, which is the Bracket Nand Normal Form (BNNF). Logical formulas are represented as lists using only two operators that play the role of negation and conjunction. When translated into BNNF, the usual distributivity laws of boolean logic are merged into a single one. This is because the bracket operator has the ability of metamorphosing between the conjunctive and disjunctive operators.

A semantic tree distribution method using BNNF expressions that uses a single extension rule is defined in similarity to NNF tableau methods. When combined with a set of simplification-reduction rules the proof of a theorem can be accelerated. Finally, some of the techniques defined for NAND are related to recent techniques in knowledge compilation systems.

Acknowledgments

This project was only possible with the help of my supervisor Dr. Krysia Broda, to whom I am deeply grateful. All the way she has been extremely enthusiastic, dedicated and patient with me, and I am definitely very happy to have had her as my supervisor.

I would also like to thank Dr. Silvana Zappacosta for being my second marker in this project.

Some material implied in this project was kindly provided by Driss, to whom I am sincerely grateful.

Nuno has been my biggest supporter for doing this M.Sc. in Computing Science at Imperial, and has encouraged me all along the way. Without him I would never have done this. My eternal thanks to him.

My mother, my father, my sister and nephew(s) have always been present in my mind during this whole year. I know they have also supported me in mind and heart, as always. My deepest thoughts and *saudades* go to them.

And I could not forget all the classmates that have shared this stressful happy times since October. I will not mention names not to leave anyone out, but I wish them all good luck. Hope to meet again soon!

To Nuno

Contents

1 Motivations and structure **1**

2 Background **5**

2.1 Proof Systems 5

2.1.1 Tableau Methods 5

2.1.2 Davis Putnam Procedure 8

2.2 Normal Forms 9

2.2.1 Ordered Binary Decision Diagrams 9

2.2.2 Decomposable Negation Normal Form 13

2.2.3 Path Dissolution 23

2.2.4 Factored Negation Normal Form 29

3 The NAND Formalism **33**

3.1 Syntax and Semantics 34

3.2 General Properties 38

3.3 Simplification Rules 41

3.4 Tree Expansion 52

3.4.1 Branch Reductions 62

3.4.2 Soundness and Completeness 65

3.5 Application to Propositional Logic 68

3.5.1 Compiling NAND into DNNF 68

3.5.2 Testing 75

3.6 Application to First-Order Logic 77

4 Conclusions and Perspectives **85**

Chapter 1

Motivations and structure

Automated theorem proving is a well developed field of automated reasoning in artificial intelligence that uses deductive reasoning to derive conclusions about formal logical sentences. The *automated* attribute means that the reasoning process is programmed and done by a machine, and not by humans. Of course, up to now, it is still the human's task to develop the program. Applications of theorem proving range from pure mathematics, with the discovery of proofs for mathematical conjectures, to applied areas like software and hardware implementation and verification.

The theorem proving landscape is densely populated. Currently established methods for classical logic vary in their applicability domain (propositional logic, first-order logic, etc.), their representation (clausal/non-clausal forms, semantic trees, semantic graphs, decision diagrams, etc.) and, most importantly, in the proof strategies they employ (choice of inference rules, sentence orderings, etc.). Traditional renowned methods are Smullyan's analytic tableaux [25], Mondadori's KE-tableaux [2], the Davis-Putnam-Logemann-Loveland procedure (DPLL) [5, 6], Ordered Binary Decision Diagrams (OBDD) [15] and, of course, Robinsons's resolution [22]. Tableau methods and resolution can be applied to both propositional and first-order logic, while DPLL and OBBD's cover the propositional domain. Other differences are that resolution and DPLL work with Conjunctive Normal Form (CNF), or clausal form for short, while tableaux can deal with Negation Normal Form (NNF). An extension of DPLL to non clausal form is described in [18]. OBDD's are themselves a canonical normal form that can be constructed for any boolean function.

Some of these techniques, originally developed for asserting the satisfiability of a theory, have later been rediscovered as powerful tools for knowledge compilation. While traditional proof systems concentrate on deciding whether a theory is valid or not and, perhaps, returning a counter model for the theory, knowledge compilation is concerned

with preprocessing the given theory so that it can be used in later queries. The goal is to optimise the theory representation so that it speeds up the query answering process. Obviously, as a bonus, the preprocessing stage also decides on the validity of the underlying theory. Although the compilation process may itself be slower than a general refutation procedure, once it is done it can be reused in many queries making it faster overall. Alongside, new normal form representations were introduced (DNNF [7, 8], OFNNF [11, 12], full dissolvents [16, 20, 17]) as an efficient representation basis for the propositional theories.

The NAND proof system we implement in this project is based on a new normal form originally introduced by Sandqvist [23], also explored by Lin Yang [26]. We call it the Bracket Nand Normal Form (BNNF). The representation is based on a generalisation of the NAND operator or Sheffer's stroke [24]. Logical formulas are represented as lists using only two operators that play the role of negation and conjunction. It is a versatile system that can reproduce NNF analytic or KE tableau methods using only one extension rule. It also includes a series of simplification rules that can be used alone or coupled to the tableau method to derive a proof of a logical expression or compile a theory into DNNF. The NAND simplification rules are equivalence rules that are also found in other proof and compilation systems. But, when translated into BNNF syntax, a single rule emulates well known inference rules like unit resolution, subsumption and factoring. This is due to the dual nature of the operator that is used to represent BNNF formulas, which is able to metamorphose between a conjunction and a disjunction as the logical formula is manipulated.

The best application of the NAND system is in the domain of propositional logic where the simplification rules can be freely applied. They play an important role in this system. The extension of NAND to first order logic is also achieved in terms of analytic tableaux style proofs.

Most of the systems we have mentioned in this introduction, including our NAND, are closely related. We shall overview some of them briefly, paying particular attention to knowledge compilation techniques which are object of relatively recent work which the reader may be less familiarised with. However, we do expect the reader to be familiar with the general concepts of propositional and first order logic.

The structure of this text is as follows. Chapter 2 contains the definition of analytic and KE NNF tableaux as well as the rules for the clausal DPLL. These are given in Section 2.1. Section 2.2 contains a description of the normal forms for knowledge compilation. The ones presented are the OBDD's in Section 2.2.1, DNNF's in Section 2.2.2, path dissolution in Section 2.2.3 and FNNF's in Section 2.2.4.

The NAND system and its applications are introduced in Chapter 3. The syntax and

semantics for BNNF logical formulas is presented in Section 3.1, where the bracket operator is defined. Properties like *depth*, *size* and *degree* and the distributivity rule are defined in Section 3.2. Section 3.3 contains the set of simplification rules, namely bracket simplifications, formula reductions and Shanon expansion. The tree expansion method that is similar to tableau methods is given in Section 3.4. Section 3.5 gives an application of the NAND system as a compilation method into DNNF normal form, and also contains the tests that where performed against the LeanTap [27] implementation of analytic tableaux. Finally, the NAND system for the first order logic is described in Section3.6.

Conclusions are summarised in Chapter 4.

Chapter 2

Background

2.1 Proof Systems

2.1.1 Tableau Methods

Analytic tableau methods are refutation procedures with application in propositional and first order logic. Here we will describe the propositional case. A good reference for tableau methods is the Smullyan book [25].

The tableau method can be defined for general logical form using all the boolean connectives, for *negation normal from* (NNF) and for *conjunctive normal form* (CNF). Negation normal form represents logical formulas using only the connectives $\wedge$, $\vee$ and $\neg$, where $\neg$ must be at the literal level. Conjunctive normal form (or *clausal form*) is defined as a conjoined set of disjoined literals as, for instance $(a \vee b \vee c) \wedge (b \vee d) \wedge (a \vee d)$. There is also another definition of tableaux that uses *signed* formulas, but we will not cover them here. We shall focus on the propositional NNF case.

During the construction of a tableau, the initial formula or formula set is decomposed by successively splitting the formulas at each connective $\wedge$ or $\vee$, until the literal level is reached. This splitting into cases draws a tree structure where each node is marked with a (sub)formula. A formula that belongs to a branch can be used to extend that branch according to the tableaux rules:

$$\frac{\alpha_1 \wedge \alpha_2}{\begin{array}{c}\alpha_1\\ \alpha_2\end{array}} \qquad \frac{\beta_1 \vee \beta_2}{\beta_1 \mid \beta_2} \qquad \frac{\begin{array}{c}p\\ \neg p\end{array}}{\times} \tag{2.1.1}$$

From left to right, the three rules are known as the α rule, β rule and closure rule.

Their interpretation is the following.

α **rule** If in a node of a given branch there exists a formula of the form $\alpha_1 \wedge \alpha_2$, then that branch can be extended by attaching both α_1 and α_2 below it;

β **rule** If in a node of a given branch there exists a formula of the form $\beta_1 \vee \beta_2$, then we can add a split at the end of that branch and attach each β_i as a leaf node;

Closure rule If a branch contains to complementary literals p and $\neg p$, then close that branch. This branch cannot be extended.

Each formula can be selected only once along each branch. Only formulas that belong to a branch can be selected for the extension of that branch. If all the branches end up closed, it means that the initial formula $\mathcal{F}$ was unsatisfiable. If there are no more formulas to be selected and some of the branches are left open, then $\mathcal{F}$ is satisfiable. It does not mean that $\mathcal{F}$ is a tautology, it only means that there are true or false assignments to the atoms of $\mathcal{F}$ that make it satisfiable. In this case we say that there is an *interpretation* I that satisfies $\mathcal{F}$. We also say that $\mathcal{F}$ is unsatisfiable if there is no interpretation that satisfies it. In other words, under all possible assignments over atoms of $\mathcal{F}$, $\mathcal{F}$ is always false.

We can see why in this way the tableau is a refutation procedure. If we negate the original theorem $\mathcal{T}$ and draw a tableau for it, and the tableau closes, it means that $\neg\mathcal{T}$ was unsatisfiable. Consequently, $\mathcal{T}$ is valid, i.e., it is a tautology.

Example 2.1.1 *Consider the following tautology*

$$\mathcal{F} \equiv (p \vee (q \wedge r)) \Rightarrow ((p \vee q) \wedge (p \vee r)) \tag{2.1.2}$$

with $\mathrm{NNF}(\neg\mathcal{F})$ *given by*

$$\mathrm{NNF}(\neg\mathcal{F}) = (p \vee (q \wedge r)) \wedge ((\neg p \wedge \neg q) \vee (\neg p \wedge \neg r)) \tag{2.1.3}$$

A closed tableau for $\mathrm{NNF}(\neg\mathcal{F})$ *is given in Figure 2.1.*

Firstly, the α rule is applied to $\mathrm{NNF}(\neg\mathcal{F})$ *and the two formulas are placed at the root of the tableau. Then the β rule is applied to $(p \vee (q \wedge r))$ splitting the branch into two, a leaf marked with p and another leaf marked with $(q \wedge r)$. Next, the second formula at the root node is select and the β rule is applied to it, under node p. At the same time, on the right branch, we apply the α rule on $q \wedge r$, which extends that branch without splitting. The second formula on the root is also selected to extend this branch with a split under the leaf node r. Finally, α rules are applied at each leaf, closing all the branches because they all contain complementary literals.*

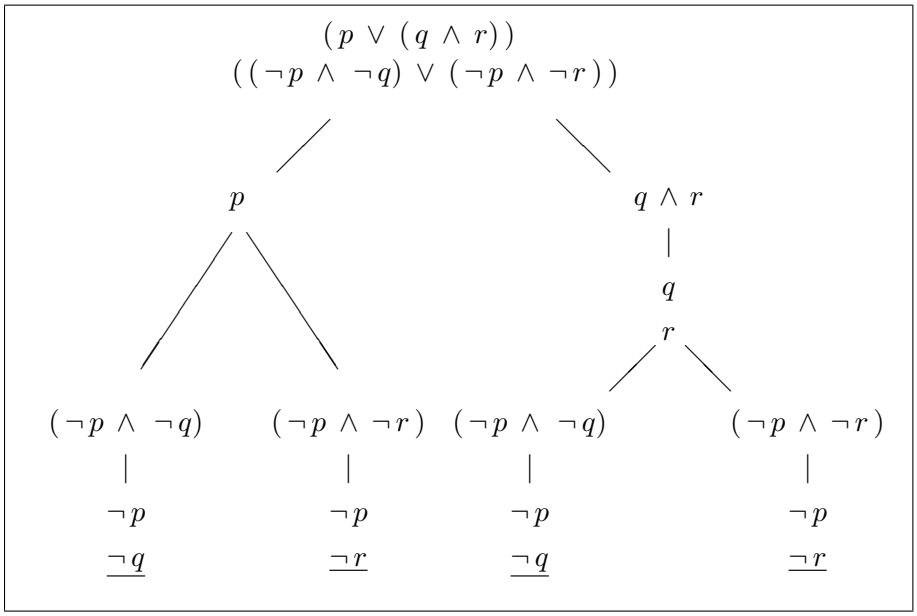


Figure 2.1: Closed Tableau

Note that even if the tableau of Figure 2.1 was not closed it would be *complete*, because no more formulas were left for selection.

Definition 2.1.2 (Complete Tableau) *A tableau is complete if it is closed or if, for every branch θ, every compound (non literal) formula in θ has been used for extending θ.*

When a tableau is completed, the result of conjoining all literals along each branch and disjoining the branches gives a *disjunctive normal form* (DNF) representation of the root node. DNF is the dual of CNF. It is a disjoined set of conjoined literals. The closed branches of a tableau correspond to those conjunctions in the DNF that result in contradictions.

Analytic tableaux are only allowed to pick formulas that exist in the branch being extended. Another type of tableaux, called the KE-tableaux [1, 2] only performs splitting on literals. A split is performed by adding two nodes marked with p and $\neg p$ as leaf nodes of the split. The literal is chosen form the atom set of formulas in that branch . This rule is called the *principle of bivalence* (PB) rule.

The rules for NNF KE-tableaux are[1]:

[1]We have used the same notation as in [11].

$$\frac{}{p \mid \neg p} \qquad \frac{\alpha_1 \wedge \alpha_2}{\begin{array}{c}\alpha_1\\ \alpha_2\end{array}} \qquad \frac{\begin{array}{c}\neg\,\beta_1\\ \beta_1 \vee \beta_2\end{array}}{\beta_2} \qquad \frac{\begin{array}{c}\neg\,\beta_2\\ \beta_1 \vee \beta_2\end{array}}{\beta_1} \qquad \frac{\begin{array}{c}p\\ \neg\, p\end{array}}{\times} \tag{2.1.4}$$

The PB rule can be found again in the Davis Putnam procedure under slightly different surroundings.

2.1.2 Davis Putnam Procedure

The Davis Putnam procedure (DP) is a decision procedure originally developed in 1960 by Martin Davis and Hillary Putnam [5] for clausal propositional theories[1].

Later refinements with contributions by Logemann and Loveland [6] improved on the original algorithm which is now known as the Davis-Putnam-Logemann-Loveland procedure (DPLL).

The algorithm is very simple and yet, or because of that, very efficient. Its is based on refutation, which means that the initial theorem is negated and an attempt to derive a contradiction is made.

The formal steps are the following. Given a clause set[2] $\mathcal{S}$:

Tautology Rule Remove all tautologous clauses from $\mathcal{S}$, i.e., those that contain a literal and its complement in the form $(\, L \vee \overline{L}\,)$.

One Literal Rule For each unit clause $L \in \mathcal{S}$: i) Delete all clauses containing L; ii) Delete all occurrences of $\overline{L}$ from the remaining clauses.

Pure Literal Rule If a literal $L \in \mathcal{S}$ but $\overline{L} \notin \mathcal{S}$, delete all clauses containing L. L is a *pure literal*.

Termination Rule If the empty clause ($false$) is generated, then there is a refutation. Otherwise, if no clauses are left, then $\mathcal{S}$ is satisfiable.

Splitting Rule If none of the above cases apply, select a literal L and split the clause set into two sets $\mathcal{S}_L$ and $\mathcal{S}_{\overline{L}}$. $\mathcal{S}_L$ is the clause set that results from $\mathcal{S}$ after all clauses containing L have been deleted and all occurrences of $\overline{L}$ removed from the remaining

[1] The Davis-Putnam procedure can also be applied to first order logic (FOL) using *Herbrand models* [25]. MACE [14], a DPLL theorem prover, still uses this technique.

[2] A clause is a disjunction of literals, and a set of clauses is a conjunction of clauses.

clauses. Conversely, $\mathcal{S}_{\overline{L}}$ is the clause set obtained form $\mathcal{S}$ after removing all clauses containing $\overline{L}$ and removing all occurrences of L from the other clauses.

Below we give an example were this method is applied.

Example 2.1.3 *Consider the clause set*

$$S = \{(\neg q \vee r),\ (\neg r \vee p),\ (\neg r \vee q),\ (\neg p \vee q \vee r),\ (p \vee q),\ (\neg p, \vee \neg q)\} \quad (2.1.5)$$

There are no unit clauses neither pure literals , so we select the literal p to apply the splitting rule. Then

$$S_p = \{(\neg q \vee r),\ (\neg r \vee q),\ (q \vee r),\ (\neg q)\} \quad (2.1.6)$$

and

$$S_{\overline{p}} = \{(\neg q \vee r),\ (\neg r),\ (\neg r \vee q),\ (q)\} \quad (2.1.7)$$

we now apply to S_p the unit literal rule on $\neg q$, and apply to $S_{\overline{p}}$ the unit literal rule on $\neg r$ to obtain

$$S_p = \{(\neg r),\ (r)\} \quad (2.1.8)$$

and

$$S_{\overline{p}} = \{(\neg q),\ (q)\} \quad (2.1.9)$$

And finally apply the unit literal rules on r and q which generates the empty clause on both S_p and $S_{\overline{p}}$. Since each set contains the empty clause, the proof has succeeded. In other words, the given clause set is unsatisfiable. If some of the clause sets had become empty, then the derivation would have failed.

There is also an extension of DPLL to non-clausal form, known as the non-clausal Davis-Putnam (NCDP) that can be found in [18].

2.2 Normal Forms

2.2.1 Ordered Binary Decision Diagrams

One of the applications of automated reasoning is hardware and software verification. *Ordered binary decision diagrams* (OBBD) are a canonical representation of Boolean functions that uses directed acyclic graphs. Satisfiability and equivalence testing between

boolean functions becomes very efficient using OBDD's. For that reason they have been widely implied in digital system design, verification and testing.

A comprehensive overview of OBBD's and its *IF normal form* representation is presented by Moore in [15]. The description we shall give here, including theorems and formal definitions, are based on his paper.

The Boyer-Moore logic represents boolean propositional variables by *IF-THEN-ELSE* statements. If we want to say, for example, that X is *true*, we can say it as *IF-X-THEN-TRUE-ELSE-FALSE*. The shorthand notation for the IF statements is

$$\text{(IF X Y Z)} \quad \equiv \quad \text{IF X THEN Y ELSE Z} \tag{2.2.10}$$

The IF axioms are

Axiom 2.2.1 *(IF T Y Z) = Y*

Axiom 2.2.2 *(IF F Y Z) = Z*

where T and F are constant symbols. We may intuitively recognise them as *true* and $false$. Since the IF variables are boolean, they are also mapped onto T or F.

A set of four theorems is needed for manipulating the normal form. Although they are all important, the last two are the "main ingredients".

Theorem 2.2.3 (IF-X-T-F) *(IF X T F) = X*

Theorem 2.2.4 (IF-X-Y-Y) *(IF X Y Y) = Y*

Theorem 2.2.5 (Reduction) *Consider the expression (IF X Φ Ψ), where X is a boolean variable. Then every occurrence of X in Φ can be replaced by T and every occurrence of X in Ψ may be replaced by F.*

We stop here for a brief moment to pay attention to this theorem. As we shall see when we get to the NAND formalism, this is the theorem that inspired the $\top - \bot$ reduction rules of Theorems 3.3.4 and 3.3.5 that are used for BNNF[1] formula simplifications. The justification for it is that once we analyse the functions Φ or Ψ, the variable X has already been tested and its value is known. It can be either T or F. If it is T we go into the *then*-branch, if X is F we go into the *else*-branch. Therefore, if we are in the *then*-branch X must have been true, and conversely for the *else*-branch.

[1]Bracket Nand Normal Form (BNNF) is defined in Chapter 3.

Example 2.2.6 *Consider the IF expression* (IF X (IF Y X F) (IF X Z Y)). *By the reduction theorem, X can be replaced by* T *in* (IF Y X F) *and by F in (*IF X Z Y). *The resulting IF expression is* (IF X (IF Y T F) (IF F Z Y)) *which later reduces to* (IF X Y Y) *by Theorem 2.2.3 and Axiom 2.2.2, and then to* Y *by Theorem 2.2.4.*

The theorem that follows is also extremely relevant. In propositional logic it is basically equivalent to what is known as Shanon expansion (SE)[1]. We have already seen an application of SE which is the *splitting rule* of the Davis Putnam procedure and, as we shall see further, it is also used in the three other normal forms described in this chapter. Furthermore, it can also be used in the NAND system for formula simplifications.

Theorem 2.2.7 (IF Distribution) *Let f_n be a function symbol acting on a set of variables $\{A_1, \ldots, An\}$ where one of them is an (IF X Y Z) expression. The IF-distribution over the function symbol f_n is given by:*

$$\begin{aligned} (\, f_n \;\; A_1 \ldots (\,\mathrm{IF\ X\ Y\ Z}\,) \ldots A_n\,) \quad = \quad & (\,\mathrm{IF\ X} \\ & (\, fn \;\; A_1 \ldots \mathrm{Y} \ldots A_n\,) \\ & (\, fn \;\; A_1 \ldots \mathrm{Z} \ldots A_n\,)\ ) \end{aligned} \tag{2.2.11}$$

What the distribution theorem is doing is splitting into cases. The case when X is T and the case where X is F. For each case the reduction theorem applies.

The IF representation defined so far is not yet normalised. The IF *normal form* sets a few extra rules as defined below.

Definition 2.2.8 (IF Normal Form) *The constant symbols T and F are in IF normal form. An expression (IF X Y Z) is in IF normal form if*

- *X contains no IF's and is neither T nor F*
- *X does not occur in Y or Z*
- *Y and Z are not identical*
- *Y and Z are in IF normal form*

No other expressions are in IF normal form.

To normalise a given IF expression the above theorems and axioms should be applied recursively. Here is an example that combines the several rules.

[1]See equations (2.2.21), (3.3.18) or (3.3.19)

Example 2.2.9 *Consider the IF expression* (IF (IF X Y T) Z (IF Z F Y)). *First of all we see that the test expression (IF X Y T) contains IF's. IF-distribution is applied on X:*

$$\begin{array}{lll} (\text{ IF X} & (\text{ IF Y Z (IF Z F Y)) } \\ & (\text{ IF T Z (IF Z F Y))) } \end{array} \tag{2.2.12}$$

Next, we apply the Axiom 2.2.1 and reduction on Y

$$(\text{ IF X} \quad (\text{ IF Y Z (IF Z F F)) } \quad (\text{ Z)) } \tag{2.2.13}$$

followed by the IF-X-Y-Y Theorem

$$(\text{ IF X} \quad (\text{ IF Y Z F) } \quad (\text{ Z)) } \tag{2.2.14}$$

Finally, theorem IF-X-T-F is used to replace the single variable Z

$$(\text{ IF X} \quad (\text{ IF Y (IF Z T F) F) } \quad (\text{ IF Z T F)) } \tag{2.2.15}$$

The OBDD corresponding to this IF canonical form is shown In Figure 2.2.

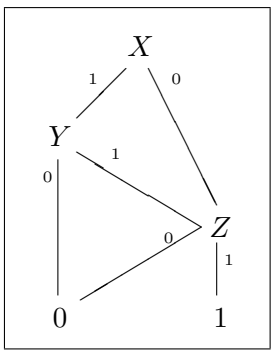


Figure 2.2: OBDD

Distributivity usually increases the size of the function because each expansion creates a duplicate. But the order of variables upon which distribution is made can be fixed according to some predefined criteria. When the IF normal form variables are ordered along all branches based on that criteria, then we have an IF *canonical form*. In the previous example, for instance, if the chosen variable ordering was $X < Y < Z$ then the result we obtained is already in canonical form. In this context, canonical means that for each variable ordering there is a unique representation of any boolean function. For

example, under the ordering $X < Y$, "X AND Y" can be represented as (IF X (IF Y T F) F). We leave to the reader the normalisation of the other connectives.

OBDD's are represented as decision diagrams (directed acyclic graphs) which have direct correspondence to IF canonical forms. Graphs are handled with less redundancy then binary decision trees because they can store identical subgraphs only once. The canonical property of OBDDs is what makes equivalence testing efficient. With that regard, OBDD's are more efficient then DNNF's, the normal form we will describe next.

2.2.2 Decomposable Negation Normal Form

Decomposable Negation Normal Form (DNNF) has been recently introduced by Darwiche [7, 8] as a formal compilation language for propositional theories. The main advantage over the existing OBDD's is that it is more space efficient, which is one of the major concerns in any compilation technique. The other two evaluation factors are *universality* and *tractability*. Universality is defined as the capability of providing a representation for any propositional theory, while the degree of tractability gives a measure on the logical operations that can be performed in polynomial time under this language. As shown by Darwiche, DNNF is universal and supports many operations — *satisfiability*, *conjoining*, *projecting*, computing *minimum-cardinality*, amongst others — in linear time. However, DNNF's are not canonical forms because they do not provide unique representations of boolean functions. As a consequence, equivalence testing is less efficient than for OBDD's.

The class of DNNF formulas is a sub-class of the more general NNF's[1] and its formulas obey the *decomposability property* defined below. The definitions and general description presented here is based on Darwiche's' papers. A few examples were added that were not in the original papers.

Definition 2.2.10 *A logical formula $\mathcal{F}$ is said to be in decomposable negation normal form (DNNF) if the following two conditions are satisfied:*

1. *$\mathcal{F}$ is in NNF*
2. *$\mathcal{F}$ satisfies the decomposability property: for every conjunction $\alpha \in \mathcal{F}$, such that $\alpha = \bigwedge_{k=1}^{n} \alpha_k$, no atoms are shared between any two conjuncts α_i, α_j $(i, j = 1, \ldots, n$ and $i \neq j$,), ie atoms(α_i) $\bigcap$ atoms(α_j) $= \emptyset$*

[1] Negation normal form represents logical formulas using only the connectives $\wedge$, $\vee$ and $\neg$, where $\neg$ must be at the literal level. *Conjunctive normal form* (CNF) is defined as a conjoined set of disjoined literals $(\ell_1 \vee \ldots \vee \ell m_1) \wedge (\ell_1 \vee \ldots \vee \ell m_2) \wedge \ldots \wedge (\ell_1 \vee \ldots \vee \ell m_p)$.

From this definition we observe that (reduced) DNF's[1] are themselves a sub-class of DNNF's, ie every DNF is automatically DNNF, but not the converse. We have appended the word "reduced" to point out that the DNF should be cleared of terms like $A \wedge \neg A$ or $A \wedge A$, which are obvious violations to the decomposability property. This is a soft assumption, and a reasonable one too, because for DNF's the variable A can only be a literal, thus making the simplification $A \wedge \neg A \simeq false$ and $A \wedge A \simeq A$ quite straightforward.

This is not the case for CNF's though, which are generally not DNNF's as we can see from the following example.

Example 2.2.11

1. *The formula $\mathcal{F} = (\, a \wedge (b \vee c)\,) \vee (b \wedge \neg a)$ is in DNNF because the two conjunctions appearing in $\mathcal{F}$ are $\wedge_1 = a \wedge (b \vee c)$ and $\wedge_2 = b \wedge \neg a$, and none of them shares atoms across its conjuncts $\wedge_{i,j}$:*

$$\left.\begin{array}{ll} \wedge_{1,1} \equiv a, & atoms(\wedge_{1,1}) = \{a\} \\ \wedge_{1,2} \equiv b \vee c, & atoms(\wedge_{1,2}) = \{b, c\} \end{array}\right\} \quad \{a\} \bigcap \{b, c\} = \emptyset$$

   and

$$\left.\begin{array}{ll} \wedge_{2,1} \equiv b, & atoms(\wedge_{2,1}) = \{b\} \\ \wedge_{2,2} \equiv \neg a, & atoms(\wedge_{2,2}) = \{a\} \end{array}\right\} \quad \{b\} \bigcap \{a\} = \emptyset$$

2. *The CNF formula $\mathcal{F} = (a \vee b) \wedge (\neg b \vee c)$ is not in DNNF because the atom b appears in both sides of the conjunction.*

3. *The formula $\mathcal{F} = (a \wedge b) \vee (b \wedge c) \vee (c \wedge a)$ is both in DNNF and DNF. Although all atoms are shared across the disjunctions, they are not shared across the conjunctions. But the equivalent formula $\mathcal{F}' = (a \wedge (b \vee c)) \vee (b \wedge c)$ is still DNNF but not DNF.*

The advantage brought by the decomposability property is that, for certain operations, each subformula becomes independent of the others. This means that the original problem can be split into a set of smaller independent problems in a way that is proportional to the size of the initial formula. This applies, for instance, when testing satisfiability. As we know, a conjunctive formula is only satisfiable if all of its conjuncts are. However, the converse is not necessarily true. Even when each argument is satisfiable on its own, their

[1] Disjunctive normal form (DNF) is the dual of CNF and represents a disjoined set of conjoined literals $(\, \ell_1 \wedge \ldots \wedge \ell m_1\,) \vee (\, \ell_1 \wedge \ldots \wedge \ell m_2\,) \vee \ \ldots \vee (\, \ell_1 \wedge \ldots \wedge \ell m_p\,)$.

conjunction will no longer be if the individual interpretations are incompatible. With DNNF's this is never the case because atoms are not shared across conjunctions and, consequently, any interference is excluded. Moreover, if the symbols *true* and *false* are not present, then the DNNF is intrinsically satisfiable, and the reason is precisely the same one: with no atoms being shared, the truth assignments amongst subformulas are always compatible and deriving a refutation is just not possible. Note that for achieving a refutation one must inevitably encounter either the explicit symbol $false$ or a contradictory expression of the form $A \wedge \neg A$. But, by assumption, no constant symbols are present and, by definition, this type of expression never occurs, neither implicitly nor explicitly, inside a DNNF. This shows how the compilation technique is at the same time a refutation procedure. Which seems reasonable enough, since the result of efficiently compiling a contradiction should be the constant symbol $false$. Therefore, a non-empty DNNF is always satisfiable.

Naturally, DNF's inherit the same property since, as mentioned previously, they are a sub-class of DNNF's. Still, DNNF formulas are potentially much smaller than equivalent DNF's because they are less restrictive and accommodate non distributed elements. Take, for instance, the DNNF formula $F = (A \vee B) \wedge (C \vee D)$, which also happens to be in CNF. Its DNF representation is given by $F = (A \wedge C) \vee (A \wedge D) \vee (B \wedge C) \vee (B \wedge D)$ where the size of F has duplicated. In the worst cases, the DNF can be exponentially bigger than an optimal DNNF.

In the process of answering queries, the DNNF theory must be combined with the query objects. For example, if one wants to test whether a theory $\mathcal{H}$ entails some sentence $\mathcal{S}$, the usual procedure is to derive a refutation of $\mathcal{H} \wedge \neg\mathcal{S}$. In other words, a satisfiability test on $\mathcal{H} \wedge \neg\mathcal{S}$ is be performed. We know that $\mathcal{H}$ alone is satisfiable because it is a compiled DNNF; and, by hypothesis, $\neg\,\mathcal{S}$ is also satisfiable, otherwise it would be pointless to test it against $\mathcal{H}$. If the extended theory $\mathcal{H} \wedge \neg\mathcal{S}$ were a DNNF, then the test would be completed and positive. However, even if $\neg\,\mathcal{S}$ is also a DNNF, the composition $\mathcal{H} \wedge \neg\mathcal{S}$ may have lost its DNNF status because DNNF's are not closed under the conjunctive operation. In the event that some of the atoms contained in $\neg\,\mathcal{S}$ happen to be present in $\mathcal{H}$ the decomposability property gets violated. Hence satisfiability is also no longer guaranteed.

As a solution for bypassing the closure problem, the following operations are defined which preserve the normal form.

Definition 2.2.12 (Conditioning) *Let $\mathcal{S}$ be a propositional sentence and $\ell = \ell_1 \wedge \ldots \wedge \ell_n$ a conjunctive set of literals. The conditioning of $\mathcal{S}$ on ℓ, denoted $\mathcal{S}\,|\,\ell$, is*

the sentence obtained from $\mathcal{S}$ *after replacing every occurrence of* ℓ_i *by true and every occurrence of* $\bar{\ell}_i$ *by* $false$*, for each* $i = 1, \ldots, n$*. Where literal* $\bar{\ell}$ *is the negative of* ℓ*, ie* $\bar{\ell} = \neg\,\ell$ *or* $\neg\,\bar{\ell} = \ell$ *.*

Definition 2.2.13 (Conjoining) *Let* $\mathcal{S}$ *be a propositional sentence and* $\ell = \ell_1 \wedge \ldots \wedge \ell_n$ *a conjunctive set of literals. The conjoining of* $\mathcal{S}$ *and* ℓ *is defined as the following operation*

$$\text{Conjoin}(\mathcal{S}, \ell) = (\mathcal{S}\,|\,\ell) \wedge \ell$$

As an example, suppose that we had a DNNF theory given by $\mathcal{H} = (\,a \wedge (\neg\, b \vee c)\,) \vee (b \wedge \neg\, a)$, and conclusion of the form $\mathcal{S} = b$. As mentioned above, in order to test the entailment $\mathcal{H} \models \mathcal{S}$ we will perform a satisfiability test on $\mathcal{H} \wedge \neg\,\mathcal{S}$. The expression $\mathcal{H} \wedge \neg\,\mathcal{S}$ is not DNNF because $\text{atoms}\,(\mathcal{H}) = \{a,\, b,\, c\,\}$ and $\text{atoms}\,(\neg\,\mathcal{S}) = \{b\}$ have the non empty intersection $\{b\}$. But if instead we perform the closed conjoining operation $(\mathcal{H}\,|\,\neg\,\mathcal{S}) \wedge \neg\,\mathcal{S}$ we get:

$$\begin{aligned} (\mathcal{H}\,|\,\neg\,\mathcal{S}) &= (\mathcal{H}\,|\,\neg\, b) \\ &= (\,a \wedge (\mathit{true} \vee c)\,) \vee (\mathit{false} \wedge \neg\, a) \\ &= a \end{aligned} \tag{2.2.16}$$

and

$$(\mathcal{H}\,|\,\neg\,\mathcal{S}) \wedge \neg\,\mathcal{S} \;=\; a \wedge \neg\, b \tag{2.2.17}$$

where occurrences of b and $\neg\, b$ on $\mathcal{H}$ have been replaced by false and true, respectively. Replacing the shared atom b by a constant symbol has renormalised $\mathcal{H} \wedge \neg\,\mathcal{S}$ towards an equivalent DNNF expression which, in this case, is satisfiable.

The effect of conjoining and conditioning is similar to the $\top - \bot$ reduction rules defined for the NAND system (see Theorems 3.3.4 and 3.3.5). Although in NAND the goal is to simplify the initial formula which is not necessarily DNNF, the aim here is to preserve the normal form. Furthermore, when the replacement operation is combined with the set of immediate simplifications $\{\,\mathit{true} \wedge A \simeq A,\quad \mathit{false} \wedge A \simeq \mathit{false},\quad \mathit{true} \vee A \simeq \mathit{true},\quad \mathit{false} \vee A \simeq A\,\}$, the whole operation is similar to the *one-literal rule* of the non-clausal Davis-Putnam (NCDP) procedure ([18]; see also [9] or the original papers [5, 6] for the clausal version).

But it may not always be that simple. When $\neg\,\mathcal{S}$ is not a conjunctive set of literals, as required by the conjoining operation preconditions, the conditioning phase cannot be

performed. Otherwise, the resulting expression would not be an equivalent one. In those cases, a possible solution we suggest is to convert $\neg\mathcal{S}$ into a DNF, where the conjoining can be independently performed on each disjunct. But perhaps a better solution is to split the conjoining operation into separate parts at every disjunction $\vee$ in $\neg\mathcal{S}$, avoiding the need to perform the prior DNF normalisation step.

Consider the case where $\neg\mathcal{S} = \neg b \wedge (\neg a \vee c)$. The conjoining can be firstly performed on $\neg b$, and afterwards on the two disjuncts a, c:

$$\begin{aligned} \mathcal{H}_1 &= \text{Conjoin}\,(\mathcal{H}, \neg b) \\ &= a \wedge \neg b \end{aligned} \tag{2.2.18}$$

$$\begin{aligned} \mathcal{H}_2 &= \text{Conjoin}\,(\mathcal{H}_1, \neg a) \\ &= (false \wedge \neg b) \wedge \neg a \\ &= false \end{aligned} \qquad \begin{aligned} \mathcal{H}_3 &= \text{Conjoin}\,(\mathcal{H}_1, c) \\ &= (a \wedge \neg b) \wedge c \\ &= a \wedge \neg b \wedge c \end{aligned} \tag{2.2.19}$$

and the final result would be

$$\begin{aligned} \text{Conjoin}\,(\mathcal{H}, \neg\mathcal{S}) &= \mathcal{H}_2 \vee \mathcal{H}_3 \\ &= \mathcal{H}_3 \end{aligned} \tag{2.2.20}$$

DNNF Compilation Algorithm

The conjoining operation pushes an expression towards DNNF. Indeed, the proposed algorithm for the actual construction of DNNF's is also based on conjoining and conditioning. One may recall at this point that the IF-normal form of OBDD's is also based on an equivalent reduction theorem. In fact, as mentioned by Darwiche, any algorithm that can be used for compiling OBDD's can also be used for DNNF's. But in principle DNNF's are expected to be more space efficient and would require a better algorithm for that achievement.

The input for the DNNF algorithm is a propositional theory in clausal form. The clauses should be previously partitioned into a *decomposition tree* — a tree constructed from the recursive binary partitioning of the initial set of clauses. In detail, if Δ is a clausal theory containing n clauses, than $\Delta_{\text{left}} \wedge \Delta_{\text{right}}$ is a binary partition of Δ into two sub-theories, one containing k clauses and the other containing $n - k$ clauses. When the partitioning progresses recursively, the decomposition tree builds up — the nodes are the partitioning points, with the first partition being the root node, and each leaf corresponds to a single clause. A possible decomposition tree for the theory $\Delta = \{(\neg F \vee G), (F \vee H), (G \vee H)\}$ is shown in Figure 2.3.

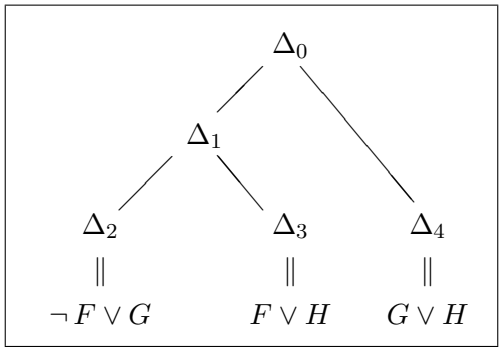


Figure 2.3: Example of a decomposition tree for the clause set $\{(\neg F \vee G), (F \vee H), (G \vee H)\}$

In a decomposition tree, a node Δ_k identifies the sub-theory composed of the clauses contained in the subtrees below it. The atoms of Δ_k, denoted $\text{atoms}(\Delta_k)$, is defined as the set of atoms appearing in the theory Δ_k. For this particular tree we have $\Delta_0 = \Delta$ and $\Delta_1 = \{(\neg F \vee G), (F \vee H)\}$, as well as $\text{atoms}(\Delta_0) = \text{atoms}(\Delta_1) = \{F, G, H\}$, $\text{atoms}(\Delta_2) = \{F, G\}$, $\text{atoms}(\Delta_3) = \{F, H\}$ and $\text{atoms}(\Delta_4) = \{G, H\}$.

The DNNF algorithm then goes as follows:

Darwiche's Algorithm for Compiling CNF into DNNF:

Given a propositional theory in clausal form, Δ, and a decomposition tree for Δ with $\Delta_{\ell,k}$ and $\Delta_{r,k}$ denoting the left and right sub-theories at each partition node k, then an equivalent DNNF theory can be constructed through recursive application of the following steps:

1. *if Δ contains a single clause ℓ, then* $\text{DNNF}(\Delta) = \ell$
2. *otherwise,* $\text{DNNF}(\Delta_k) = \bigvee_\beta \text{DNNF}(\Delta_{\ell,k} \,|\, \beta) \wedge \text{DNNF}(\Delta_{r,k} \,|\, \beta) \wedge \beta$

where β is an instantiation $\beta_1 \wedge \ldots \wedge \beta_n$ over the atoms p_i ($i = 1, \ldots, n$) shared by the left and right sub-theories $\Delta_{\ell,k}$ and $\Delta_{r,k}$. An instantiation β_i over an atom p_i is one of $\beta_i = p_i$ or $\beta_i = \neg p_i$. The disjunctive operator $\bigvee_\beta$ ranges over all possible instantiations.

The central operation performed in this algorithm is basically the same as a Shanon expansion (repeated over several instantiations). In terms of the conjoining and conditioning

operators, the Shanon expansion of Δ with respect to atom p is defined as:

$$SE(\Delta, p) \quad = \quad (\,(\,\Delta\,|\,p\,)\,\wedge\,p\,)\quad\vee\quad(\,(\,\Delta\,|\,\neg p\,)\,\wedge\,\neg p\,) \tag{2.2.21}$$

And again, this is a similar operation to the IF-distribution-reduction operation for compiling OBDD's. But there are at least two crucial differences with respect to an OBDD-type algorithm: i) the partitioning of the clause set conditions the choice of atoms, even if an ordering is predefined; ii) distributing over instantiations is localised around each sub-theory according to the atom sets.

Our next example (based on the *Prawitz rule*[1]) will give a clearer view of what is meant by these remarks.

Example 2.2.14 *Consider the clausal theory* $\Delta = \{(\neg F \vee G), (F \vee H), (G \vee H)\}$ *with decomposition tree given as in Figure 2.3. Direct application of the DNNF algorithm on* Δ *gives the following trace:*

$$\mathrm{DNNF}(\Delta_0) = \bigvee_{\beta}\ \mathrm{DNNF}(\Delta_1|\beta)\ \wedge\ \mathrm{DNNF}(\Delta_4|\beta)\ \wedge\ \beta$$

where Δ_1 *and* Δ_4 *are the left and right subtrees of* Δ_0*, respectively, and* β *ranges over the instantiations over the atom set* $\{G, H\}$*, obtained by calculating* $\mathrm{atoms}(\Delta_1) \bigcap \mathrm{atoms}(\Delta_4) = \{F, G, H\} \bigcap \{G, H\} = \{G, H\}$*. So, in principle,* β *should be iterating over* $\{(G \wedge H), (G \wedge \neg H), (\neg G \wedge H), (\neg G \wedge \neg H)\}$*, producing the following result:*

$$\begin{aligned}
(\Delta_1 |\ \ G\ \wedge\ H\ \ ) &= (\,(\,\neg F \vee G\,)\ \wedge\ (\,F \vee H\,)\,|\,(\,G \wedge H\,)\,)\\
&= (\,\neg F \vee true\,)\ \wedge\ (\,F \vee true\,)\\
&= true
\end{aligned}$$

$$\begin{aligned}
(\Delta_1 \,|\, \neg G\ \wedge\ H\ ) &= (\,\neg F \vee false\,)\ \wedge\ (\,F \vee true\,)\\
&= \neg F
\end{aligned}$$

$$\begin{aligned}
(\Delta_1 |\ \ G\ \wedge \neg H\ ) &= (\,\neg F \vee true)\ \wedge\ (\,F \vee false\,)\\
&= F
\end{aligned}$$

$$\begin{aligned}
(\Delta_1 \,|\, \neg G \wedge \neg H) &= (\,\neg F \vee false\,)\ \wedge\ (\,F \vee false\,)\\
&= false
\end{aligned}$$

[1] See equation (2.2.44).

and, similarly,

$$(\Delta_4 \mid G \wedge H) = true \qquad (\Delta_4 \mid \neg G \wedge H) = true$$
$$(\Delta_4 \mid G \wedge \neg H) = true \qquad (\Delta_4 \mid \neg G \wedge \neg H) = false$$

obtaining,

$$\mathrm{DNNF}(\Delta_0) = (G \wedge H) \vee (\neg F \wedge \neg G \wedge H) \vee (F \wedge G \wedge \neg H)$$

which is only a sub-optimal DNNF!

It is actually DNF. Note how both H and G appear twice. Even something like

$$(H \wedge (G \vee \neg F)) \vee (F \wedge G \wedge \neg H)$$

would have already been better.

The previous example has shown how (not) to construct a DNNF. One of the reasons for the poor result resides in how the algorithm has been applied. In practice, the conditioning operation should not be performed over all possible instantiations simultaneously, but only over one shared atom at a time. A single conditioning operation over one variable may induce other variables, besides the selected one, to disappear from the sub-theories. When that happens, distributing over those variables is pointless. What should have been done in the above example was to first distribute around G (or H) and then, if still and where required, around H (or G). Choosing between G and H to be the starting variable is another issue that also has to be analysed. Either some predefined ordering $G < H$ (or $H < G$) is obeyed or a *case analysis* is performed. The case analysis may be to test which of G or H removes the most variables, which may be expensive computation wise, or to simply select the one that appears in the largest number of clauses. In our case it is indifferent whether to start with G or H, so we will assume a lexicographic ordering and start with G. As a reminder, the implied nodes are $\Delta_1 = \{(\neg F \vee G), (F \vee H)\}$ and $\Delta_4 = \{G \vee H\}$:

$$(\Delta_1 \mid G) = F \vee H \qquad (\Delta_1 \mid \neg G) = \neg F \wedge (F \vee H) \qquad (2.2.22)$$
$$= \neg F \wedge (false \vee H)$$
$$= \neg F \wedge H$$

$$(\Delta_4 \mid G) = true \qquad (\Delta_4 \mid \neg G) = H$$

Notice now how the new sub-theories $(\Delta_1 \mid G)$ and $(\Delta_4 \mid G)$ do not share any atoms, so that conditioning on H or $\neg H$ can be skipped. Note as well how further simplifications

can take place as soon as fresh new atoms become available for conjoining, like it has been done for $(\Delta_1 \,|\, \neg G)$.

Regarding the $\neg G$ cases, both $(\Delta_1 \,|\, \neg G)$ and $(\Delta_4 \,|\, \neg G)$ still contain the shared atom H, so conditioning on H and $\neg H$ should be performed for them:

$$(\Delta_1 \,|\, \neg G \wedge H) = \neg F \wedge true \qquad (\Delta_1 \,|\, \neg G \wedge \neg H) = false \tag{2.2.23}$$

$$(\Delta_4 \,|\, \neg G \wedge H) = true \qquad (\Delta_4 \,|\, \neg G \wedge \neg H) = false$$

Alternatively, further simplification on $(\Delta_1 \,|\, \neg G) \wedge (\Delta_4 \,|\, \neg G)$ would directly give $(\neg F \wedge H) \wedge H = \neg F \wedge H$. Either way, the final result is:

$$\text{DNNF}(\Delta) \;=\; (G \wedge (F \vee H)) \quad \vee \quad (\neg G \wedge H \wedge \neg F) \tag{2.2.24}$$

Even if this is a better solution than the one obtained in Example 2.2.14, it is still not optimal.

This brings us to the problem of choosing an optimal decomposition tree. If, in the same example, the leaf nodes Δ_3 and Δ_4 were switched, as shown in Figure 2.4, the outcome would be an equivalent, but different, DNNF.

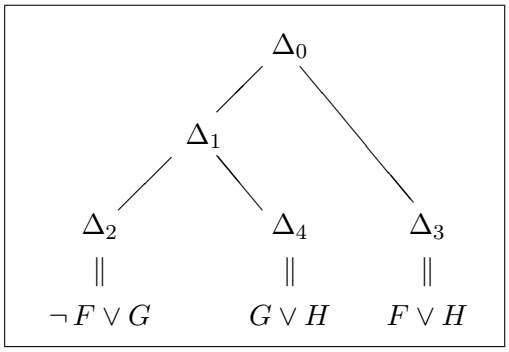


Figure 2.4: Decomposition tree for the clause set $\{(\neg F \vee G), (F \vee H), (G \vee H)\}$

Based on this new decomposition tree, the atom set shared by the left and right subtrees of Δ_0, which are Δ_1 and Δ_3, respectively, is no longer $\{ G, H \}$, but is instead given by:

$$\text{atoms}\,(\Delta_1) \bigcap \text{atoms}\,(\Delta_3) \quad = \quad \{F, H\} \tag{2.2.25}$$

A case analysis on F and H indicates that conjoining with F, rather than H, removes the largest number of variables. Under the new sub-theories $\Delta_1 = (\neg F \vee G) \wedge (G \vee H)$

and $\Delta_3 = F \vee H$, we obtain:

$$\begin{aligned}(\Delta_1 \mid F) &= G \wedge (G \vee H) \\ &= G \wedge (\mathit{true} \vee H) \\ &= G\end{aligned} \qquad (\Delta_1 \mid \neg F) = G \vee H \tag{2.2.26}$$

$$(\Delta_3 \mid F) = \mathit{true} \qquad (\Delta_3 \mid \neg F) = H$$

After the simplification $(\Delta_1 \mid \neg F) \wedge (\Delta_3 \mid \neg F) = (G \vee H) \wedge H = H$, we finally obtain:

$$\mathrm{DNNF}(\Delta) \quad = \quad (F \wedge G) \vee (\neg F \wedge H) \tag{2.2.27}$$

Clearly, this is the optimal DNNF for the set of clauses that were given.

What this tells us is that the final structure of a DNNF is very sensitive to the partitioning of the decomposition tree, since it directly affects the order in which atoms can be selected. Good decomposition trees are generally those with small *treewidth* — a measure on the degree of connectivity of the tree, given by the number of shared atoms between sibling subtrees of a node or between the node and its ancestors. In this way formula expansion is delayed or pushed to relevant groups of subformulas. But this is not always a decisive factor because the trees in Figures 2.3 and 2.4 have the same width and result in different DNNF optimisations. We shall not go in detail into the subject of building decomposition trees, but advise the interested reader to consult [8] for a more insightful discussion on this matter.

Alternative Compilation Methods

The disadvantage of using decomposition trees for compiling DNNFs is their reliance on clausal form. Alternatively, as suggested in [17], the method can be generalised to any normal form if Shanon expansion (also known as *semantic factoring*) is iteratively applied throughout the formula. Nevertheless, the problem of how and when atoms and subformulas should be selected for expansion remains. The general principle that aims to prevent exponential growth of the theory is, again, to localise the expansions around shared atoms only, with priority given to those with the highest reduction factor.

Other procedures that can be used for constructing DNNFs are regular NNF tableaux, KE-tableaux [1, 2], FNNF tableaux [11] or the NAND system. FNNF tableaux will be discussed later in Section 2.2.4, and NAND compilation is described in Chapter 3, Section 3.5.1.

As for regular tableaux, we know that any complete strictly analytic regular tableau reproduces a minimal DNF model of its root. Because closed branches are ignored, the model does not contain contradictory terms of type $\neg A \wedge A$. If, in addition, regularity is imposed, then no atom will appear twice in any one branch. By combination of these two properties, decomposability is ensured. Mondadori's KE-tableaux are even more versatile because they are able to reproduce Shanon's expansion by virtue of the *principle of bivalence* (PB) or the atomic cut rule. In either case, the compilation efficiency depends on the strategies implemented throughout the tableaux, like formula selection, ordering of literals, etc. The complexities inherent to choosing the best strategy are again related the problem of building an optimal decomposition tree and deciding on variable ordering.

2.2.3 Path Dissolution

In the previous section the class of DNNF formulas was introduced. Here we describe a superset of DNNFs studied by Murray [16, 20, 17], called the *full dissolvents*. A *full dissolvent* is an NNF formula whose DNF does not contain any complementary literals across conjunctions. Decomposability has been relaxed to a *linklessness* property, where a *link* is any conjunction of two complementary literals.

As an alternative to DNNF's, this new class of formulas also encloses interesting properties with regards to knowledge compilation. Because linklessness is less restrictive than decomposability, full dissolvents are also universal — ie, they are able to represent any propositional theory — and are more space efficient than DNNFs. Moreover, although atoms can be shared across conjunctions, entailment, projection and conditioning operations are still performed in linear time. In general, however, full dissolvents are not as tractable as DNNF's, like when computing minimum cardinalities.

Path dissolution is an inference rule that transforms any NNF formula into a full dissolvent by eliminating from it all links. In [16] it has been defined in terms of *semantic graphs*, which are a nice diagrammatic way to globally visualise the connections within complex NNF formulas. Below, a semantic graph is depicted next to an equivalent NNF formula:

$$[\,(A \vee \overline{C}) \wedge (B \vee (C \wedge \overline{A})\,] \;\vee\; [\,\overline{D} \wedge E\,] \tag{2.2.28}$$

$$\begin{array}{ccccccc} A & \vee & \overline{C} & & & & E \\ & \wedge & & & \vee & & \wedge \\ & & C & & & & \overline{D} \\ B & \vee & \wedge & & & & \\ & & \overline{A} & & & & \end{array} \tag{2.2.29}$$

Each literal in the graph is called a *node*, and is also a *subgraph*. A *subgraph* is a semantic graph that is contained in another semantic graph. Furthermore, a c-arc (d-arc) is defined as a $\wedge$-connection ($\vee$-connection) between any two semantic graphs. The total number of c-arcs (d-arcs) matches the number of symbols $\wedge$ ($\vee$). In the above graph there are three c-arcs and three d-arcs: for example, a d-arc between A and $\overline{C}$ and a c-arc between E and $\overline{D}$.

The following definitions are important for subsequent discussion, so we should also be acquainted with them.

Definition 2.2.15 (*c-d* Connection) *Two nodes in a semantic graph are c-connected (d-connected) if there is a c-arc (d-arc) joining its subgraphs.*

Definition 2.2.16 (*c-d* Path) *In a semantic graph, a c-path (d-path) is a maximal set of c-connected (d-connected) nodes.*

For example, in our graph, A and C are c-connected, while $\overline{A}$ and C are d-connected. The c-paths are $\{A, B\}$, $\{A, C, \overline{A}\}$, $\{\overline{C}, B\}$, $\{\overline{C}, C, \overline{A}\}$ and $\{E, \overline{D}\}$, while the d-paths are $\{A, \overline{C}, E\}$, $\{A, \overline{C}, \overline{D}\}$, $\{B, C, E\}$, $\{B, C, \overline{D}\}$, $\{B, \overline{A}, E\}$ and $\{B, \overline{A}, \overline{D}\}$. Therefore, when the graph is read from left to right, systematically, conjoining all possible d-paths, one obtains a CNF representation of the original formula. Conversely, reading it in a top-bottom way and disjoining all c-paths returns an equivalent DNF. We also see that there are two links on variables C and A within the first two c-paths listed above.

In order to eliminate links, the path dissolution technique operates by replacing the subgraph containing the link by an equivalent one where the link has been *dissolved*, i.e, the c-path containing the link is removed and paths are restructured. The following concepts will be used in the definition of the dissolvent operator.

Definition 2.2.17 (CPE, CC) *Let N be a node in a semantic graph $\mathcal{G}$.*

- *The c-path extension of N in $\mathcal{G}$, denoted* $\mathrm{CPE}(N, \mathcal{G})$*, is the subgraph of $\mathcal{G}$ consisting of all the c-paths through $\mathcal{G}$ that contain N.*

- *The c-path complement N in $\mathcal{G}$, denoted* $\mathrm{CC}(N, \mathcal{G})$*, is the subgraph of $\mathcal{G}$ consisting of all the c-paths through $\mathcal{G}$ that do not contain N.*

Upon this definition it is clear that for any two subgraphs $\mathcal{G}$ and $\mathcal{H}$ and a node N, we always have the identities

$$CPE(N, \mathcal{G}) \ \vee \ CC(N, \mathcal{G}) \ = \ \mathcal{G} \tag{2.2.30}$$

$$\mathrm{CPE}(N, \mathcal{G}) \ \wedge \ \mathrm{CPE}(\overline{N}, \mathcal{H}) \ = \ \emptyset \tag{2.2.31}$$

where the empty set represents the empty graph which in turn represents the constant symbol $false$. The equations hold whether or not $N \in \mathcal{G}$ or $N \in \mathcal{H}$, because for $N \notin \mathcal{G}$ it follows that $\mathrm{CPE}(N, \mathcal{G}) = \emptyset$ and $\mathrm{CC}(N, \mathcal{G}) = \mathcal{G}$.

We procede with an example.

Example 2.2.18 *Consider $\mathcal{G}$ to be the complete semantic graph displayed in (2.2.29). The c-path extension of $\overline{C}$ in $\mathcal{G}$ is the subgraph*

$$\mathrm{CPE}(\overline{C}, \mathcal{G}) \quad = \quad \begin{array}{ccccc} & & \overline{C} & & \\ & & \wedge & & \\ & & & & C \\ & B & \vee & & \wedge \\ & & & & \overline{A} \end{array} \tag{2.2.32}$$

While the c-path complement of $\overline{C}$ in $\mathcal{G}$ is precisely the complementary subgraph

$$\mathrm{CC}(\overline{C}, \mathcal{G}) \quad = \quad \begin{array}{ccccccc} & & A & & & & E \\ & & \wedge & & & & \wedge \\ & & & C & & & \overline{D} \\ & B & \vee & \wedge & & & \\ & & & \overline{A} & & & \end{array} \tag{2.2.33}$$

We may now give a definition for the *dissolution* operator DV[1].

Definition 2.2.19 (DV) *Let $\mathcal{G}$ and $\mathcal{H}$ be two semantic graphs connected by a c-arc, such that $O = \mathcal{G} \wedge \mathcal{H}$, and let N and $\overline{N}$ be two complementary nodes with $N \in \mathcal{G}$ and $\overline{N} \in \mathcal{H}$. The dissolvent of the link $\mathcal{N} = \{N, \overline{N}\}$ in O, denoted* $\mathrm{DV}(\mathcal{N}, \mathcal{O})$*, is given by*

[1]The original definitions given in [16] for CPE, CC and DV are more general than those presented here, because they do not restrict to nodes, but to subgraphs. However, they do require the introduction of further concepts on semantic graph theory and a level of detail that would overload this overview on path dissolution.

$$\mathrm{DV}(\mathcal{N},\,\mathcal{O}) \;=\; \begin{array}{c}\mathrm{CPE}\,(N,\mathcal{G})\\ \wedge\\ \mathrm{CC}\,(\overline{N},\mathcal{H})\end{array} \;\vee\; \begin{array}{c}\mathrm{CC}\,(N,\mathcal{G})\\ \wedge\\ \mathrm{CC}\,(\overline{N},\mathcal{H})\end{array} \;\vee\; \begin{array}{c}\mathrm{CC}\,(N,\mathcal{G})\\ \wedge\\ \mathrm{CPE}\,(\overline{N},\mathcal{H})\end{array} \tag{2.2.34}$$

Dissolution is a re-write rule, and the expression above is obtained from the identities (2.2.30) and (2.2.31) applied to $\mathcal{G}$ and $\mathcal{H}$ and through distributivity. Hence, the following are equivalent:

$$\mathrm{DV}\,(\mathcal{N},\,\mathcal{G}\wedge\mathcal{H}) \;\simeq\; \begin{array}{c}\mathcal{G}\\ \wedge\\ \mathcal{H}\end{array} \;\simeq\; \begin{array}{c}CPE(N,\mathcal{G}) \;\vee\; CC(N,\mathcal{G})\\ \wedge\\ CPE(\overline{N},\mathcal{H}) \;\vee\; CC(\overline{N},\mathcal{H})\end{array} \tag{2.2.35}$$

After distributing (2.2.35) over $\wedge$, the c-path $\mathrm{CPE}\,(N,\mathcal{G}) \wedge \mathrm{CPE}\,(\overline{N},\mathcal{H})$ disappears by virtue of (2.2.31), yielding $\mathrm{DV}\,(\mathcal{N},\,\mathcal{G}\wedge\mathcal{H})$ as in (2.2.34).

In practice, the dissolution operator is usually replaced by one of the more compact, equivalent transformations:

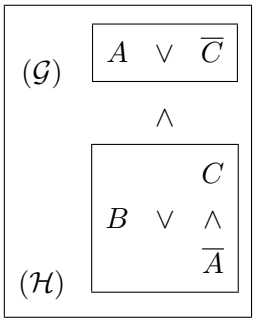
$$\mathrm{DV}(\mathcal{N},\,\mathcal{G}\wedge\mathcal{H}) \;=\; \begin{array}{c}\mathcal{G}\\ \wedge\\ \mathrm{CC}\,(\overline{N},\mathcal{H})\end{array} \;\vee\; \begin{array}{c}\mathrm{CC}\,(N,\mathcal{G})\\ \wedge\\ \mathrm{CPE}\,(\overline{N},\mathcal{H})\end{array} \tag{2.2.36}$$

or

$$\mathrm{DV}(\mathcal{N},\,\mathcal{G}\wedge\mathcal{H}) \;=\; \begin{array}{c}\mathrm{CPE}\,(N,\mathcal{G})\\ \wedge\\ \mathrm{CC}\,(\overline{N},\mathcal{H})\end{array} \;\vee\; \begin{array}{c}\mathrm{CC}\,(N,\mathcal{G})\\ \wedge\\ \mathcal{H}\end{array} \tag{2.2.37}$$

Lets see an application of dissolution with an example.

The semantic graph displayed in (2.2.29) had two links $\mathcal{C} = \{C, \overline{C}\}$ and $\mathcal{A} = \{A, \overline{A}\}$, both contained in c-paths traversing the left hand side subgraph $\mathcal{L} = \mathcal{G} \wedge \mathcal{H}$:



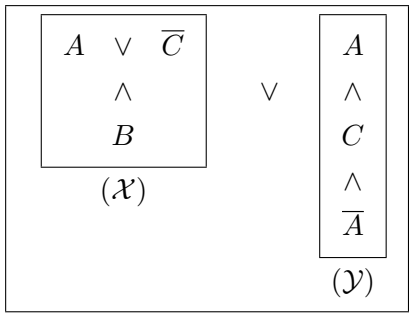
(2.2.38)

To eliminate those links, the dissolvent operator should be applied both on $\mathcal{C}$ and on $\mathcal{A}$. We start with $\mathcal{C}$, and note that $C \in \mathcal{H}$ and $\overline{C} \in \mathcal{G}$. Therefore, the c-path extensions and complements we must compute for the link $\{C, \overline{C}\}$ are the following:

$$\begin{aligned} \mathrm{CPE}\,(\overline{C},\, \mathcal{G}) &= \overline{C} & \qquad \mathrm{CC}\,(\overline{C},\, \mathcal{G}) &= A \\ \mathrm{CC}\,(C,\, \mathcal{H}) &= B & \qquad \mathrm{CPE}\,(C,\, \mathcal{H}) &= C \wedge \overline{A} \end{aligned} \tag{2.2.39}$$

In fact we do not need $\mathrm{CPE}\,(\overline{C},\, \mathcal{G})$ if equation (2.2.36) is used, leading to the subgraph $\mathcal{L}_{\mathcal{XY}}$

$$\mathrm{DV}\,(\mathcal{C},\, \mathcal{L}) \quad = \quad$$



(2.2.40)

Alternatively, if equation (2.2.37) is used instead, then we do not need $\mathrm{CPE}\,(C,\, \mathcal{H})$ to obtain the equivalent subgraph $\mathcal{L}_{\mathcal{WZ}}$

$$\mathrm{DV}\,(\mathcal{C},\,\mathcal{L}) \quad = \quad \boxed{\begin{array}{ccc} \boxed{\begin{array}{c}\overline{C}\\ \wedge \\ B\end{array}}_{(\mathcal{W})} & \vee & \boxed{\begin{array}{ccc} & A & \\ & \wedge & \\ & & C \\ B & \vee & \wedge \\ & & \overline{A}\end{array}}_{(\mathcal{Z})} \end{array}}_{(\mathcal{L}_{WZ})} \tag{2.2.41}$$

where C is playing the role of the $\overline{N}$ that appears on equations (2.2.36) and (2.2.37).

The link $\mathcal{C} = \{C, \overline{C}\}$ has been dissolved from both $\mathcal{L}_{\mathcal{XY}}$ and $\mathcal{L}_{\mathcal{WZ}}$, and the link $\mathcal{A} = \{A, \overline{A}\}$ is now confined to the right hand side subgraphs $(\mathcal{Y})$ and $(\mathcal{Z})$. Therefore, it suffices to apply dissolution on $\mathcal{A}$ using either $(\mathcal{Y})$ or $(\mathcal{Z})$. Obviously, $(\mathcal{Y})$ is the easiest case since the whole subgraph consists of a single c-path along the link. Hence we can directly replace $(\mathcal{Y})$ by $\emptyset$ in $\mathcal{L}_{XY}$ to generate the (linkless) full dissolvent for $\mathcal{L}$

$$\mathrm{FD}\,(\mathcal{L}) \quad = \quad \begin{array}{ccc} A & \vee & \overline{C} \\ & \wedge & \\ & B & \end{array} \tag{2.2.42}$$

If we were to use $(\mathcal{Z})$ instead, then we should repartition $(\mathcal{Y})$ into two c-connected subgraphs, one containing A and the other containing $\overline{A}$, and repeat for $(\mathcal{Y})$ the steps we performed for $\mathcal{L}$.

One important special case of the dissolution operator is the following. Consider again the definition of DV and its transformation (2.2.34). When $\overline{\mathrm{CPE}\,(N, \mathcal{G})} = \mathrm{CPE}\,(\overline{N}, \mathcal{H})$, then the subgraph $\mathrm{CC}\,(N, \mathcal{G}) \wedge \mathrm{CC}\,(\overline{N}, \mathcal{H})$ is subsumed and the dissolvent becomes

$$\begin{array}{ccc} \mathrm{CPE}\,(N, \mathcal{G}) & & \mathrm{CC}\,(N, \mathcal{G}) \\ \wedge & \vee & \wedge \\ \mathrm{CC}\,(\overline{N}, \mathcal{H}) & & \mathrm{CPE}\,(\overline{N}, \mathcal{H}) \end{array} \tag{2.2.43}$$

This derives from what is commonly know as the *Prawitz rule*, here expressed in terms of semantic graphs:

$$\begin{array}{ccccc} N & \vee & X & & N \quad \overline{N} \\ & \wedge & & \simeq & \wedge \quad \vee \quad \wedge \\ \overline{N} & \vee & Y & & Y \quad X \end{array} \tag{2.2.44}$$

which is itself a special case of Shanon expansion[1]. The term that has been subsumed on the right hand side corresponds to $X \wedge Y$. This transformation is particularly useful because the size of the formula does not increase and not only the link $\{N, \overline{N}\}$ is dissolved but also any possible links between X and Y. Furthermore, if Y (or X) contain a node $\overline{N}$ (or N), it is now possible to apply *unit dissolution* on N to the subgraphs $N \wedge Y$ (or $\overline{N} \wedge X$). Unit dissolution is another special case of dissolution that occurs when one of the subgraphs $\mathcal{G}$ or $\mathcal{H}$ is a node. Here, if we name $\mathcal{G} \equiv N$ and $\mathcal{H} \equiv Y$, then we have that $\mathrm{CPE}(N, \mathcal{G}) = N$ and $\mathrm{CC}(N, \mathcal{G}) = \emptyset$. The only subgraph of (2.2.34) that survives (if $\mathcal{H}$ is not itself a node too, in which case the whole subgraph $\mathcal{G} \wedge \mathcal{H}$ is empty) is $\mathrm{CPE}(N, \mathcal{G}) \wedge \mathrm{CC}(\overline{N}, \mathcal{H})$. This has the same effect as replacing all occurrences of $\overline{N}$ in $\mathcal{H}$ by $false$, which is a linear operation in the size of $\mathcal{H}$. In [17], a special class of formulas is presented that can be compiled in linear time and space into DNNF essentially using the Prawitz rule and dissolution.

This overview of path dissolution has concentrated on techniques to remove unsatisfiable paths from a formula, where the operators acted at the literal level. A wider perspective of this theory is presented in [16, 20]. It includes the dual operators of DV, CPE and CC that act on d-paths as well as on disjunctive links. A *link* is disjunctive if it refers to complementary literals that are d-connected. The dual of DV has the effect of removing tautologies form a formula. In [20], in particular, an *anti-link* operator is used to detect subsumed paths. Where an anti-link, as opposed to a link, is an identical pair of connected literals.

2.2.4 Factored Negation Normal Form

One of the major problems inherent to CNF, DNF or even NNF normal forms is the exponential increase in the size of a formula when the original logical expression contains equivalences. Each equivalence sign that is normalised upon into $\wedge$'s and $\vee$'s doubles in size:

$$A \Leftrightarrow B \quad \simeq \quad (A \wedge B) \vee (\neg A \wedge \neg B) \tag{2.2.45}$$

This affects any method, either a proof or knowledge compilation system that takes as input an NNF. This basically includes all the methods introduced so far — DPLL, tableaux, OBBD, DNNF and full dissolvents — of which the first three are renowned mainstream techniques in theorem proving.

An alternative method that operates on *negated form* (NF) as a generalisation of NNF has been suggested by Hähnle, Murray and Rosenthal [11]. Negated form formulas are

[1] Already defined in (2.2.21) and later in (3.3.19)

those constructed with the binary connectives $\wedge$, $\vee$ and $\Leftrightarrow$ and where all negations $\neg$ are at the atomic level. The only (crucial) difference between NF and NNF is the presence of the equivalence symbol in the former, which has been introduced to solve the duplication issue. Therefore, what is needed, is a new (or adapted) method that is able to interpret NF.

The NF procedure they introduce is an efficient proof system resembling an NCDP at the tableaux level which is an improvement over KE-tableaux (even if non NF input is used). It is also a knowledge compilation system producing formulas in *factored negation normal form* (FNNF) or *ordered factored negation normal form* (OFNNF) which, in a similar way to OBBD's, provide a canonical representation for any logical formula. FNNF itself is not NF, and that is due to main inference rule of the compilation algorithm which, again, is Shanon expansion (SE)[1] or, in tableaux vocabulary, the atomic cut. But the FNNF is the end stage of the compilation, and avoiding exponential initial or middle stages can dramatically speed up the process, particularly if formula reduction is taking place.

The NF algorithm is similar to DPLL and BDD's in the sense that it applies Shanon expansion for each atom of an NF formula, as well as formula reductions (although it does not apply the pure literal rule). It is different form DNNF's because the choice of atoms is not oriented towards removing links or anti-links[2]. Moreover, it also becomes similar to OBDD's when atoms are selected in a preset order. In this case, what results is an *ordered factored negation normal form* (OFNNF).

The set of simplifications that are applied in combination with Shanon expansion are the idempotency laws

$$(A \vee A) \simeq A \qquad\qquad (A \wedge A) \simeq A \tag{2.2.46}$$

and all the usual simplifications pertaining to the constant symbols $true$ and $false$, of which we state the ones involving equivalences:

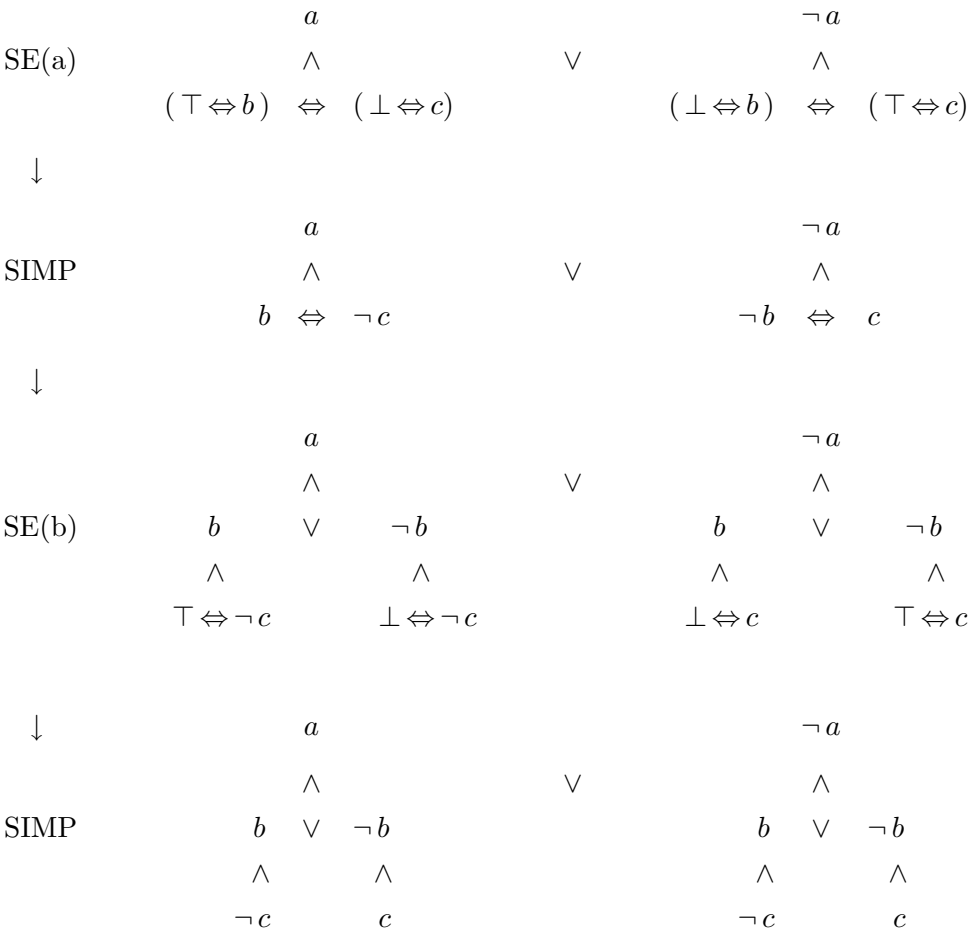
$$\begin{array}{rcl@{\qquad\qquad}rcl} A \Leftrightarrow false & \simeq & \neg A & A \Leftrightarrow \neg A & \simeq & false \\ A \Leftrightarrow true & \simeq & A & A \Leftrightarrow A & \simeq & true \end{array} \tag{2.2.47}$$

The next example shows how compilation is done without ever having to distribute equivalences.

[1] See equations (2.2.21), (3.3.18) or (3.3.19).

[2] Borrowing some of the path dissolution vocabulary, a *link* is a conjoined complementary pair of literals, and an *anti-link* is a conjoined identical pair of literals. See Section 2.2.3.

Example 2.2.20 *Consider the NF formula* $\mathcal{F} = (\, a \Leftrightarrow b\,) \Leftrightarrow (\, \neg\, a \Leftrightarrow c)$ *and the ordered atom set* $\{a, b, c\}$ *. The conversion steps into OFNNF(*$\mathcal{F}$*) are shown below in a semantic graph type representation, where* $\top \equiv true$ *and* $\bot \equiv false$ *. Simplifications (SIMP) alternate with Shanon expansions SE(p) on atom* $p \in \{a, b, c\}$*:*



FNNF compilation can also be defined in terms of tableaux, as a generalisation of Mondadori's KE-tableaux. In order to handle the equivalence connective from NF formulas, the KE β-rules are replaced by the more general Massacci's linear simplifications (the two right most). Atomic cut and α-rules (two left most) are left the same:

$$\frac{}{p \mid \neg p} \qquad \frac{\alpha_1 \wedge \alpha_2}{\begin{array}{c}\alpha_1\\ \alpha_2\end{array}} \qquad \frac{\begin{array}{c}\phi\\ (\psi[\,\phi\,])\end{array}}{\psi[\,true/\phi\,]} \qquad \frac{\begin{array}{c}\neg\, p\\ (\psi[\,p\,])\end{array}}{\psi[\,false/p\,]} \tag{2.2.48}$$

where p is an atom, ψ and ϕ are NF expressions and $\psi[true/\phi]$ represents ψ after the replacement of ϕ by $true$. Similarly for $\psi[false/p]$. Note that, in the right most Massacci rule, $\neg p$ must be a literal (or p an atom) because in NF negations only appear at the literal level. The atomic cut plus the Massacci's simplifications emulate Shanon expansion. Since they can never increase the size of the formula nor add branches to the tableaux, they should be given priority over the atomic cut or alpha rules. We will see on next Chapter that Massacci's simplifications are essentially to the $\top - \bot$ reduction rules for NAND (See Theorems 3.3.4 and 3.3.5).

Chapter 3

The NAND Formalism

The NAND system introduces a formal language that relies on classical logic representation constrained by the usage of the "NAND" boolean operator (or, in fact, an extension of it), meaning "not and" or, in natural English, "not both". The "NAND" operator, also known as Sheffer's *stroke* [24], is usually represented by the symbol $\uparrow$:

$$s \uparrow r \quad \equiv \quad \neg (s \wedge r) \tag{3.0.1}$$

The restriction to the use of $\uparrow$ is not an expressiveness limitation in the sense that $\{\uparrow\}$ forms a complete set of connectives. As we know, the usual boolean operators — conjunction ($\wedge$), disjunction ($\vee$), implication ($\Rightarrow$), equivalence ($\Leftrightarrow$) and negation ($\neg$) — can all be defined by means of Sheffer's stroke:

$$\begin{aligned}
(s \wedge r) &\quad\Leftrightarrow\quad (s \uparrow r) \uparrow (s \uparrow r) \\
(s \vee r) &\quad\Leftrightarrow\quad (s \uparrow s) \uparrow (r \uparrow r) \\
(s \Rightarrow r) &\quad\Leftrightarrow\quad s \uparrow (r \uparrow r) \\
(s \Leftrightarrow r) &\quad\Leftrightarrow\quad (s \uparrow r) \uparrow ((s \uparrow s) \uparrow (r \uparrow r)) \\
\neg s &\quad\Leftrightarrow\quad s \uparrow s
\end{aligned} \tag{3.0.2}$$

While deriving the above expressions we realise that we need to use the trick of replacing every occurrence of $\neg N$ by its equivalent $\neg(N \wedge N)$. This does not seem very practical or efficient. Moreover, the operator $\uparrow$ is not associative and is limited to its binary scope. By this we mean that in addition to $\neg(N_1 \wedge N_2)$ we would like to have an easy way of writing more general expressions like $\neg (N_1 \wedge N_2 \wedge \ldots \wedge N_n)$ or even just $\neg N$. This can be achieved by decoupling $\{\uparrow\}$ into $\{\neg, \wedge\}$ as to provide a more flexible usage of negation and conjunction. The resulting operator is then comparable to a Sheffer's stroke with variable

arity. As for the above expressions, they may be re-written in the following normal form, which from here on shall be referred to as the *NAND normal form*.

$$\begin{aligned}
(s \wedge r) &\quad\Leftrightarrow\quad \neg\,\neg\,(s \wedge r) \\
(s \vee r) &\quad\Leftrightarrow\quad \neg\,(\neg\, s \wedge \neg\, r) \\
(s \Rightarrow r) &\quad\Leftrightarrow\quad \neg\,(s \wedge \neg\, r) \\
(s \Leftrightarrow r) &\quad\Leftrightarrow\quad \neg\,(\neg(s \wedge r) \wedge \neg\,(\neg\, s \wedge \neg\, r)) \\
\neg\, s &\quad\Leftrightarrow\quad \neg\, s \\
s &\quad\Leftrightarrow\quad \neg\,\neg\, s
\end{aligned} \tag{3.0.3}$$

Note that in NAND normal form each sentence must be preceded by (at least) one occurrence of $\neg$. We call these the *outer level* negations. Conversely, *inner level* double negations are those that occur inside an expression (ie, they are not the main connective), and may be dropped whenever judged appropriate.

As an aside, it is worth mentioning that instead of "NAND" we could have explored the "NOR" connective, symbolically represented by $\downarrow$, meaning "not or" or "neither". As a matter of fact, the framework developed here works both for NAND and NOR under very few minor adaptations that do not affect the core procedures. We shall come back to this in the context of tree expansions.

3.1 Syntax and Semantics

We have followed the symbolic primitives suggested by Lin Yang [26] and have chosen to adopt the square bracket normal form "[ ]" which works both as a connective and a grouping indicator. This notation is also very convenient to be used in Prolog when interpreted as a list. The square bracket pairs act as group negations $\neg\,(\ \ )$ and commas are introduced in the place of the $\wedge$-connections:

$$\neg\,(N_1 \wedge N_2 \wedge \ldots \wedge N_n) \qquad \rightleftharpoons \qquad [\,N_1,\, N_2,\, \ldots,\, N_n\,] \tag{3.1.4}$$

Conversion to bracket NAND normal form (BNNF) happens recursively, so that each N_i that is a compound formula on itself should also be converted, and so on and so forth until the only operators participating in the final formula are '[ ]' and ','. Commas will sometimes be omitted for the sake of compactness and whenever clearness of reading is not affected. What we now have is a global operator that partially inherits the associativity of $\wedge$, and which happens to behave like $\uparrow$ for the special case of $n = 2$.

From here on we shall refer to expressions written in BNNF as NAND expressions or NAND formulas. To better see how this works, let us return to the NAND normal form scheme (3.0.3) and convert it to BNNF:

$$\begin{aligned} (p \wedge q) &\rightleftharpoons [[p,\, q\,]] \\ (p \vee q) &\rightleftharpoons [[p\,],\, [\,q]] \\ (p \Rightarrow q) &\rightleftharpoons [\,p,\, [\,q\,]\,] \\ (p \Leftrightarrow q) &\rightleftharpoons [\ [\,p,\, q\,],\, [[\,p\,],\, [\,q]]\ ] \\ \neg p &\rightleftharpoons [\,p\,] \\ p &\rightleftharpoons [[\,p]] \end{aligned} \tag{3.1.5}$$

All we have done was to directly replace every occurrence of '$\neg$( )' by a '[ ]' and every '$\wedge$' by a ',', as indicated by (3.1.4). This set of conversion rules is denominated the NAND *conversion primitives*, and will make our life easier when translating formal logical expressions into NAND expressions. Again, note the presence of external brackets on each formula. This requisite can become somehow annoying if we want to refer to inner parts of a formula, like when saying that formula $[\,p,\, [\,q\,]\,]$ contains a p and a $[\,q\,]$. Omitting external brackets when referring to p seems, in this case, to be perfectly reasonable and even preferable. For this reason we shall relax our concept of NAND formula to allow for these cases and introduce the strict notion of a *well formed formula* (wff).

Definition 3.1.1 (Well-formed formula) *A NAND formula is a well-formed formula (*wff*) if and only if it matches one of the following:*

1. $[\ ]$
2. $[\,\alpha\,]$, *where* α *is either an atom or a wff.*
3. $[\,\alpha_1,\, \ldots,\, \alpha_n\,]$, *where* $n \in \mathbb{N}$ *and each* α_i $(i = 1, \ldots, n)$ *is either an atom or a wff.*

Note from this definition that even when α is an atom, α by itself is not a wff. At least one pair of external square brackets is required to "hold" any wff. For atomic α to become a wff we would instead have to write it as $[[\alpha]]$. Thus every wff is a NAND formula, but not the converse.

Example 3.1.2

- $[\,p,\, q,\, r\,]\,,\ \ [[\,p,\, q,\, r\,]]\,,\ \ [\cdots[[\,p,\, q,\, r\,]]\cdots]$ *are wffs.*
- $p,\, q,\, r$ *is neither a wff nor a NAND formula.*

- $[\,p\,]\,,\ \ [[\,p\,]]\,,\ \ [[[\,p\,]]]\,,\ \ [\cdots[[[\,p\,]]]\cdots]$ *are wffs.*
- $p,$ *if atomic, is a* NAND *formula in the broad sense but not a wff.*
- $[\ ]\,,\ \ [[\ ]]\,,\ \ [[[\ ]]]\,,\ \ [\cdots[[[\ ]]]\cdots]$ *are all wffs.*

In the above, dots stand for any number of bracket pairs and p*,* q *and* r *are* NAND *formulas.*

At this point the reader has probably guessed what do empty bracketed expressions like $[\ ]$ and $[[\ ]]$ mean. Indeed, they stand for *false* ($\bot$) and *true* ($\top$), respectively. This convention is not an arbitrary one but instead arises naturally within the NAND formalism. A nice way to see this is by doing the following exercise. Start by writing down the logical expression $\top \wedge N_1 \wedge N_2$, which is of course equivalent to $N_1 \wedge N_2$, together with its NAND counterpart $[[\,N_1,\,N_2\,]]$. Then progress from there by successively removing one N_i at a time, until no more $N_i's$ are left. At the final stage we should have found the NAND symbol for $\top$. The derivation is schematised below.

$$\begin{array}{lcl} \top \wedge N_1 \wedge N_2 & \rightleftharpoons & [[\,N_1,\,N_2\,]] \\ \top \wedge N_1 & \rightleftharpoons & [[\,N_1\,]] \\ \top & \rightleftharpoons & [[\ ]] \end{array} \tag{3.1.6}$$

In the same way, repeating these steps for $\neg\,(\top \wedge N_1 \wedge N_2)$ and $[\,N_1,\,N_2\,]$ leads to obtaining:

$$\begin{array}{lcl} \neg\,(\top \wedge N_1 \wedge N_2) & \rightleftharpoons & [\,N_1,\,N_2\,] \\ \neg\,(\top \wedge N_1) & \rightleftharpoons & [\,N_1\,] \\ \neg\,\top & \rightleftharpoons & [\ ] \\ \ \ \bot & \rightleftharpoons & [\ ] \end{array} \tag{3.1.7}$$

Obviously, the relation $\neg\bot = \top$ still holds since the negation of $[\ ]$ is precisely $[[\ ]]$. Similarly, $\neg\top = \bot$ should also hold, and so it does. Although negating $[[\ ]]$ introduces an extra pair of brackets, thus originating $[[[\ ]]]$, we know that $[[[\ ]]]$ amounts to $\neg\,\neg\,\bot$. And since double negations cancel out we then retrieve $\bot$. What this shows is that consecutive doubled bracket pairs are self-eliminating. In other words, every even number of inner square bracket pairs can be dropped, while odd numbered ones reduce to a single pair. As it is, this is quite an elegant feature of the NAND system. Suppose that we are given the logical expression $N_1 \wedge N_2 \wedge N_3$, and that at some stage we have the information that N_2 is a tautology. We would then replace N_2 by $\top$ in the original expression to obtain $N_1 \wedge \top \wedge N_3$. Since $\top$ is the neutral element for the $\wedge$ operator, it is absorbed by the remaining arguments, thus obtaining $N_1 \wedge N_3$. In order to do the same thing using NAND

Table 3.1:

Logical Expression	NAND Expression
$a \wedge b \wedge c$	$[\,[\,a\,b\,c\,]\,]$
$a \vee b \vee c$	$[\,[\,a\,]\,[\,b\,]\,[\,c\,]\,]$
$(a \wedge b) \vee c$	$[\,[\,a\,b\,]\,[\,c\,]\,]$
$(a \wedge b) \Rightarrow (c \wedge d)$	$[\,a\,b\,[\,c\,d\,]\,]$
$(a \wedge b) \Rightarrow (c \vee d)$	$[\,a\,b\,[\,c\,]\,[\,d\,]\,]$
$(a \vee b) \Rightarrow (c \wedge d)$	$[\,[\,[\,a\,]\,[\,b\,]\,]\,[\,c\,d\,]\,]$
$(a \vee b) \Rightarrow (c \vee d)$	$[\,[\,[\,a\,]\,[\,b\,]\,]\,[\,c\,]\,[\,d\,]\,]$

expressions, we must replace N_2 by $[[\ \ ]]$ in the initial formula $[[\,N_1,\,N_2,\,N_3\,]]$ and get $[[\,N_1,\,[[\ \ ]],\,N_3\,]]$. Now, using the self-eliminating property of double bracket pairs, the inner level $[[\ \ ]]$ can be dropped to naturally produce $[[\,N_1,\,N_3\,]]$.

We shall review the subject of bracket cancellation within the context of the simplification rules. For the moment, it is worth looking at a few examples that will help bring some intuition and mechanisation to the translation process. The table below contains some logical expressions alongside with their equivalent NAND representation.

The reader is invited to reproduce these transformations by first expressing each sentence in terms of $\neg$ and $\wedge$ and only afterwards writing them in bracket notation. This is one possible way to automate the translation method. It is advised to cancel out all double negations after the first step is performed since it will avoid a lot of extra bracketing on the final NAND formula.

An alternative way of transforming formal logical expressions into NAND expressions consists of directly applying the NAND conversion primitives displayed in scheme (3.1.5). The advantage in this case is skipping the first step of the former method. The disadvantage is that many unnecessary brackets will appear in the final resulting formula. Table 3.2 lists the same translated expressions as in Table 3.1 but this time using the later method.

As expected, Tables 3.1 and 3.2 give the same result after removing double bracket pairs. We will do in detail one of these examples using both methods.

Example 3.1.3

Method 1. *Let us consider the logical expression* $(a \wedge b) \Rightarrow (c \vee d)$. *The first step of the translation is to re-write it in* NAND *normal form as* $\neg(a \wedge b \wedge \neg\, c \wedge \neg\, d)$. *No double*

Table 3.2:

Logical Expression	NAND Expression
$(a \wedge b) \wedge c$	$[[\ [[a\,b]],\ c\ ]]$
$(a \vee b) \vee c$	$[\ [[\ [a][b]\ ]],\ [c]\ ]$
$(a \wedge b) \vee c$	$[\ [\ [[a\,b]]\ ],\ [c]\ ]$
$(a \wedge b) \Rightarrow (d \wedge e)$	$[\ [[a\,b]],\ [\ [[c\,d]]\ ]\ ]$
$(a \wedge b) \Rightarrow (c \vee d)$	$[\ [[a\,b]],\ [[\ [c][d]\ ]]\ ]$
$(a \vee b) \Rightarrow (c \wedge d)$	$[\ [[a][b]],\ [[c\,d]]\ ]$
$(a \vee b) \Rightarrow (c \vee d)$	$[\ [[a][b]],\ [[\ [c][d]\ ]]\ ]$

negations are present so we go right to the second step which is to replace negations by brackets and conjunctions by (implicit) commas where we get $[\,a\,b\,[\,c\,]\,[\,d\,]\,]$.

Method 2. *Consider the same logical expression as before,* $(a \wedge b) \Rightarrow (c \vee d)$. *Its main connective is the implication sign, so re-write it as* $\mathrm{L} \Rightarrow \mathrm{R}$, *where* L *stands for* $a \wedge b$ *and* R *for* $c \vee d$. *In* NAND *syntax implications are written as* $[\,\mathrm{L}\,[\,\mathrm{R}\,]\,]$, *where* L *is always the lhs argument and* R *the rhs argument (see (3.1.5)). In our case,* L *has the syntactic value* $[[\,a\,b\,]]$, *which means* $a \wedge b$, *while* R *is given by* $[\,[\,c\,]\,[\,d\,]\,]$, *meaning* $c \vee d$. *Now take the template* $[\,\mathrm{L}\,[\,\mathrm{R}\,]\,]$ *and replace* L *and* R *by their new values to finally obtain* $[\ [[\,a\,b\,]],\ [\ [[\,c\,][\,d\,]]\ ]\ ]$.

3.2 General Properties

A NAND formula can be viewed as a list of elements, where each element is either an atom or another list. By decomposing an initial list L into all of its elements and sub-elements, recursively, we obtain the set of *proper subformulas* of L. On a broader sense we further define the *set of subformulas* of L to be the new set $\{\text{proper subformulas of } L\} \bigcup L$. Atomic elements are also considered subformulas (or just formulas), even if, by definition, they are not wffs.

Example 3.2.1 *The set of proper subformulas of* $[[\,[\,a\,b\,]\,c\,]]$ *is the set* $\{\,[[\,a\,b\,]\,c\,],\ [\,a\,b\,],\ a,\ b,\ c\,\}$.

Each sub-formula has an associated depth level, which matches the number of subformulas that have to be extracted before reaching it. More precisely, the depth level can be calculated in the following way.

Definition 3.2.2 (Depth level) *Let X be a NAND formula and let $Y = [\alpha_1, \ldots, \alpha_n]$ $(n \geq 1)$ belong to the set of subformulas of X (which contains X). We say that a sub-formula Z of X is at a given depth level relative to X, and denote it by* $\mathrm{depth}_X(Z)$*, according to the following rules:*

1. $\mathrm{depth}_X(X) = 0$
2. $\mathrm{depth}_X(\alpha_1) = 1 + \mathrm{depth}_X(Y)$
3. $\alpha_1, \ldots, \alpha_n$ *are all at the same depth level.*

Based on this definition we shall further define the (absolute) depth of a formula as the maximum depth level of all its subformulas:

$$\mathrm{depth}(X) = \max_{Z} \{ \mathrm{depth}_X(Z) \}, \qquad Z \text{ sub-formula of } X$$

Thus, both an atom and the empty formula [ ] have zero absolute depth.

As stated by Lin Yang [26], and here restated in terms of the above definitions, we can say the following.

Definition 3.2.3 (Immediate sub-formula) *Let X be a NAND formula and Z one of its subformulas. Then Z is an immediate sub-formula of X, denoted $Z \prec X$, if and only if* $\mathrm{depth}_X(Z) = 1$*. Conversely, X is said to be the immediate ancestor or parent of Z.*

In practice, all the immediate subformulas of a general NAND expression $[N_1, \cdots, N_n]$ are $N_1, \cdots, N_n$.

Example 3.2.4 *The only immediate sub-formula of $[[\,[\,a\,b\,]\,c\,]]$ is $[[\,a\,b\,]\,c\,]$, while the immediate subformulas of $[[\,a\,b\,]\,c\,]$ are $[\,a\,b\,]$ and c .*

The concept of immediate sub-formula will become useful when creating refutation models by tree expansion (discussed later in section 3.4).

Together with depth, there are other measures like *size* and *degree*. The *size* of a NAND formula is defined to be the total count of atoms contained in it, while its degree corresponds to the bracket pair count. The concept of *degree* has been previously defined in [26] and is given again below, together with that of *size*.

Definition 3.2.5 (Degree) *The degree of a NAND formula is determined by the following rules:*

1. $\deg([\,]) = 1$
2. $\deg(\alpha) = 0$, *for atomic* α
3. $\deg([\alpha_1, \ldots, \alpha_n]) = 1 + \sum_{i=1}^{n} \deg(\alpha_i)$, *for* $n \in \mathbb{N}$.

Definition 3.2.6 (Size) *The size of a NAND formula is taken to be the total count of atom occurrences within it, over all depth levels.*

1. $\mathrm{size}([\,]) = 0$
2. $\mathrm{size}(\alpha) = 1$, *for atomic* α
3. $\mathrm{size}([\alpha_1, \ldots, \alpha_n]) = \sum_{i=1}^{n} \mathrm{size}(\alpha_i)$, *for* $n \in \mathbb{N}$.

Depth, size and degree are not perfectly correlated. For example, the expression $[p, [q, [r, s]], [t]]$ has depth 3, degree 4 and size 5, whereas $[[[p], [q], [r], [t]]]$ has the same depth, degree 6 and size 4. By making use of these parameters different formula selection strategies can be explored on view of improving efficiency.

NAND also inherits the distributivity rules for the boolean operators $\wedge$ and $\vee$ from formal logic. Let us recall them:

- Distributivity of $\vee$ over $\wedge$: $\quad c \vee (a \wedge b) \;\Leftrightarrow\; (c \vee a) \wedge (c \vee b)$
- Distributivity of $\wedge$ over $\vee$: $\quad c \wedge (a \vee b) \;\Leftrightarrow\; (c \wedge a) \vee (c \wedge b)$

If we translate both sides of these equivalences into NAND form we can directly deduce the distributivity rules for NAND.

- Rule 1: $\quad [\,[c]\,[a\,b]\,] \;\Leftrightarrow\; [[\,[[c]\,[a]]\,[[c]\,[b]]\,]]$
- Rule 2: $\quad [[\,c\,[[a]\,[b]]\,]] \;\Leftrightarrow\; [\,[c\,a]\,[c\,b]\,]$

Now the interesting thing to notice is that if we apply a small change to these rules then they can be merged into a single one which suits our purposes. Simply take Rule 1 and add one external bracket to the left hand side (lhs) while removing it from the right hand side (rhs). The equivalence is preserved and we obtain:

- Rule 1: $[[\ [c]\,[a\,b]\ ]] \ \Leftrightarrow\ [\ [[c]\,[a]]\,[[c]\,[b]]\ ]$
- Rule 2: $[[\ c\,[[a]\,[b]]\ ]] \ \Leftrightarrow\ [\ [c\,a]\,[c\,b]\ ]$

Apart from the fact that each literal in Rule 1 is the negated version of each literal in Rule 2, the two rules are now the same. It is quite neat though not surprising that this occurs, since when defining the NAND formalism the disjunctive operator got absorbed by the conjunctive one, so that in practice we are left with a single ambivalent operator, thus a single distributivity rule. We shall redefine distributivity in the following way which, we believe, might be the clearest way to read it. Nevertheless, the reader should feel free to use any other equivalent form:

- **Distributivity of NAND:** $[[\ C\,[A,\,B]\ ]] \ \Leftrightarrow\ [\ \ [C\,[A]]\ \ [C\,[B]]\ \ ]$

Here, each of A, B and C may stand for any general NAND expression, and not necessarily atoms or negated atoms. The reader should notice that if A, B and C assume the values a, b and $[\,c\,]$, respectively, as specified on the lhs of Rule 1, we then recover Rule 1. Conversely, if A, B and C assume, respectively, the values $[\,a\,]$, $[\,b\,]$ and c, then Rule 2 gets reproduced. Remember that double brackets are double negations that can cancel out inside a formula, so that $[[\,a\,]]$ reduces to a and, similarly, $[[\,b\,]]$ reduces to b. This property should also bring us attention to the fact that the lhs of the distributivity rule above is surrounded by double bracketing. Therefore, the brackets disappear whenever the expression occurs inside another formula, which is usually the case. This is indeed the significant detail that renders distributivity useful.

When adequately applied, the distributivity in NAND has the effect of raising/lowering the depth levels of targeted subformulas. This is of particular importance when performing formula simplifications, which is the subject of next section. At the same time, distributivity can reduce/increase the size of NAND formulas which is quite useful for knowledge compilation systems [7]. An example containing the application of the distributivity rule will be postponed until after some simplification techniques have been introduced.

3.3 Simplification Rules

Formula reduction can be achieved by applying systematic simplification techniques. Our first aim is to be able to provide the shortest/simplest version that a given NAND expression can adopt by trying to reduce the number of atoms there involved. Our final goal is to find either a proof or a counter model for a given set of logical sentences when they are represented as a single NAND expression. For example, given the formula $[[\,a, a, a\,]]$ meaning

$a \wedge a \wedge a$, we should obviously reduce it to its equivalent $[[\, a \,]]$, meaning a. A tautology would be reflected by the initial formula being reduced to $[[\,]]$, while a contradiction should result in $[\ ]$.

The simplification rules here described reproduce some of the well known inference rules and techniques typically used in other formal automated deduction systems. In particular, we will later pay some attention to semantic tableaux methods and the Davis Putnam procedure [3, 9, 25, 6]. Some of the rules they employ, like subsumption, Modus ponens or unit resolution and tautology removal, have also an interpretation in terms of the NAND formalism.

Double Bracket Elimination

The reader is probably by now well familiarised with the bracket simplification rules. However, they have not been properly formalised and that's what we will do here. As already mentioned, double bracket elimination is the counterpart of double negation elimination:

$$\frac{\neg \neg N}{N} \quad \rightleftharpoons \quad \frac{[[\, N \,]]}{N} \tag{3.3.8}$$

The only restriction of double bracket elimination is that it may not be applied to NAND expressions having exactly two external bracket pairs that do not occur inside another formula. Should we do so, the resulting expression would not be a wff. The important idea to retain here is that if NAND expressions are to be treated as lists then they must be globally delimited by at least one pair of outer brackets. For proper subformulas this is no longer an issue since they are already contained within a wff.

In the example below we use the term *simplified* for meaning that a formula contains no redundancies. Formal definition of simplification is given afterwards.

Example 3.3.1 *In the following examples consider N to be atomic. Then,*

- $[[\, N \,]]$ is simplified
- $[[[\, N \,]]]$ simplifies to $[\, N \,]$
- $[\ [[\, N_1,\, N_2 \,]],\ N_3\ ]$ simplifies to $[\, N_1,\, N_2,\, N_3 \,]$
- $[\ N_1,\, [[\ ]]\,,\ N_3\ ]$ simplifies to $[\, N_1,\, N_3 \,]$
- $[[\ [\, N_1 \,],\, [[\, [\, N_2,\, N_3 \,],\, [[\, N_4 \,]]\,]]\ ]]$ simplifies to $[[\ [\, N_1 \,],\, [\, N_2,\, N_3 \,],\, N_4\ ]]$

But what does $[\ N_1, [\ ]\,,\ N_3\ ]$ simplify to? Naturally, $[\ ]$ stands for $\perp$ so it will absorb every sub-formula present at the same depth level, in this case N_1 and N_3, finally

reducing to [[ ]]. Thus the whole initial expression either reduces to [[ ]] or, if it was itself a sub-formula within another larger formula, then it disappears completely because [[ ]] is self-eliminated. So any formula containing [ ] as an immediate sub-formula is a tautology and the bracket absorption rule holds:

$$\frac{[\, N_1, \cdots, [\ \ ], \cdots, N_n\,]}{[\,[\ \ ]\,]} \tag{3.3.9}$$

Definition 3.3.2 (Bracket Simplification) *A NAND formula is said to be (bracket) simplified if it remains unchanged after recursive applications of both double bracket elimination and bracket absorption rules.*

Example 3.3.3 *In the following examples consider N to be atomic. Then,*

- $[\ [[\,[\ \ ]\,N_1,\, N_2\,]],\ N_3\ ]$ simplifies to $[\,[\ \ ]\,N_3\,]$ and then to $[[\ \ ]]$
- $[\ N_1,\, [\,[\ \ ]\,N_3\,]\ ]$ simplifies to $[\,N_1,\, [[\ \ ]]\,]$ and then to $[\,N_1\,]$

Bracket simplification rules are usually immediately applied since they reduce formula depth and size.

Formula Reduction

When defining the formula simplification rules for NAND we were initially inspired by the IF-normal form of OBDDs [15]. The reduction theorem of IF basically says that once a variable has been evaluated to $\top$ (or $\bot$), then any subsequent occurrence of that variable can be replaced by $\top$ (or $\bot$). NAND also deals with Boolean logic, and the same type of reasoning can be reproduced.

Theorem 3.3.4 ($\top$ - Reduction Rule) *Consider the NAND formula $\mathcal{N} = [N_1, \cdots, N_n]$, where each N_i is a NAND formula itself, not necessarily atomic. Then every occurrence of N_i as a sub-formula of N_j $(j \neq i)$ can be removed.*

Theorem 3.3.5 ($\bot$ - Reduction Rule) *Consider the NAND formula $\mathcal{N} = [\,N_1, \cdots, [\,N_p\,], \cdots, N_n\,]$ where each N_i is a NAND formula itself, not necessarily atomic. Then every occurrence of N_p as a sub-formula of N_i $(i \neq p)$ can be replaced by $[\ ]$. In particular, if N_p is a proper sub-formula of N_i then N_i can be removed from $\mathcal{N}$.*

Proof ($\top$) Without loss of generality, because NAND expressions are commutative, take $i = 1$, $j = 2$ and N_1 a sub-formula of N_2. A trivial case is when $N_1 = N_2$. In this case the theorem is a simple re-write rule or duplicate removal, and is proved. The other case is when N_1 is a proper sub-formula of N_2. We first note that the act of deleting a sub-formula is equivalent to replacing it by [[ ]] (ie, $\top$), since double brackets are self-eliminating. So what we need to show is that, given $\mathcal{N} = [\, N_1,\, N_2,\, \cdots\,]$, the replacement N_1 by [[ ]] inside N_2 preserves equivalence to $\mathcal{N}$. Two separate situations may arise, depending on the semantic value of N_1:

i) If $N_1 \Leftrightarrow \top$ then the expression N_2' that results from N_2 by replacing all occurrences of N_1 by $\top$ is equivalent to N_2. Likewise, having $N_2 \Leftrightarrow N_2'$ and $\mathcal{N}' = [\, N_1,\, N_2',\, \cdots\,]$ leads to $\mathcal{N} \Leftrightarrow \mathcal{N}'$.

ii) On the other hand, when $N_1 \Leftrightarrow \bot$ we no longer obtain $N_2 \Leftrightarrow N_2'$, where N_2' is the same as in case i). However, we now have the equivalence $\mathcal{N} \Leftrightarrow \mathcal{N}''$, in which $\mathcal{N}'' = [\, [\ ],\, N_2,\, \cdots\,]$. Through the absorption rule $\mathcal{N}''$ reduces to [[ ]]. But so does the expression $[\, [\ ],\, N_2',\, \cdots\,]$, which is this time equivalent to $\mathcal{N}'$. Thus at the end we still obtain $\mathcal{N} \Leftrightarrow \mathcal{N}'$, and the fact that $N_2 \not\Leftrightarrow N_2'$ has vanished completely.

What we have shown is that, in every circumstance, the $\top$-reduction rule preserves equivalence to the original expression. Therefore, it is itself and equivalence rule.

($\bot$) The proof for the $\bot$ reduction rule follows similar steps and will be skipped. □

Example 3.3.6 *Consider the NAND expression* $N = [\, a,\, [\, b\,],\, [\, a,\, b,\, c\,]\,]$. *Application of the* $\top$*-rule on* a *reduces it to* $[\, a,\, [\, b\,],\, [\, b,\, c\,]\,]$. *Furthermore, application of the* $\bot$*-rule on* b *reduces it to* $[\, a,\, [\, b\,],\, [\, [\ ],\, c\,]\,]$, *which finally becomes* $[\, a,\, [\, b\,]\,]$ *by the absorption rule.*

Example 3.3.7 *Consider the NAND expression* $N = [\, [\, a,\, b\,],\, [\, [\, a,\, b\,],\, c\,]\,]$. *Application of the* $\top$*-rule on* $[\, a,\, b\,]$ *reduces it to* $[\, [\, a,\, b\,],\, [\, c\,]\,]$.

Example 3.3.8 *Consider now the NAND expression* $[\, [[\, a,\, b\,]],\, [\, [\, a,\, b\,],\, c\,]\,]$. *Application of the* $\bot$*-rule on* $[\, a,\, b\,]$ *will reduce it to* $[\, [[\, a,\, b\,]],\, [\, [\ ],\, c\,]\,]$. *At this point, the absorption rule transforms it into* $[\, [[\, a,\, b\,]]\,]$ *and the double bracket elimination further simplifies it to* $[\ a,\, b\ ]$.

In practice, application of the $\bot$-rule over compound subformulas never takes place, even if example 3.3.8 seems to use it. The only situation where it can happen is when the target sub-formula is of type $[[\, N_1,\, \cdots\,,\, N_n\,]]$. However, in these cases double bracket

elimination will have caused it to be merged with the other neighbouring subformulas before we get a chance to apply our rule. There is absolutely no problem about this because the rule can still be applied on each of the immediate subformulas $N_1, \cdots, N_n$ that have just been exposed. Retaking example 3.3.8, the initial formula $[\,[[\,a,\,b\,]],\,[\,[\,a,\,b\,],\,c\,]\,]$ gets immediately simplified to $[\,a,\,b,\,[\,[\,a,\,b\,],\,c\,]\,]$ through double bracket elimination. Now the $\top$-rule can be applied independently on a and b to obtain $[\,a,\,b,\,[\,[\,\,],\,c\,]\,]$ which again reduces to $[\,a,\,b\,]$.

Whereas the IF reduction rules only apply to atoms, we see that in NAND we can apply them to compound subformulas as well. IF-normal form expressions are always fully distributed over each atom in a Shanon expansion [1] way, so that the $\top$-$\bot$ replacements are focussed at the atomic level. This is not the case with the NAND structure which does not necessarily rely on atomic Shanon expansion.

Reduction rules are extremely effective in decreasing the formula size. It turns out that many examples can be solved by recursive application of these rules, without ever needing to go into tree expansion techniques or even make use of any other simplification rules. All three examples presented before show this since each formula reaches a state of maximum compactness where no more simplifications can take place. What we mean by maximum compactness is that each different atom only appears once and thus formula size is minimal. Nevertheless, there are examples which are more tricky and for which the $\top$-$\bot$ reduction rules are not enough. Take, for instance, the formula $[\,[\,a,\,b\,]\,[\,a,b,c\,]\,]$ corresponding to the logical expression $(a \wedge b) \vee (a \wedge b \wedge c)$. It is clear that the left term $a \wedge b$ subsumes the right term $a \wedge b \wedge c$, so that the whole formula is equivalent to $a \wedge b$ or, in bracket notation, $[[\,a,\,b\,]]$. Unfortunately, the $\top$-reduction rule does not automatically detect it unless the initial formula is given as $[\,[\,a,\,b\,],\,[\,[\,[\,a,\,b\,]\,],\,c\,]\,]$. In this case we would be able to $\top$-eliminate the term $[\,a,\,b\,]$ on the right to get $[\,[\,a,\,b\,],\,[\,[\,],\,c\,]\,]$ and consequently $[[\,a,\,b\,]]$. Since this is usually not the case, in particular due to double bracket elimination, we must deal with $[\,[\,a,\,b\,]\,[\,a,b,c\,]\,]$ as it stands.

Theorem 3.3.9 (Subsumption Rule) *A NAND expression* $\mathcal{N} = [\,N_1, \cdots, N_n\,]$ *subsumes another NAND expression* $\mathcal{S} = [\,S_1, \cdots, S_n, \cdots, S_p\,]$ *that is a proper subformula of the immediate ancestor (parent) of* $\mathcal{N}$*, occurring at any relative depth level* $\text{depth}_{\mathcal{N}}(\mathcal{S}) \geq 1$*. The subsumed formula* $\mathcal{S}$ *can be removed.*

The proof of the theorem is contained in the paragraph immediately preceding it, and relies on surrounding the sequence $N_1, \cdots, N_n$ by double brackets and then applying the $\top$-rule.

[1] Shanon expansion is defined at the end of this section.

So in the end is the subsumption rule a reduction rule or is it the converse? It so happens that reduction rules provide an abstraction to other well known deduction rules like tautology removal, subsumption and unit resolution. The subsumption rule itself is one example.

To see how this works, building an analogy with CNF will be helpful. Imagine we were given the following NAND expression:

$$[\ \ N_1\ [N_2\ [N_3\ [N_4\ [N_5\ [N_6\ [N_7 \cdots\ ]]]]]]\ ] \tag{3.3.10}$$

where, for simplicity, each N_i is assumed to be a literal. The indexes marking the N_i's represent the relative depth levels, ie, N_1 has depth level 1, N_2 has depth level 2, etc.. To make things even simpler we will abstract the outermost bracket pair and re-interpret it as a container for the set of (conjuncted) immediate subformulas. The motivation behind this arises from recalling that NAND will be used as a refutation procedure. In practice, the initial set of sentences will be negated, and therefore an extra pair of external brackets will be added. Or, if you prefer, the existing pair can be retracted. Furthermore, and most importantly, it is perfectly admissible to ignore the higher level context on which this formula occurs because our reduction rules are re-write rules, not one-way inferences. In other words, we can just plug in the external brackets at any later time.

So, returning to our line of though, we want to convert the NAND expression above into clausal form. It is clearer if we do it by steps, starting from a smaller example and progress from there to the generalisation.

Take the expression that goes up to level 3, $[\, N_1\, [N_2\, [N_3\,]\,]\,]$. Ignoring the external brackets it reads $N_1 \wedge \neg\,(N_2 \wedge \neg\, N_3)$, which has the CNF representation

$$CNF(3) \quad = \quad N_1 \wedge (\neg\, N_2 \vee N_3) \tag{3.3.11}$$

Going up to level 4 we must replace $[N_3]$ by $[\, N_3\, [N_4\,]\,]$, meaning that in (3.3.11) N_3 becomes $(N_3\ \wedge \neg\, N_4)$. The new full expression is thus $N_1 \wedge (\neg\, N_2 \vee (N_3 \wedge \neg\, N_4))$ and its CNF equivalent is given by

$$CNF(4) \quad = \quad N_1 \wedge (\neg\, N_2 \vee N_3) \wedge (\neg\, N_2 \vee \neg\, N_4) \tag{3.3.12}$$

Adding yet another level one obtains,

$$CNF(5) \quad = \quad N_1 \wedge (\neg\, N_2 \vee N_3) \wedge (\neg\, N_2 \vee \neg\, N_4 \vee N_5) \tag{3.3.13}$$

and so on and so forth. As the depth level keeps increasing, one soon recognises the following pattern:

$$\begin{aligned} & N_1 \\ & \wedge \\ & \neg N_2 \vee N_3 \\ & \wedge \\ & \neg N_2 \vee \neg N_4 \vee N_5 \\ & \wedge \\ & \neg N_2 \vee \neg N_4 \vee \neg N_6 \vee N_7 \\ & \vdots \end{aligned} \tag{3.3.14}$$

The first thing to notice is that each odd level term gives rise to a new independent sentence, while even level terms are progressively concatenated onto each clause. Therefore, we can interpret odd levels as of conjunctive type, and even levels as disjunctive. Note as well that each even level term becomes negated when transposed to clausal form. That is how the bracket operator is able to metamorphose between the conjunctive operator and the disjunctive operator — it pushes negations inwards as the nesting deepens.

If it is the case that the N_i's can be unified, then we observe the following:

1. odd levels can *resolve* with even levels
2. odd levels can create *tautologies* with even levels
3. odd levels can *subsume* other odd levels
4. even levels can be *factored*

For example,

(1) Under the unification $N_2 = N_1$, we have that, in (3.3.14), the sentence composed of the isolated term N_1 can be unit-resolved with each of the other sentences, because they all contain the negated literal $\neg\, N_2 = \neg\, N_1$. In the $\top$-reduction rule this would amount to deleting N_2 from (3.3.10):

$$\begin{aligned} & [\ \ N_1\ [N_1\ [N_3\ [N_4\ [N_5\ [N_6\ [N_7\ \cdots]]]]]]\ ] \\ & (\top\text{-reduction rule on } N_1) \qquad \updownarrow \\ & [\ N_1\, N_3\ [N_4\ [N_5\ [N_6\ [N_7\ \cdots]]]]\ ] \end{aligned} \tag{3.3.15}$$

giving the same CNF expression as after the unit-resolution steps:

$$N_1 \ \wedge \ N_3 \ \wedge \ (\neg N_4 \vee N_5) \ \wedge \ (\neg N_4 \vee \neg N_6 \vee N_7) \ \wedge \ \ldots \tag{3.3.16}$$

For this particular example there seems to be an advantage of the $\top$-reduction rule for NAND over clausal form, because a single deletion operation achieves the same as a series of unit-resolution steps.

(2) Now, for instance, having $N_5 = N_7$ would cause $(\neg N_4 \vee N_5)$ to subsume $(\neg N_4 \vee \neg N_6 \vee N_7)$. Again, this would have amounted to deleting N_7 from (3.3.10). But we see now that despite the fact that this is a similar $\top$-reduction operation as before, the outcome is different when transposed to clausal form.

(3) If instead we have $N_5 = N_4$, then the sentence $(\neg N_4 \vee N_5)$ becomes a tautology.

(4) Finally, for $N_4 = N_6$, the expression $(\neg N_4 \vee \neg N_6 \vee N_7)$ produces the (ground) factor $(\neg N_4 \vee N_7)$, which then subsumes it[1].

The above examples try to give an idea of how our reduction rule simulates some common inference rules. Note, however, how a general resolution step never actually occurs, although, for instance, setting $N_5 = N_6$ may seem to indicate it:

$$\frac{(\neg N_4 \vee N_5), \quad (\neg N_4 \vee \neg N_6 \vee N_7)}{(\neg N_4 \vee N_7)} \tag{3.3.17}$$

In fact, since the term $\neg N_4$ is present in both premises, what we have is

$$\neg N_4 \ \vee \ (N_5 \wedge (\neg N_6 \vee N_7))$$

and we are simply applying unit-resolution on the left argument of the main disjunction. This is of course consistent with the fact that reduction rules are equivalence preserving while general resolution is sound but only refutation complete, so that the two could never be compared.

We should point out the fact that, up to here, we have been interpreting each N_i as a positive literal, so that only the $\top$-reduction rule was being applied. If literals happen to be negative then the words "odd" and "even" would possibly need to be interchanged to obtain the correct correspondences between the reduction rules and the resolution, factoring and subsumption operations. Nevertheless, the main structure of the sentences on

[1]Ground factoring is commonly referred to as *merging* because it is based on the application of the idempotency law: $N \vee N \ \simeq \ N$.

scheme (3.3.14) would still prevail, with negations being switched on and off where appropriate. Clausal form representation of NAND expressions is thus determined by the depth levels of its subformulas measured up to literals. More precisely, given the expression $[a, [b], [c, d]]$, it is more useful to say that both a and $[b]$ have depth level 1 and c and d have depth level 2, rather than saying that a has depth level 1 while b, c and d have depth level 2.

We have seen so far various types of formula simplifications. Soon it will be important to have a proper distinction between NAND expressions and *simplified* NAND expressions, and to know exactly what we mean by that. With that in mind, we introduce below our standard notion of formula simplification.

Definition 3.3.10 (Simplified Formula) *A NAND expression is said to be simplified if it is bracket simplified and the $\top$-$\bot$ reduction rules have been applied, at least at the literal level.*

So our general simplification procedure is not fully extended. That is, the subsumption rule may not have been applied yet nor the $\top$-$\bot$ reduction rules that act on compound subformulas (of which the subsumption rule is a special case). In situations where we wish to point out that simplification is extended to compound subformulas as well, we shall sometimes use the term *compound-simplified*. Any other cases shall be referred to explicitly.

We are now ready to return to our distributivity rule and on the way present Shanon expansion in the context of NAND. We shall see that distributivity is a special case of Shanon expansion.

Shanon Expansion

When all other things fail, we can always rely on Shanon expansion... Objectively, whenever there are no conditions to directly apply any of the previous rules — $\top$-$\bot$ reduction, subsumption or distributivity — we may try to apply Shanon expansion. In fact, all of those rules end up being special cases of localised Shanon expansion for which the formula is guaranteed to reduce in size. The definition of Shanon expansion we present here is adapted for NAND expressions, but it is obviously equivalent to the usual definition also given in [11]:

Definition 3.3.11 (Shanon Expansion) *Given a NAND expression N and an atom p, the Shanon expansion of N with respect to p, denoted $SE(N, p)$, is given by the following*

transformation:

$$SE(N,p) \;=\; [\;\; [p\;\; N_{\top/p}\,]\;\; [\,[p\,]\;\; N_{\bot/p}\,]\;\;] \tag{3.3.18}$$

where $N_{\top/p}$ and $N_{\bot/p}$ are the expressions obtained from N after replacing every occurrence of p by $\top$ or $\bot$, respectively. In practice, $\top$ means $[[\;\;]]$ and $\bot$ means $[\;]$.

First of all, we should point out a few things: i) Shanon expansion does not impose any pre-conditions on N; ii) It is perfectly possible to extend its definition so that p is not necessarily an atom (basically, we just have to omit that one condition in the former definition, all the rest being kept the same); iii) Application of the $\top - \bot$ reduction rules on $[\,[p\,N\,]\,[\,[p\,]\,N\,]\,]$, restricted to act on p and $[\,p\,]$ only, gives exactly $SE(N,p)$; iv) Last, but not least, N and $SE(N,p)$ are logically equivalent.

Our final remark is straightforward, but perhaps easily understood if we take a look at the corresponding classical logical representation:

$$SE(N,p) \;=\; (p \wedge N_{\top/p}) \vee (\neg p \wedge N_{\bot/p}) \tag{3.3.19}$$

This should need no more explanations.

With respect to the first remark, even when p is not a sub-formula of N, Shanon expansion can still be applied (although, unless required by a specific procedure, we would probably be reluctant to do so since it would just duplicate the formula size). Usually, it will be combined with the former simplification rules to produce the best results. The following example shows step by step how this can be achieved.

Example 3.3.12 *Let $N = [\,[\,p\,[\,q\,[\,s\,]\,]\,]\;,\;[\,q\,[\,p\,s\,]\,]\,]$. Neither distributivity nor the $\top$-$\bot$ reduction rules can be applied on N. However, we have:*

$$\begin{aligned}
N_{\top/p} &= [\,[\,[[\;]]\,[\,q\,[\,s\,]\,]\,]\;,\;[\,q\,[\,[[\;]]\,s\,]\,]\;\;] \\
&= [\;\; q,\;[\,s\,],\;\;[\,q\,[\,s\,]\,]\;\;] && (\text{ double bracket elimination }) \\
&= [\;\; q,\;[\,s\,],\;\;[\,q\,]\;\;] && (\text{ }\top\text{-rule on }[\,s\,]\text{ }) \\
&= [\;\; q,\;[\,s\,],\;\;[\;]\;\;] && (\text{ }\top\text{-rule on } q\text{ }) \\
&= [[\;]] && (\text{ bracket absorption rule })
\end{aligned}$$

and

$$\begin{aligned}
N_{\bot/p} &= [\,[\,[\;]\,[\,q\,[\,s\,]\,]\,]\;,\;[\,q\,[\,[\;]\,s\,]\,]\,] \\
&= [[\,q\,]] && (\text{ bracket simplification })
\end{aligned}$$

so that

$$\begin{array}{rcll} SE(N,p) & = & [\,[p,[[\,]]\,]\,\,[\,[p\,],\,[[q\,]]\,]\,] & \\ & = & [\,[p\,]\,\,[\,[p\,]\,q\,]\,] & (\text{ bracket simplification }) \\ & = & [\,[p\,]\,\,[q\,]\,] & (\text{ }\top\text{-rule on }[p\,]\,) \end{array}$$

This example was a good illustration of how the simplification techniques can be very effective. The size of N decreased to less than half its original size. Also note how the s has disappeared in $SE(N,p)$. Whenever this happens we say that s was a *redundant* variable in N, because the logical value of N did not depend on s. Generalising, whenever two equivalent expressions do not agree on their variable sets, any variable that does not belong to the intersection of both sets is redundant.

The next example will show how the distributivity law can be obtained by using Shanon expansion.

Example 3.3.13 *Consider the following NAND expression* $N = [\,[p,\,q\,][p,\,r\,][p,\,s\,]\,]$. *The atom* p *is a common element of each immediate sub-formula of* N*, so that the distributivity rule yields* $N = [[\;p\;\;[\,[q\,]\;[r\,]\;[s\,]\,]\;\;]]$. *Now Shanon expansion of* N *with respect to* p *will give:*

$$\begin{array}{rcll} N_{\top/p} & = & [\;\;[\;[[\,]]\;q\;]\;\;[\;[[\,]]\;r\;]\;\;[\;[[\,]]\;s\;]\;\;] & \\ & = & [\,[q\,]\;[r\,]\;[s\,]\,] & (\text{ double bracket elimination }) \end{array}$$

$$\begin{array}{rcll} N_{\bot/p} & = & [\;\;[\;[\,]\;q\;]\;\;[\;[\,]\;r\;]\;\;[\;[\,]\;s\;]\;\;] & \\ & = & [\;\;] & (\text{ bracket simplification }) \end{array}$$

Due to $N_{\bot/p} = [\;]$, *the* $[p\,]$ *term of* $SE(N,p)$ *is absorbed and we get:*

$$SE(N,p) \quad = \quad [[\;p\;\;[\,[q\,]\;[r\,]\;[s\,]\,]\;\;]]$$

In the previous examples the variable used to factor on, ie p, was given. This is usually the case in automated procedures where a specific variable ordering is pre-defined — like in computing unique representations of logical expressions, as is the case of OBBD's [15] or OFNNF's [11]. However, blind choice of variables will not generally lead to the most compact representation. If the main goal is decreasing the formula size, then the expansion should be focused on particular subformulas and on carefully chosen variables

(either a priori or a posteriori) where the compactness effect can be maximal. Shanon expansion has also wide application in knowledge compilation problems like DNNF's [8] and path dissolution [17]. In [8] they define a pair of transformations denoted *conjoining* and *conditioning* which can be combined to reproduce Shanon expansion. And in [17], the path dissolution technique used to build full dissolvents (weaker versions of DNNF's) and DNNF's is based on semantic factoring (another name for Shanon expansion) and the Prawitz Rule (a special case of Shanon expansion).

The NAND simplification rules along with Shanon expansion are enough to derive a proof or a model for a ground theory. But the NAND system can also be implemented as a tableaux method, provided extension rules are appropriately defined for BNNF. This is what will be described in the next section.

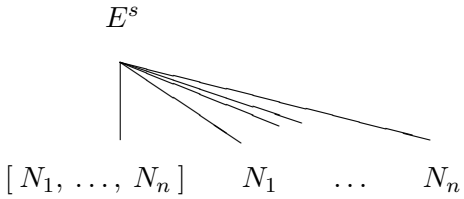
3.4 Tree Expansion

The sentential language (alias NAND) introduced by Sandqvist [23] was originally implied in the building of a proof system in the style of tableaux methods — the C^s-system. Here, we will couple the simplification rules defined in the previous section with the original C^s-rules. As a remark on notation, through the rest of the text we shall indistinguishably use the terms NAND -tree and C^s-tree. We will also use the term C^s-proof to indicate that a theorem has been proved by means of a C^s-tree.

A NAND tree is a finite branching tree whose nodes are NAND expressions. The tree is extended by means of a single *extension rule* E^s:

$$\frac{}{[\,N_1,\,\ldots,\,N_n\,]\;\Big|\;N_1\;\Big|\;\ldots\;\Big|\;N_n} \tag{3.4.1}$$

where N_i $(i = 1, \ldots, n)$ is an arbitrary atom or compound BNNF formula. When an E^s extension is applied to a branch, the $n+1$ new nodes featured in (3.4.1) are added below the leaf node of that branch:



We also refer to E^s as the *split rule*, where $[N_1, \ldots, N_n]$ is called the *major component* of the split. The *minor components* are its immediate subformulas $N_1, \ldots, N_n$. When

we say a "*split on* ϕ", we mean a split with major component ϕ, and when we write "a split $[\psi_1, \ldots, \psi_n]$" we identify a split with components (major and minor) $\psi_1, \ldots, \psi_n$.

Figure 3.1 illustrates a NAND tree with root $[a\,[b\,c]]$ that has been extended by a split on $[c]$ followed by a split on $[a\,b]$ rooted at node c:

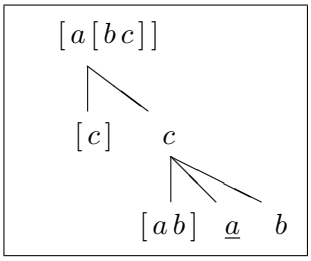


Figure 3.1: C^s-tree for $[a\,[b\,c]]$ extended by two splits.

Note that, for $n = 1$ and for atomic N_1, the E^s extension rule becomes equivalent to the PB rule of KE-tableaux and the atomic cut of FNNF-tableaux. Moreover, the E^s-rule also replaces both the α and β rules of analytic tableaux when the major component of the split is an immediate subformula of any of the nodes along the branch. As we shall see further on, this is due to the intrinsic structure of BNNF formulas and the dual nature of the bracket operator.

A tree grows downwards as splits are successively added to leaf nodes. *Levels* increase by one at each extension, with the root node at level 0. So, in Figure 3.1, the tree has three levels, with $[c]$ and c at level 1 and $[a\,b]$, a and b at level 2.

The node at the origin of a split is also called a *parent* node relative to the split components, which are themselves its *immediate descendants*. In relation to a given node, the nodes above it that lie on the same branch are its *ancestors*, while the nodes below it are its *descendants*. The immediate descendants of a node are also said to be *siblings*, as well as the subtrees descending from them. And it seems the family is complete! So, back to Figure 3.1, we see for example that b has c and the root as its ancestors but has no descendants, while c has three descendants.

A tree will stop growing if all its branches are *closed*. The closure of a branch is specified by the following *closure rule*.

Definition 3.4.1 (Closure Rule) *A branch of a C^s-tree is closed if it contains two nodes n_i and n_j such that $n_i \prec n_j$. The levels at which the nodes occur is indifferent, n_i may be a descendant or an ancestor of n_j. A tree is closed if all its branches are closed. Otherwise, if it contains at least one open branch, the tree is said open.*

Recall that $n_i \prec n_j$ means that n_i is an immediate subformula of n_j. Therefore, in Figure 3.1, the branch ending at leaf node a is closed because $a \prec [\, a\, [\, b\, c\,]\,]$. That is why it has been underlined. Closed branches are marked with underlines to indicate they cannot be extended.

The smallest open tree is the single root node $[\quad]$. This tree can never be closed. The formal proof of this statement is entailed by the soundness of the C^s-system which will be later demonstrated. For the moment, we might accept the following intuitive argument. Suppose we extend it with a split on $[\, N\,]$. Then two open branches with leafs $[\, N\,]$ and N are created. To close $[\, N\,]$ we must extend it with a split that has an N component. It is useless to add another split on $[\, N\,]$ because we would be left with the same problem. Therefore, a split on $[\, M\ N\,]$, for example, could be added, producing the immediate descendants $[\, M\ N\,]$, N and M. The branch marked with N would get indeed closed, but at the cost of generating two other open branches. No matter how we choose the splits, new open branches will continue to be generated!

On the other hand, the smallest closed tree is the one with root node $[\,[\quad]\,]$:

$$\begin{array}{c} [\,[\quad]\,] \\ \diagdown \\ \underline{[\quad]} \end{array} \tag{3.4.2}$$

Since $[\quad]$ has no immediate subformulas, the split is unary. We conclude that the tree with root node $\bot$ is open, while the tree with root node $\top$ is closed. This is a hint on how the C^s-proof system works — that valid formulas have closed trees and the others have open trees.

The closure rule is motivated by the following argument, which also explains how to read a NAND tree. Each node that is added to the tree is assumed false. Nodes along a branch are conjoined, while sibling nodes or sibling subtrees are disjoined. Explicitly, suppose $\mathcal{F} = [\, N_1\ \ldots\ N_n\,]$ is a node of the tree being extended by a split $\mathcal{S}$ with major component $\mathcal{S}_M = [\, \phi_1\ \ldots\ \phi_p\,]$ and minor components $\phi_1, \ldots, \phi_p$, as depicted in Figure 3.2.

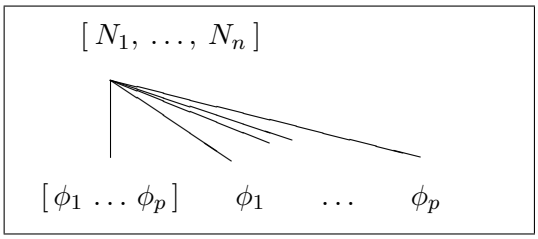


Figure 3.2: C^s-split on $\mathcal{S}_M = [\, \phi_1\ \ldots\ \phi_p\,]$, rooted at $\mathcal{F} = [\, N_1,\ \ldots,\ N_n\,]$.

Then what we must read is:

$$\begin{array}{c} \neg\mathcal{F} \ \wedge \ (\neg\mathcal{S}_M \ \vee \ \neg\phi_1 \ \vee \ \ldots \vee \neg\phi_p\,) \\ \Updownarrow \\ [[\quad N_1,\ldots,N_n,\,[\,\mathcal{S}_M,\ \phi_1,\ \ldots,\ \phi_p\,]\quad]] \end{array}$$

Therefore, we are adding to the assumption $\neg\mathcal{F}$ the new assumption $\mathcal{S} = [\mathcal{S}_M,\ \phi_1,\ \ldots,\ \phi_p\,]$. But $\mathcal{S}$ itself is, by construction, a tautology, so each split as a whole is not really an assumption. To see that $\mathcal{S}$ is indeed a tautology it is enough to apply to it the $\top$-reduction rule on each ϕ_i:

$$\begin{aligned} \mathcal{S} &= [\,\mathcal{S}_M,\ \phi_1,\ \ldots,\ \phi_p\,] \\ &\equiv [\,[\,\phi_1\ldots\phi_p\,]\,\phi_1,\ \ldots,\ \phi_p\,] \\ &= [\,[\quad]\,\phi_1,\ \ldots,\ \phi_p\,] \\ &= [[\quad]] \end{aligned} \tag{3.4.3}$$

This ensures that the only real assumption made in the tree is the one that the root node is false. The root node was not originated by a split, it is a premise.

By distributing over each branch, the branches can be analysed independently of each other. Below, each conjoined set of terms corresponds to a branch, and the $p+1$ branches are disjoined:

$$\begin{array}{c} (\neg\mathcal{F}\ \wedge\ \neg\mathcal{S}_M\,)\ \vee\ (\neg\mathcal{F}\ \wedge\ \neg\phi_1\,)\ \vee\ \ldots\vee\ (\neg\mathcal{F}\ \wedge\ \neg\phi_p\,) \\ \Updownarrow \\ [\quad [\,[\mathcal{F}]\,[\mathcal{S}_M]\,]\quad [\,[\mathcal{F}]\,[\phi_1]\,]\ \ldots\ [\,[\mathcal{F}]\,[\phi_p]\,]\quad] \end{array}$$

Along a single branch each new node inputs an assumption that is no longer tautologous in relation to that branch. For example, along the branch with nodes $\mathcal{F}$ and ϕ_1, we are making the assumption $\neg\mathcal{F}$ and $\neg\phi_1$. Consider now the case were $\phi_1 = N_1$, such that $\phi_1 \prec \mathcal{F}$. If $\phi_1 = false$, then $N_1 = false$. But we have also assumed that $\mathcal{F} = false$, which means that each N_i must be true. This contradicts the assumption on this branch that $\phi_1 = N_1 = false$. Therefore, whenever in a given branch a node ϕ_1 is an immediate subformula of another node $\mathcal{F}$, a conflict arises that creates an inconsistency. The branch is thus closed. If all branches are inconsistent, then it must be the case that $\neg\mathcal{F}\ \wedge\ (\neg\mathcal{S}_M\ \vee\ \neg\phi_1\ \vee\ \ldots\vee\neg\phi_p\,)$ is also inconsistent. But since the split as a whole is a tautology, it is $\neg\mathcal{F}$ that must be inconsistent, meaning that $\mathcal{F}$ is valid.

Corollary 3.4.2 (Soundness and Completeness) *Let $\mathcal{F}$ be a NAND expression. A closed C^s-tree for $\mathcal{F}$, denoted $C^s(\mathcal{F}) \vdash \bot$, exists iff $\mathcal{F}$ is a valid (tautologous) formula. Where $C^s(\mathcal{F})$ refers to a C^s-tree with root $\mathcal{F}$.*

Therefore, if a BNNF formula $\mathcal{F}$ is a contingency or a contradiction, $C^s(\mathcal{F})$ is always an open tree. The above corollary translates the soundness and completeness of the C^s-system. Its proof entailed by Theorems 3.4.12 and 3.4.13 that will be demonstrated later on. To convince ourselves for the moment, here is an example of two distinct C^s-proofs for the same valid logical expression.

Example 3.4.3 *Consider the valid logical expression* $\mathcal{F} = ((p \vee q) \wedge (p \Rightarrow q)) \Rightarrow q$ *with* $\mathrm{BNNF}(\mathcal{F}) = [\ [[p][q]]\ \ [p[q]]\ \ [q]\ ]$. *Two possible* C^s*-proofs for* $\mathcal{F}$ *are shown in Figures 3.3 and 3.4.*

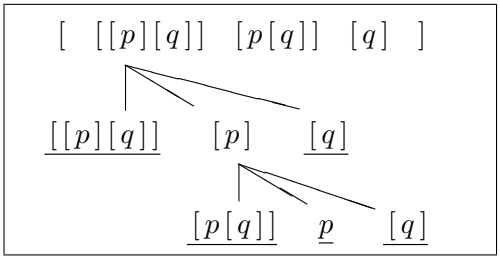


Figure 3.3: Strongly analytic C^s-proof for $\mathcal{F}$.

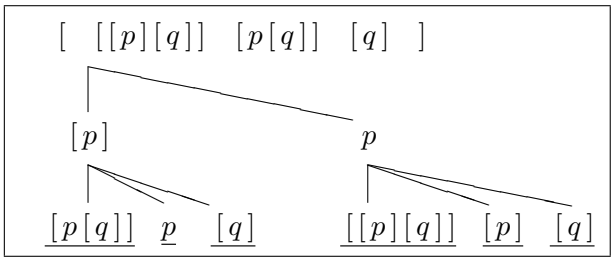


Figure 3.4: Weakly analytic C^s-proof for $\mathcal{F}$.

We can also read the tree in, for example, Figure 3.3 as follows. We start by assuming that the root node is false. This means that each of its immediate subformulas ψ_i must be true. We will test one at a time. First, we pull down formula $\psi_1 = [[p][q]]$ and assume it is false. Naturally, as we said, ψ_1 must be true and so that branch is closed by inconsistency. But if ψ_1 is true, than at least one of its immediate subformulas ϕ_i must be false. Here we consider $\phi_2 = [q]$ first, and assume it to be false. Since it is an immediate subformula of the root, we have already seen that it must be true. Again, this branch closes. The other possibility is for $\phi_1 = [p]$ to be false. This assumption does not explicitly conflict with any of the previous assumptions, so the branch stays open with the assignment $\phi_1 = [p] = false$. Next, we pull down $\psi_2 = [p[q]]$. Again, we assume it to

be false and obtain an inconsistency that immediately closes that branch. The process is repeated. For ψ_2 to be true, at least one of its immediate subformulas must be false. But neither p or $[\,q\,]$ can be false because we already have $[\,p\,] = false$ and $[\,q\,] = true$. All the branches are now closed, which means that the initial assumption was wrong.

There is a particularity about the C^s-trees of Figures 3.3 and 3.4 which is the fact that each split was constructed using subformulas of nodes above the split. This is not imposed by the extension rule but it is sensible that we do so in a refutation proof. A categorisation of C^s-trees according to the type of extensions they allow is accomplished by the following definitions.

Definition 3.4.4 (Strong Analyticity) *An application of the E^s extension rule on a leaf node ϕ of an open branch θ is strongly analytic if its major component is an immediate subformula of any of its ancestor nodes. That same extension will not be repeated again below ϕ. If all applications of E^s are strongly analytic, then the C^s-tree is also strongly analytic.*

Definition 3.4.5 (Weak Analyticity) *An application of the E^s extension rule on an open branch θ is weakly analytic if any one of the following two conditions holds:*

- *If the major component of the split is a proper subformula of any of its the ancestor nodes;*
- *Or if the split has major component $[\,\psi\,]$ and ψ is a proper subformula of any of its ancestor nodes.*

When all applications of E^s are weakly analytic then the C^s-tree is also said to be weakly analytic.

Under these definitions, a strongly analytic C^s-tree is also weakly analytic. The C^s-tree of Figure 3.4 is not strongly analytic because on the first split, $[\,[\,p\,]\,p\,]$, the major component $[\,p\,]$ is not an immediate subformula of the root node, its only ancestor. But it satisfies the condition of weak analyticity: the major component $[\,p\,]$ is a proper subformula of the root. In fact, the second condition is also satisfied because p is also a proper subformula of the root. But, of course, satisfying one of the two conditions would have been enough. Note as well how in a strongly analytic tree the major component of each split is immediately closed.

Analyticity is important for deciding when to stop growing a tree. A tree that is closed immediately guarantees that the root is a valid formula. However, even if the root node is valid, it is still possible to keep extending the tree indefinitely. All we have to do is

to choose totally unrelated splits, and that will easily prevent the closure rule form being applied to all or at least some of the branches. Therefore, some degree of connectivity between the nodes is required to achieve closure, if one exists.

To that purpose, the following properties are defined.

Definition 3.4.6 (Saturated Branch) *A branch of a C^s-tree is saturated if all possible strongly analytic extensions have been applied to it.*

Definition 3.4.7 (Complete C^s-tree) *A C^s-tree is complete if it is closed or if all its open branches are saturated.*

Completeness alone does not exclude non-analytic extensions. But it guarantees that if some node N of an open tree contains an immediate compound subformula S, then S must explicitly appear in the subtree rooted at N as a major component of a split. Below, we state in advance the completeness theorem for the C^s-system that will be proved and restated as Theorem 3.4.13.

Theorem 3.4.8 *If an open C^s-tree is complete, then its root node is either a contingency or a contradiction.*

A simple example of a complete open C^s-tree for the contingent logical expression $p \Rightarrow (p \wedge q)$ is shown in Figure 3.5.

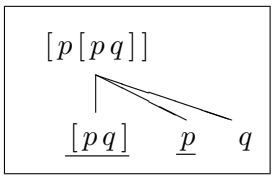


Figure 3.5: Complete strongly analytic open C^s-tree.

Note how the atom p at the root node has not been extended — the extension rule is only defined for wffs. If we do want to split on an atom p, then the major component must be $[[p]]$, instead of p, but which is nonetheless equivalent to p. The minor component would then be $[p]$. Otherwise, we may always split on $[p]$ and obtain p as a minor component. This produces a semantically equivalent split but is preferable because p as depth level 0 while $[[p]]$ has depth level 2.

Another example of a complete (strongly analytic) open tree is illustrated in Figure 3.6. The branches ending at the leaf nodes $[p]$ and s are open and saturated.

The example shown before in Figure 3.1 depicts an incomplete tree because the immediate subformula $[bc]$ at the root node has not been extended. The tree is also not

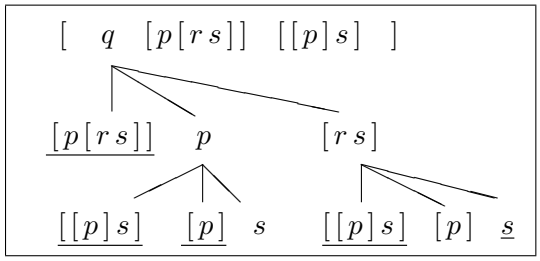


Figure 3.6: Complete open C^s-tree for the logical expression $q \Rightarrow ((p \wedge (\neg r \vee \neg s)) \vee (\neg p \wedge s))$.

analytic because $[\,a\,b\,]$, the major component of the second split, is not a proper subformula of any of the nodes above it.

When a complete analytic C^s-tree for $\mathcal{F}$ is open, the *counter models* for $\mathcal{F}$ can be extracted. A *counter model* for $\mathcal{F}$ is an interpretation over the atoms of $\mathcal{F}$ that falsifies $\mathcal{F}$. The following steps indicate how to obtain the counter models for $\mathcal{F}$:

Step 1. Assign to each atomic node in an open branch the value $\bot$;

Step 2. Assign to each atom that is an immediate subformula of a node in an open branch the value $\top$;

Step 3. For each open branch, conjoin all the assignments made in steps 1 and 2, ;

Step 4. Disjoin all the conjoined sets from step 3.

where $\top = true$ and $\bot = false$. Atoms that have not been assigned in steps 1 and 2 for a given branch are redundant in that branch and can be assigned either $\top$ or $\bot$.

Under steps 1 to 3, each branch yields a distinct counter model for $\mathcal{F}$. When all counter models are gathered, by step 4, a DNF model for $\neg\mathcal{F}$ results. If a tree for $\mathcal{F}$ is closed than it has no counter models and DNF $(\neg\mathcal{F}) = [\quad]$. Recall from equation (3.4.3) that each split $\mathcal{S} = [S_M\, S_1, \ldots\, S_n]$ that is added to $\mathcal{F}$ is a tautology. Conjoining node $\mathcal{F}$ with the split is equivalent to $\neg\mathcal{F} \wedge (\neg S_M \vee \neg S_1 \ldots \vee \neg S_n)$, which is of course still equivalent to $\neg\mathcal{F}$. Therefore, no matter how many splits we add, we always end up with an expression that is equivalent to $\neg\mathcal{F}$. Distributing over all $\vee$'s, which is what we do by looking first along each branch and then across all branches, produces a DNF, which must be DNF $(\neg\mathcal{F})$. Closed branches are not considered because they represent inconsistent paths in the DNF. Since a DNF is a set of disjunctions, the inconsistent symbol $false$ is absorbed by the other disjuncts.

Example 3.4.9 *In Figure 3.5 there is only one open branch with nodes* $[\,p\,[\,p\,q\,]\,]$ *and* q*. By step 1,* $q = \bot$ *and, by step 2,* $p = \top$*. Following step 3 we have the conjunction* $p \wedge \neg q$*. Since there are no more open branches, the only counter model for* $\mathcal{F} = [\,p\,[\,p\,q\,]\,]$ *is* $p \wedge \neg q$*. We can see that this is also equivalent to* $\neg\mathcal{F} = [[\,p\,[\,p\,q\,]\,]]$ *once the* $\top$*-reduction rule is applied on* p *to give* $\neg\mathcal{F} = [[\,p\,[\,q\,]\,]] \equiv p \wedge \neg q$*.*

Example 3.4.10 *In Figure 3.6 there are two open branches. The left most contains the two atomic nodes* p *and* s *and the atom* q *which is an immediate subformula of the root. Therefore,* $p = s = \bot$ *and* $q = \top$*, such that the first counter model is* $[[\,q\,[\,p\,]\,[\,s\,]\,]]$ *(i.e.,* $q \wedge \neg p \wedge \neg s$ *). The other open branch contains* p*,* r*,* s *and* q *all of which are immediate subformulas of their nodes. Therefore we have* $p = q = r = s = \top$ *which, by step 3, corresponds to the counter model* $[[\,p\,q\,r\,s\,]]$ *(i.e.,* $p \wedge q \wedge r \wedge s$*). Hence, by step 4, the disjunction of the counter models is given by* $[\;\;[\,p\,q\,r\,s\,]\;\;[\,q\,[\,p\,]\,[\,s\,]\,]\;\;] = \mathrm{DNF}(\neg\mathcal{F})$*. Remark that this is indeed the DNF representation of* $\neg\mathcal{F}$*. To confirm it, we shall apply to* $\neg\mathcal{F} = [[\;\;q\;\;\;[\,p\,[\,r\,s\,]\,]\;\;\;[\,[\,p\,]\,s\,]\;\;]]$ *a Shanon expansion on* p *followed by simplification:*

$$\begin{aligned}
\neg\mathcal{F}_{\top/p} &= [[\;\;q\;\;\;[[\,r\,s\,]]\;\;\;[\,[\;\;]\,s\,]\;\;\;]] \\
&= [[\;\;q\,r\,s\;\;\;]]
\end{aligned} \tag{3.4.4}$$

$$\begin{aligned}
\neg\mathcal{F}_{\bot/p} &= [[\;\;q\;\;\;[\,[\;\;]\,[\,r\,s\,]]\;\;\;[\,[[\;\;]]\,s\,]\;\;\;]] \\
&= [[\;\;q\;\;\;[\,s\,]\;\;\;]]
\end{aligned}$$

$$\begin{aligned}
\mathrm{SE}\,(\neg\mathcal{F}, p) &= [\;\;[\,p\;\;\neg\mathcal{F}_{\top/p}\,]\;\;\;[\,[\,p\,]\;\;\neg\mathcal{F}_{\bot/p}\,]\;\;] \\
&= [\;\;[\,p\;\;[[\;\;q\,r\,s\;\;\;]]\,]\;\;\;[\,[\,p\,]\;\;[[\;\;q\;\;\;[\,s\,]\;\;\;]]\,]\;\;] \\
&= [\;\;[\,p\,q\,r\,s\,]\;\;\;[\,[\,p\,]\,q\,[\,s\,]\,]\;\;] \\
&= \mathrm{DNF}\,(\neg\mathcal{F})
\end{aligned}$$

Steps 1 and 2 derive from and are equivalent to assigning each node of an open branch to $false$. However, because the tree is complete, all compound subformulas have been split into smaller components, recursively, up to the literal level. Consequently, all atoms in that branch are also present as either a node or as an immediate subformula of a node. Therefore, it is enough to look at those atomic assignments. If an atom p is a node, it is assigned $\bot$. If it is an immediate subformula of a node N, and since N must be assigned $\bot$, then p must be assigned $\top$. Remember that for an arbitrary BNNF formula $[N_1, \ldots, N_n]$ to be false, each immediate subformula N_i must be true. The fact that these assignments do not conflict with each other is guaranteed by the closure rule. Whenever

two nodes n_i, n_j appear in the same branch and are connected by $n_i \prec n_j$, then obviously they cannot both be assigned $\bot$. And in that case the branch is closed. The formal proof of these arguments is given as part of the proof for theorem 3.4.13. In that proof, the following definition will be used.

Definition 3.4.11 ($\bot$-Inconsistency) *Two nodes in an open branch of a C^s-tree are $\bot$-inconsistent if they cannot be simultaneously interpreted as $\bot$. Otherwise, they are $\bot$-consistent.*

We can see that the NAND -tree differs from general tableaux in the sense that in a tableaux each node of an open branch has a truth interpretation, while in NAND it has a false interpretation. But that is why a NAND -tree is closed if the root node is valid, while a tableaux closes when it is unsatisfiable. And that is also why a tableaux for $\mathcal{F}$ gives a DNF representation of $\mathcal{F}$, while a C^s-tree for $\mathcal{F}$ yields a DNF representation of $\neg\mathcal{F}$.

In fact, any NNF analytic tableaux for $\mathcal{F}$ is equivalent to a strongly analytic NAND -tree for $\neg\mathcal{F}$. Figure 3.7 shows two partial trees. A NAND-tree on the left and an NNF tableaux on the right. For the NAND -tree, the node at the origin of the split is $\neg\mathcal{F} = [\,N_1, \ldots, N_n\,]$, and the major component of the split is $M = [\,\phi_1, \ldots, \phi_p\,]$. The equivalent origin node on the tableaux is marked with $\mathcal{F} = N_1 \wedge \ldots \wedge N_n$, and the β-rule is also applied on $M = \neg\phi_1 \vee \ldots \vee \neg\phi_p$. If we wish, we can make $M = N_1$ or just assume that M appears as an immediate subformula of an ancestor node of $\neg\mathcal{F}$ in order to guarantee analyticity. Similarly, for the tableaux, we can also assume that M appears rootwards of $\mathcal{F}$.

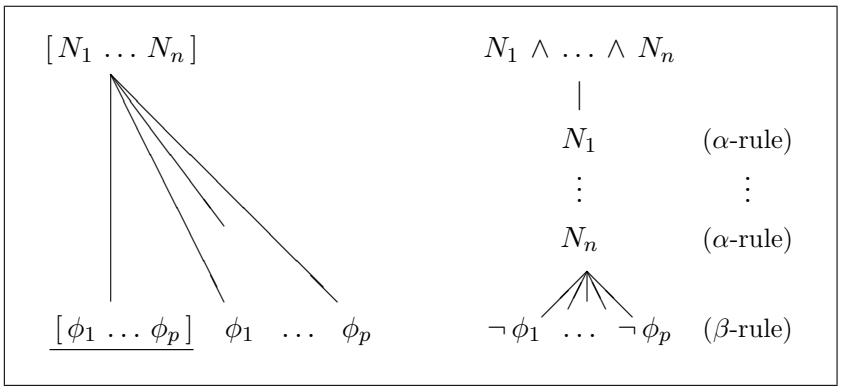


Figure 3.7: NAND vs. Tableaux.

If, for example, $N_2 = \phi_1$, than the branch marked with ϕ_1 on the NAND -tree would be closed because $\phi_1 = N_2 \prec [\,N_1 \ldots N_n\,]$, and they cannot all be false at the same time.

But, in that case, the branch ending at $\neg\,\phi_1$ on the tableau would also close because N_2 appears above $\neg\,\phi_1$, and $\neg\,\phi_1$ is incompatible with $N_2 = \phi_1$. Since the major component of the split in the NAND -tree is closed by construction, that branch can be discarded. The branches that are left match the branches in the tableau, provided each node is negated. As a consequence, a strongly analytic C^s-split with major component $\mathcal{S} = [N_1 \ldots N_n]$ is equivalent to an α and β-rule on $\mathcal{S}$ on an NNF analytic tableaux (assuming the α-rule was indeed required). If $\mathcal{S} = [N_1]$, then the split is only equivalent to an α-rule, because there is only one literal. For the NAND -tree, α-rules are usually implicit as each node $[\,N_1,\,\ldots,\,N_n\,]$ is treated as a list of formulas.

3.4.1 Branch Reductions

Simplifications can be added during the construction of a NAND tree. They have the effect of subsuming unnecessary extensions and produce earlier closure of the branches. This usually accelerates the proof procedure.

We have already seen that, along a branch, the complement of each node should be conjoined with the other nodes in that branch. Therefore, if along a branch we have the set of nodes $\{\,N_1, \ldots, N_n\}$, then we are assuming that, in that branch, $\mathcal{H}_1 = [[\,[\,N_1\,],\,\ldots,\,[\,N_n\,]\,]]$ holds (i.e., $\mathcal{H}_1 = \neg\,N_1 \wedge \ldots \wedge \neg\,N_n$). If we add another node P to that branch, then its complement $[\,P\,]$ can also be conjoined with $\mathcal{H}_1$ to give $\mathcal{H}_2 = [[\,[\,N_1\,],\,\ldots,\,[\,N_n\,]\,[\,P\,]\,]]$. Basically, at this point, $\mathcal{H}_2$ is our full assumption. It is perfectly possible to replace $\mathcal{H}_2$ with an equivalent assumption without altering the semantic content of the tree. Ideally, the new assumption should be syntactically simpler then $\mathcal{H}_2$. This can be achieved if we apply the simplification rules on $\mathcal{H}_2$.

Consider the complete C^s-tree for $\mathcal{F} = [\;\; q \;\; [\,p\,[\,r\,s\,]\,] \;\;\; [\,[\,p\,]\,[\,s\,]\,] \;\;]$ shown in Figure 3.8. The counter models of $\mathcal{F}$ are $M_1 = [[\,q\,[\,p\,]\,s\,]]$, $M_2 = [[\,q\,r\,s\,p\,]]$ and $M_3 = [[\,q\,r\,s\,s\,]]$. Each model corresponds to one open branch according to steps 1 to 3 of previous section.

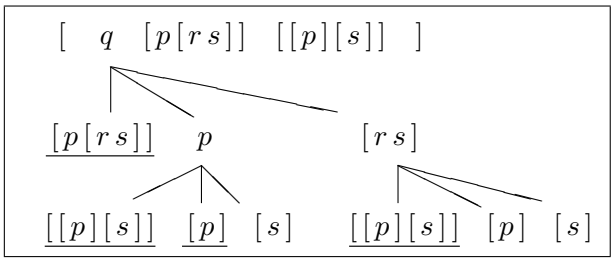


Figure 3.8: Complete open C^s-tree for the logical expression $q \Rightarrow ((p \wedge (\neg r \vee \neg s)) \vee (\neg p \wedge \neg s))$.

With relation to the three models we may observe the following: i) Model M_3 should have been reduced to $[\,q\,r\,s\,]$; ii) Model M_3 subsumes M_2. When we conjoin all the models together we obtain:

$$\begin{aligned} \text{DNF}\,(\neg\mathcal{F}) &= [\quad [\,q\,[p\,]\,s\,]\ [\,q\,r\,s\,p\,]\ [\,q\,r\,s\,s\,]\quad] && (3.4.5)\\ &= [\quad [\,q\,[p\,]\,s\,]\ [\,q\,r\,s\,p\,]\ [\,q\,r\,s\,]\quad] && \text{(Duplicate removal)}\\ &= [\quad [\,q\,[p\,]\,s\,]\ [\,q\,r\,s\,]\quad] && \text{(Subsumption rule)} \end{aligned}$$

Therefore, the rightmost split on $[\,[p\,]\,s\,]$ was redundant. Its only effect was to replace model $[\,q\,r\,s]$ that already existed up to node $[\,r\,s\,]$ with the two more specific models $[\,q\,r\,s\,s\,]$ or $[\,q\,r\,s\,p]$. Well, $[\,q\,r\,s\,]$ is the same as $[\,q\,r\,s\,s\,]$, and $[\,q\,r\,s\,]$ is more general than $[q\,r\,s\,p]$. Hence $[\,q\,r\,s\,]$ is more general then the other two together and subsumes them. Redundant (or subsumed) extensions are those for which one of the split components, $\mathcal{S}_i$, is already present in that branch. In those cases, the siblings of $\mathcal{S}_i$ will only produce subsumed models.

When simplifications are added at each split, by conjoining the complement of the new node with the set of previous assumptions and simplifying them, the alternative tree of Figure 3.9 follows.

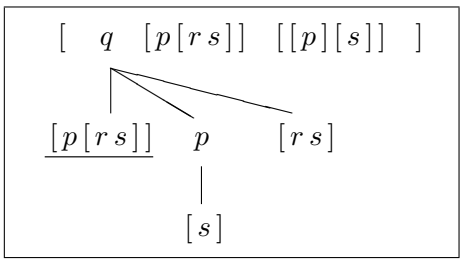


Figure 3.9: Complete open C^s-tree with branch reduction.

It is quite clear that this tree directly yields the same models as the simplified DNF $(\neg\mathcal{F})$ of equation (3.4.5). The tree was constructed as follows. When the split on $\mathcal{S}_1 = [\,p\,[\,r\,s\,]\,]$ is applied, $\mathcal{S}_1$ can be removed from the root node. In this case the reason is because $\mathcal{S}_1$ is closed and must be true, which means it can be replaced by $\top$ in the root node. This also means that $\mathcal{S}_1$ will not be used for another split. And, in general, that is why the same extension is never made twice on the same branch. The assumptions that are left are $\mathcal{H} = [[\ \ q\ \ [\,[p\,]\,[\,s\,]\,]\ \ ]]$. We now look at the middle branch. When p is added to the tree, we are making the assumption that p is false, i.e., $[\,p\,] = \top$. We add this assumption

to the existing set and obtain:

$$\begin{aligned}\mathcal{H}_1 &= [[\quad q \quad [\,p\,] \quad [\,[\,p\,][\,s\,]\,] \quad]] && (3.4.6)\\ &= [[\quad q \quad [\,p\,] \quad [\,[\,s\,]\,] \quad]] && (\top\text{-reduction on } [\,p\,])\\ &= [[\quad q \quad [\,p\,] \quad s \quad]] && (\text{Bracket simplification})\end{aligned}$$

The simplified $\mathcal{H}_1$ no longer contains $[\,[\,p\,][\,s\,]\,]$ but only the atom s survives. The act of removing $[\,p\,]$ form $[\,[\,p\,][\,s\,]\,]$ is a shortcut to closing the branch with split component $[\,p\,]$ that appears in Figure 3.8. The resulting s can be directly added to the model without extensions because it is already a literal. We see in Figure 3.9 that the node $[\,s\,]$ alone as been added below p to indicate that s it is part of the model.

In relation to the other sibling $[\,r\,s\,]$, the same method is used. except now we must add r and s to the set of assumptions:

$$\begin{aligned}\mathcal{H}_2 &= [[\quad q\ r\ s \quad [\,[\,p\,][\,s\,]\,] \quad]] && (3.4.7)\\ &= [[\quad q\ r\ s \quad [\,[\,p\,][\quad]\,] \quad]]\\ &= [[\quad q\ r\ s \quad [\,[\quad]\,] \quad]]\\ &= [[\quad q\ r\ s \quad]]\end{aligned}$$

We recognise in $\mathcal{H}_2$ the model M_3 after duplicate removal. We also see that the extension $[\,[\,p\,][\,s\,]\,]$ has been subsumed by the assumption s. Therefore, it is not necessary to perform that split and that branch ends at leaf $[\,r\,s\,]$.

When this type of branch reductions is be translated into tableaux rules we discover the Massacci's rules mentioned earlier in Chapter 2, Section 2.2.4, with relation to FNNF's.

$$\frac{\phi}{\dfrac{(\psi[\,\phi\,])}{\psi[\,true/\phi\,]}} \qquad \frac{\neg\, p}{\dfrac{(\psi[\,p\,])}{\psi[\,false/p\,]}} \tag{3.4.8}$$

where p is an atom, ψ and ϕ are now BNNF expressions and $\psi[\,true/\phi\,]$ represents ψ after the replacement of ϕ by $true$. Similarly for $\psi[\,false/p\,]$. We also recognise that they are the $\top - \bot$ reduction rules for NAND.

In NAND trees we apply them thoroughly to each assumption set, in a recursive way, and not only to recently added assumptions. We have seen in the section describing the simplification rules that each application of the $\top - \bot$ reductions may induce new formulas and atoms to drive new simplification opportunities. But, of course, it is each new assumption that triggers a new set of simplifications. Note as well that p or ϕ are not restricted to be at depth level 1 in a BNNF formula. The rules can be applied starting at any depth level.

When extensions are weakly analytic and restricted to splits whose major component is a literal of the form $[\,p\,]$, (i.e., atomic cut) this method resembles the FNNF tableaux. The main difference is that FNNF works with NF formulas while NAND interprets BNNF's. And, as mentioned in the context of FNNF's, the atomic cut plus simplifications also resembles the splitting rule of DPLL.

3.4.2 Soundness and Completeness

Soundness and completeness in a formal logic system is concerned in guaranteeing: i) that every syntactic derivation $X \vdash Y$ is semantically valid $X \models Y$; ii) that if Y is a semantic consequence of X, $X \models Y$, than a syntactic derivation $X \vdash Y$ can be found in that system. In the NAND tree expansion method soundness and completeness are ensured by the following two theorems.

Theorem 3.4.12 (Soundness) *Let N be a wff. If a closed tree for N exists, then N is semantically valid:*

$$C^s(N) \;\vdash\; \bot \qquad \Rightarrow \qquad \models N$$

The next theorem states the converse.

Theorem 3.4.13 (Completeness) *Let N be a wff. If N is semantically valid then every complete tree for N is closed:*

$$\models N \qquad \Rightarrow \qquad CC^s(N) \;\vdash\; \bot$$

There are two independent approaches for working out the proof of these theorems. The first one is the (more abstract) formal proof by mathematical induction on the tree depth. The second is based on showing that, by construction, the model generated by a complete C^s-tree for N yields an equivalent DNF representation for $\neg N$. Thus, since the model obtained from a closed tree is the empty model $\bot$, we have that $\neg N \Leftrightarrow \bot$ or, equivalently, $N \Leftrightarrow \top$.

The induction proofs presented here are adapted from of the tableau method proofs described by Smullyan in [25] and by Sandqvist in [23].

Proof of Theorem 3.4.12 Showing that

$$C^s(N) \;\vdash\; \bot \qquad \Rightarrow \qquad \models N$$

is the same as showing

$$\not\models N \qquad \Rightarrow \qquad C^s(N) \;\nvdash\; \bot$$

ie, that if N is not valid than any C^s-tree for N will be open.
By construction, a C^s-tree for N starts at the root node marked with N and is grown downwards by adding tautologies at each split, through the *law of excluded middle*:

$$\frac{\top}{[\ [\ N_1\ \ldots\ N_n\ ]\ \ N_1\ \ldots\ N_n\ ]} \tag{3.4.9}$$

meaning that at least one of the $n+1$ nodes — $[\ N_1\ \ldots\ N_n\ ]$, $N_1, \ldots, N_n$ — generating $n+1$ new branches must be false.

So, including the root node, each node added to the tree is independently assumed false. The tree only closes if, along each branch, two nodes (not necessarily distinct) are $\perp$-inconsistent. Intuitively, closure can only occur if the inconsistency is caused at the root node, since each extension is tautologous as a whole and falsifying the root was the sole real assumption made.

Suppose then that there is an interpretation I under which every node along a given (open) branch β of a C^s-tree for N is falsifiable. We say in this case that the C^s-tree is falsifiable. We want to show that by extending this tree at least one branch will still remain open. The only way β can be closed is by adding new nodes along to make it $\perp$-inconsistent. But nodes can only be added through the split rule and the split rule is itself a tautology, so that at each split at least one of the descendent nodes must be false. Let's name it Γ. As a consequence, the branch β' that is obtained from β by adding Γ to it is still open.

What we have shown so far is that an immediate extension of a falsifiable C^s-tree under a given interpretation I is still falsifiable under I. This means that if the origin of a C^s-tree is false under the interpretation I, then that C^s-tree must also be falsifiable under I.

We know that a closed C^s-tree cannot be falsifiable under any interpretation by definition of branch closure. Therefore, if a tree is closed at level K, the level $K-1$ cannot be falsifiable either. Because if it were, by what we just proved, at least one branch at level K would be open. Therefore, progressing inductively until level $k=0$ we conclude that a closed tree must have a root that is not falsifiable under any interpretation, which means the root node must be true. This concludes the proof. □

Proof of Theorem 3.4.12 We wish to show

$$\vDash N \qquad \Rightarrow \qquad CC^s(N) \ \vdash\ \perp$$

which is equivalent to showing its contraposition

$$CC^s(N) \nvdash \perp \quad \Rightarrow \quad \not\models N$$

Suppose we have found a complete C^s-tree for N, which we name CC_n, that is not closed. Then there must be a (saturated) branch $\beta \in CC_n$ such that along β every node has been successfully assigned the value false. Consequently, no two nodes Γ_i and Γ_j belonging to β can be linked by any one of the relations $\Gamma_i \prec \Gamma_j$ or $\Gamma_j \prec \Gamma_i$.

The next step of the proof is to show that, for any saturated open branch, the node at the origin — the root node — is falsifiable. In fact, we shall prove something stronger: that each node $\Gamma_i \in \beta$ is falsifiable under some interpretation I.

Assign each atom l that is a node $\in \beta$ the value $\perp$. For all other atoms that are not nodes of β, but are subformulas of some nodes of β, assign $\top$. Since no positive and negative occurrences of the same literal can simultaneously appear as nodes of β, otherwise $l \prec [\, l \,]$ would hold, the atomic assignments are consistent with each other.

Suppose now that we have a compound formula at a node $\Gamma_m = [\, \Upsilon_1 \cdots \Upsilon_p \,] \in \beta$. In order for Γ_m to be false, each $\Upsilon_i \prec \Gamma_m$, $i = 1, \ldots, p$, must be true. Let us choose an arbitrary Υ_i and then do the same for all of the rest. If Υ_i is atomic, then it has already been assigned the value $\top$, and we are done. Otherwise, we may assume the non-atomic $\Upsilon_i = [\, \gamma_1 \cdots \gamma_r \,]$.

Because β is saturated, then, for each Υ_i, exactly one of Υ_i or of any of its immediate subformulas $\gamma_k \prec \Upsilon_i$ must be in β. It cannot be Υ_i since the relation $\Upsilon_i \prec \Gamma_m$ between the two nodes has, by hypothesis, been excluded. Therefore, we must have a node $\gamma_k \in \beta$ for exactly one $k \in \{1, \ldots, r\}$.

Now, for Υ_i to be true, it is enough that γ_k is made false. If γ_k is an atom, then it has already been assigned the value $\perp$ at the start and indeed one gets $\Upsilon_i = \top$. On the other hand, if γ_k is another compound formula, then the same reasoning that has been used for Γ_m can be repeated for γ_k, and recursively thereafter.

For each individual node Γ_m this procedure is bound to terminate if the degree of Γ_m is finite, which we assume to be the case. Note that

$$\deg(\Gamma_m) = 1 + \sum_{i=1}^{p} \deg(\Upsilon_i)$$

As we have seen, each Υ_i introduces into the branch at most one new node γ_k of degree $\deg(\gamma_k) < \deg(\Upsilon_i)$. Unless $\deg(\Upsilon_i) = 0$, in which case no node is added.

Therefore, our new set of waiting tasks has overall degree

$$D \quad < \quad \sum_{i=1}^{s} \deg(\Upsilon_i), \qquad \text{for} \quad s \leq p, \quad \deg(\Upsilon_i) > 0$$

where s is the number of Υ_i's for which $\deg(\Upsilon_i) > 0$ holds. Remark that we can similarly write

$$\deg(\Gamma_m) = 1 + \sum_{i=1}^{s} \deg(\Upsilon_i) \qquad \text{for} \quad s \leq p, \quad \deg(\Upsilon_i) > 0$$

from which the following inequality arises:

$$D \quad < \quad \deg(\Gamma_m) - 1$$

Given that at each step we are replacing a task Γ_m by a strictly smaller set of tasks D, we will eventually reach the convergence point $D = 0$.

We therefore have shown that each node along a saturated open branch β is falsifiable under the interpretation I that assigns each atom along β the value $\bot$ and all other atoms the value $\top$. Thus, the node N at the root must also be falsifiable under I.

If the initial tree was not complete, ie if some branches were not saturated, all we need to do is extended until every branch becomes saturated. If the tree remains open, the previous proof applies (we are not interested in the case in which the tree closes). □

3.5 Application to Propositional Logic

The formalism and the techniques described in the previous section can be globally applied to propositional theories without restrictions. They can be used as refutation procedures or for knowledge compilation. Some additional strategies like the usage of *lemmas* can be added to accelerate or optimise the process. This is described in the Section 3.5.1 where DNNF compilation is performed using NAND trees.

3.5.1 Compiling NAND into DNNF

Like other proof systems, NAND can also be used as a compilation tool. Either by combined application of the simplification rules — Shanon expansion, distributivity, $\top - \bot$ reduction and subsumption — or through tree expansion. We shall concentrate here on tree expansion. Due to its structure, a DNNF model for any theory is always guaranteed, even without adopting any strategic refinements.

In Section 3.4 we have seen that BNNF is isomorphic to NNF and is able to simulate a tableau style proof through tree expansion. Regularity is ensured by the set of simplification rules that can be combined with the extension rule to remove subsumed nodes or branches. As with regular tableaux, the model constructed form the open branches is in DNF and is decomposable — thanks to regularity, no atom appears twice in the same open branch. An important detail to remember is that a complete analytic C^s-tree for $\mathcal{T}$ returns a DNNF model not for $\mathcal{T}$, but for $\neg\mathcal{T}$, ie a counter model. Therefore, if we want a model for $\mathcal{T}$, we must derive a C^s-tree of $\neg\mathcal{T}$.

Although the normal form representation of any model $\mathcal{T}$ obtained by tree expansion is only DNF, better space efficiency may be achieved by regrouping some of the common terms through application of the distributivity rule. Additional strategies may also be implemented, like avoiding to either split or even extend *disconnected* DNNF subformulas — a subformula is *connected* if it shares atoms with other subformulas, *disconnected* otherwise. First of all, these are formulas that make no contribution towards branch closure. Being DNNF, they are individually satisfiable; and being disconnected and individually satisfiable makes them globally satisfiable. Secondly, they would end up being regrouped by distributivity, for they would never be subsumed and would appear in every open branch. Although in this way the resulting tree may be *incomplete*, the unfulfilled formulas still participate in the final DNF model because they are at least attached to the root node, hence to every branch.

Another interesting idea is to include a generalisation of *lemma* extensions borrowed from *model elimination* (ME) tableaux. The typical usage of lemmas works as follows. When a branch rooted at node N has been closed, then each of its unprocessed *siblings*[1], S_i, can be extended by $\neg N$. Or, equivalently, $\neg N$ can be added as an ancestor node of S_i. As a consequence, if a node marked with N occurs somewhere below S_i, it may be automatically closed by conflict with $\neg N$ without the need to reprocess it. This shortcut is self-consistent because the nodes S_i share the same ancestors as N, and all the information that was used for refuting N the first time is still available. Although in ME-tableau the information is only transmitted if N has been previously closed, we show here that it still works for open branches. Under this generalisation, inclusion of lemmas at the atomic level partially simulates the atomic cut rule.

The following example illustrates a C^s-tree for a DNNF compilation using some of the strategies described above.

[1] A branch rooted at S is a *sibling* of N if S and N have the same parent.

Example 3.5.1 *Consider once more the theory*

$$\mathcal{T} \;=\; \{\;\; (F \vee G),\; (\neg F \vee H),\; (G \vee H) \;\;\} \tag{3.5.1}$$

and its NAND representation:

$$\neg\mathcal{T} \;=\; [\quad [[F]\,[G]],\quad [F\,[H]],\quad [[G]\,[H]]\quad] \tag{3.5.2}$$

Distinct C^s-trees for $\neg\mathcal{T}$ are possible upon variation of sentence orderings and choice of simplification strategies.

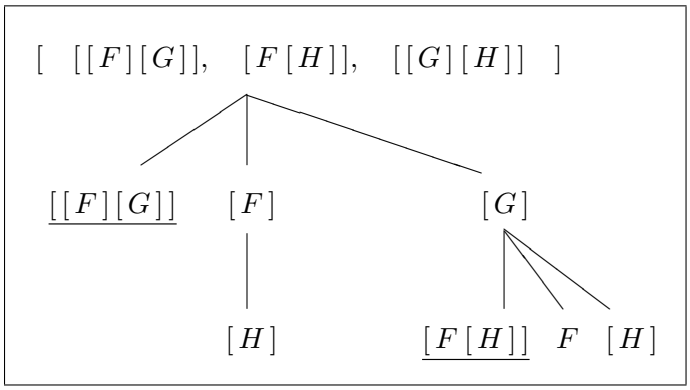


Figure 3.10: Analytic C^s-compilation without lemmas.

In Figure 3.10 a complete analytic C^s-tree of $\neg\mathcal{T}$ is presented. The formulas were selected in the order they are stored, from left to right. Three counter models for $\neg\mathcal{T}$ are found, one for each open branch, leading to the following DNF representation of $\mathcal{T}$.

$$\begin{aligned}\mathrm{DNF}(\mathcal{T}) \;&=\; [\quad [F\,H],\quad [G\,[F]],\quad [G\,H]\quad] \\ &=\; (F \wedge H) \;\vee\; (G \wedge \neg F) \;\vee\; (G \wedge H)\end{aligned} \tag{3.5.3}$$

Since the node $[G]$ is shared by the two right most open branches, distributivity can be applied

$$[[\quad [G\,[F]]\;[G\,H]\quad]] \;=\; [\,G\,[F\,[H]]\,] \tag{3.5.4}$$

to obtain a more compact DNNF

$$\mathrm{DNNF}(\mathcal{T}) \;=\; [\quad [F\,H],\quad [G\,[F\,[H]]]\quad] \tag{3.5.5}$$

Simplification rules have been applied to the selective sets at each node, automatically subsuming the immediate subformula $[[G]\,[H]]$ from the root. Note how this formula

never reached to be selected. However, under this strategy, the model $[\,G\,H\,]$*, matching the right most branch, which should have also been subsumed has failed to be identified as such.*

A possible solution is to add lemmas, as shown in Figure 3.11. As soon as node $[\,F\,]$

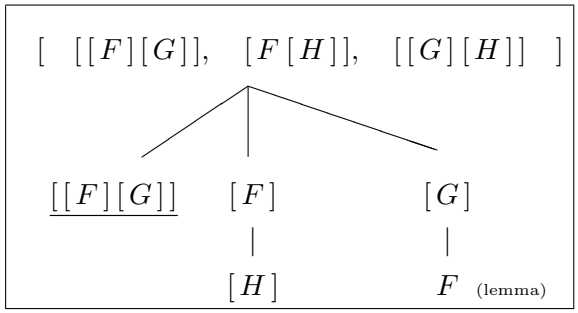


Figure 3.11: Analytic C^s-compilation using lemmas, equivalent to a split on $[\,F\,[\,F\,]\,]$.

is extended, its sibling $[\,G\,]$ *is also extended with the complement of* $[\,F\,]$*, which is* F*. The presence of* F *in the branch immediately subsumes* $[\,F\,[H]]$*. The final compiled model is thus*

$$\mathrm{DNNF}\,(\mathcal{T}) \;=\; [\;[\,F\,H\,]\;[\,G\,[\,F\,]\,]\;] \tag{3.5.6}$$

which can be recognised as be the same optimal model displayed in (2.2.27) that was found by using decomposition trees.

Inclusion of lemmas is only a partial approximation of the atomic cut. It is systematically done from left to right, across siblings, which makes it very sensitive to the order of literals inside the extended formulas, assuming the order is kept. But this is required to preserve equivalence. Note how G, the complement of $[\,G\,]$, has not been added to the branch rooted at $[\,F\,]$.

The tree in Figure 3.11 is indeed equivalent to the tree that would be obtained by starting with an atomic split on F and $[\,F\,]$. However, if the subformula $[\,[\,F\,]\,[\,G\,]\,]$ was reordered as $[\,[\,G\,]\,[\,F\,]\,]$, then it would instead be equivalent to a split on G and $[\,G\,]$, producing the different result seen in Figure 3.12.

Under an actual split on $[\;G\,[\,G\,]\,]$, the branch marked with $[\,G\,]$ would simplify to

$$\begin{array}{c} [\quad G,\quad [\,[\,F\,]\,[\,G\,]\,],\quad [\,F\,[\,H\,]\,],\quad [\,[\,G\,]\,[\,H\,]\,]\quad] \\ \Updownarrow \\ [\;G\,[\,F\,[\,H\,]\,]\,] \end{array} \tag{3.5.7}$$

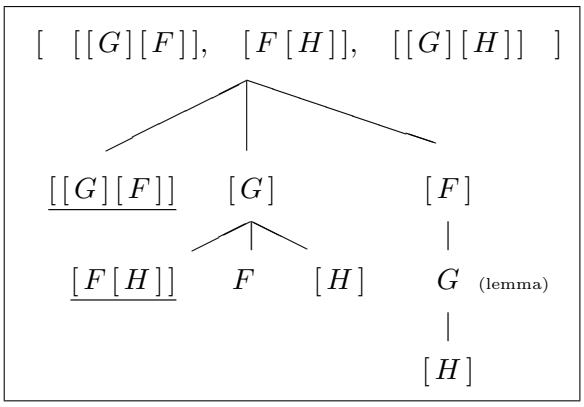


Figure 3.12: Analytic C^s-compilation using lemmas, equivalent to a split on $[\,G[G]\,]$.

by the $\top - \bot$ reduction rules. Similarly, the branch marked with G would reduce to:

$$\begin{array}{c} [\quad [G],\quad [[F][G]],\quad [F[H]],\quad [[G][H]]\quad] \\ \Updownarrow \\ [\quad [G],\ F,\ [F[H]],\ H\quad] \\ \Updownarrow \\ [\,[G]\,F\,H\,] \end{array} \tag{3.5.8}$$

It is quite clear that, after regrouping common terms, the reduced models (3.5.7) and (3.5.8) match the open branches of the tree in Figure 3.12[1].

Upon comparison with Figure 3.10, we realise that the lemma has added the spurious term $[\,G\,]$ to the model: instead of $[\,F\,H\,]$, like in (3.5.5), we now have $[\,[\,G\,]\,F\,H\,]$, as seen in (3.5.8) and in the right most branch of Figure 3.12. The reason is that lemma G does not bring any useful information that can be used to subsume nodes or branches below node $[\,F\,]$, $[\,F\,]$ alone suffices. Based on this observation, we should stick to the following general principle. In C^s-compilations, lemmas should not be added to branches where their presence is transparent; in C^s-proofs, however, where the size of the final model is not important, they are usually helpful, just like atomic splits.

Sticking to this principle, the tree from Figure 3.12 may be replaced by the alternative tree illustrated in Figure 3.13, where lemma G has been removed. In addition, lemma $[\,F\,]$ has been added to the mid branch by virtue of its sibling node F, to its left. Having done that, the mid branch composed of the node set $\{\,[\,G\,],\,[\,H\,],\,[\,F\,]\,\}$ is subsumed by the

[1]This is true for this particular theory, but, in general, lemmas are more restrictive than atomic cuts

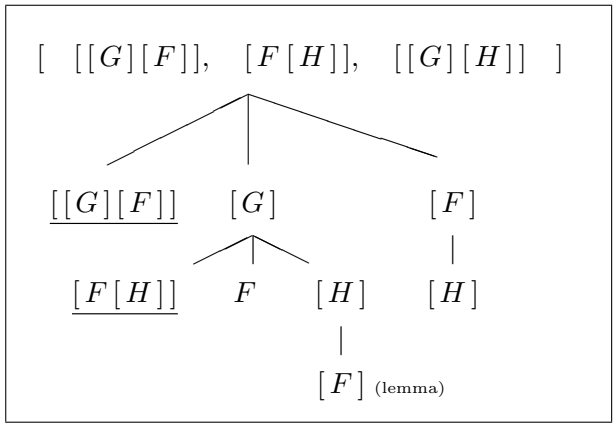


Figure 3.13: Analytic C^s-compilation by adequate usage of lemmas.

right most branch:

$$\begin{aligned} \mathrm{DNNF}(\mathcal{T}) &= [\ \ [\,G\,[\,F\,[\,F\,H\,]\,]\,]\,,\ \ [\,F\,H\,]\ \ ] \\ &= [\ \ [\,G\,[\,F\,]\,]\,,\ \ [\,F\,H\,]\ \ ] \end{aligned} \tag{3.5.9}$$

obtained through application of the $\top$-reduction rule on $[\,F, H\,]$.

The relevance of DNNF normal form has been discussed Section 2.2.2. It basically resides in the fact that decomposability ensures independence between subformulas, which allows tests like satisfiability to be decided in linear time relative to the size of the formula. At the same time, any DNNF is also satisfiable on its own by virtue of the same property.

It is actually interesting to analyse this under a different perspective. By compiling a theory $\mathcal{T}$ into DNNF, we are removing from it all the paths that would systematically lead to a dead end in satisfiability tests. Rephrasing, if $\mathcal{T}$ is not DNNF, then a C^s-proof for $\mathcal{T} \models \mathcal{S}$, for some query $\mathcal{S}$, may contain branches that will be closed or subsumed not due to the conjunction $\mathcal{T} \wedge \neg\,\mathcal{S}$, but to $\mathcal{T}$ alone. We can see how this is a waste of time. In a series of queries $\mathcal{S}_i$, the same branches would keep being extended and inevitably discarded each time, independently of $\mathcal{S}_i$. The DNNF compilation cleans the tree from superfluous computations. In fact, each time a C^s-tree for $[\,\mathcal{T}\,[\,\mathcal{S}\,]\,]$ (ie, $\mathcal{T} \Rightarrow \mathcal{S}$) is performed, $\mathcal{T}$ is recompiled.

Let us consider at the theory $\mathcal{T} = \neg\,(A \Leftrightarrow B)$ with the BNNF/NNF representation given below:

$$\begin{aligned} \mathcal{T} &= [[\ \ [\,[\,A\,]\,[\,B\,]\,]\ \ \ [\,A\,B\,]\ \ ]] \\ &= (A \vee B) \wedge (\neg\,A \vee \neg\,B) \end{aligned} \tag{3.5.10}$$

and its equivalent DNNF compilation

$$\begin{aligned} \mathrm{DNNF}(\mathcal{T}) &= [\ \ [[A]\,B]\ \ \ [A\,[B]]\ \ ] \\ &= (\neg A \wedge B) \vee (A \wedge \neg B) \end{aligned} \tag{3.5.11}$$

We have chosen this example because both theories have the same size, not to influence the result that follows. The variables A and B can be any complicated compound formulas. A C^s-proof for $[\,\mathcal{T}\,[\mathcal{S}\,]]$, where the original (non DNNF) representation of $\mathcal{T}$ is used, is partially derived in Figure 3.14.

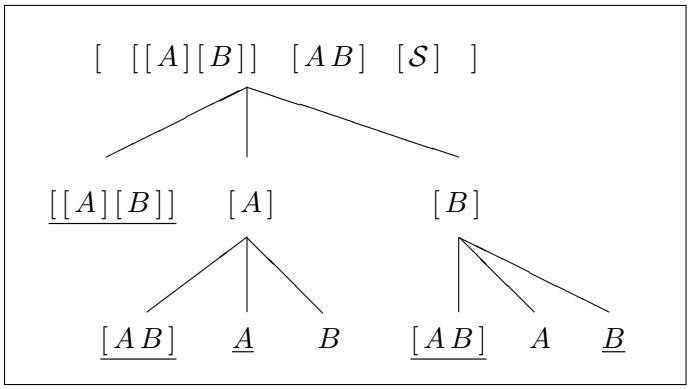


Figure 3.14: Incomplete C^s-tree of $[\,\mathcal{T}\,[\mathcal{S}\,]\,]$.

The external double brackets surrounding $\mathcal{T}$ in the root node disappear upon merging with $[\mathcal{S}\,]$. Furthermore, no simplifications have been applied so that closed or subsumed branches get explicitly represented in the tree. And since we are not interested in the actual query $\mathcal{S}$, the tree is left incomplete and only terms related to $\mathcal{T}$ have been extended. As it stands, ignoring the unfulfilled formula $[\mathcal{S}\,]$ at the root, the tree represents as well a complete C^s-tree for $[\mathcal{T}\,]$, or a compilation for $\mathcal{T}$, and the open branches reproduce the DNNF model in (3.5.11).

Next, in Figure 3.15, the alternative C^s-tree for $[\,\mathrm{DNNF}(\mathcal{T})\ \ [\mathcal{S}\,]\,]$ is illustrated.

It is evident how the DNNF theory encloses two independent sub-theories that can be directly combined with $[\mathcal{S}\,]$ without much effort. On the contrary, the original theory $\mathcal{T}$ hides two closed branches that do not contribute to the satisfiability test and requires extension steps up to level 2. This appears to be partly explained by the fact that $\mathcal{T}$'s main connection is a conjunction, while in $\mathrm{DNNF}(\mathcal{T})$ we find a disjunction. But one should not be misled because the problem with $\mathcal{T}$ is the lack of independence between its clauses. If they were decomposable, they could be individually tested against $\mathcal{S}$, no matter what the main connection was. The fact that it is a conjunction only means that

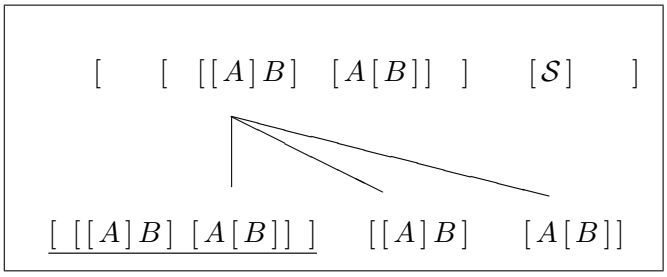


Figure 3.15: Incomplete C^s-tree for [DNNF($\mathcal{T}$) [$\mathcal{S}$]].

Table 3.3: Runtime statistics (milliseconds)

TPTP Problem	Theorem	NAND	LeanTap
NewGRA001-1	yes	10	10
NewLCL181-2	yes	0	0
NewLCL230-2	yes	0	0
NewMSC007-1.008	no	-	-
NewNUM285-1	no	140	34720
NewPUZ004-1	yes	0	10
NewPUZ009-1	yes	10	30
NewPUZ013-1	yes	0	8140
NewPUZ014-1	yes	20	8340
NewPUZ015-2.006	yes	27570	-
NewPUZ016-2.005	yes	1820	-
NewPUZ030-2	yes	580	-
NewPUZ033-1	yes	10	300

entailment holds if and only if the test is satisfiable for each immediate subformula of [$\mathcal{T}$], while with a disjunction this is a sufficient condition, and it is only necessary that one immediate subformula of $\mathcal{T}$ entails $\mathcal{S}$.

3.5.2 Testing

The propositional NAND system has been tested against an analytic tableau method. Since NAND is implemented in Prolog, we have chosen to compare it against another Prolog program. For that purpose we selected LeanTap [3, 27], a well known Prolog algorithm that implements NNF analytic tableaux as a refutation procedure.

Table 3.4: Runtime statistics (continued)

TPTP Problem	Theorem	NAND	LeanTap
NewSYN001-1.005	yes	240	-
NewSYN003-1.006	yes	10	-
NewSYN008-1	yes	0	0
NewSYN011-1	yes	0	0
NewSYN028-1	yes	0	0
NewSYN029-1	yes	0	0
NewSYN030-1	yes	0	0
NewSYN032-1	yes	10	10
NewSYN040-1	yes	0	0
NewSYN041-1	yes	0	0
NewSYN044-1	yes	0	0
NewSYN045-1	yes	0	0
NewSYN046-1	yes	0	0
NewSYN047-1	yes	0	0
NewSYN085-1.010	yes	0	10
NewSYN086-1.003	no	10	0
NewSYN087-1.003	no	30	0
NewSYN089-1.002	yes	0	10
NewSYN090-1.008	yes	190	-
NewSYN091-1.003	no	60	10
NewSYN092-1.003	no	120	10
NewSYN093-1.002	yes	20	620
NewSYN094-1.005	yes	-	-
NewSYN097-1.002	yes	50	590
NewSYN098-1.002	yes	1120	-
NewSYN302-1.003	no	170	10

What we expect is that, thanks to simplifications and branch reduction, NAND will perform better than LeanTap. The only simplifications that are being used for NAND here are the $\top - \bot$ reduction rules that act upon literals. These are applied to the initial theorem and then during the tree expansion, whenever new nodes are added. Neither the compound simplification nor the subsumption rule is being used.

Both procedures were set to stop after the first counter model was found. A single counter model is enough to invalidate a theory. That means that as soon as the first open branch is found, the search stops. However, if the theorem is valid, all branches must be searched in order to be closed.

In order to perform the tests a set of propositional theorems was selected from the TPTP Problem Library[1] [28]. The results are shown in Tables 3.3 and 3.4. The first column contains the TPTP theorem identification. The second column indicates whether the theorem is valid ("yes"), or if is has counter models ("no"). The last two columns refer to the runtime statistics for each program (NAND or LeanTap). Times were obtained with Prolog predicate statistics/2 under the key 'runtime', with units in milliseconds. Timeouts are marked with '-', and correspond to times over 100 seconds.

As suspected, NAND performs better or equally better in most of the problems. Nevertheless, we do find a few cases were LeanTap wins. These seem to be linked to invalid theorems, which have open trees. In principle these trees terminate faster than the closed ones because we are only looking for one model. Therefore, it is likely that the effect of simplifications is less significant and does not compensate for the overhead it causes.

3.6 Application to First-Order Logic

First order logic, also known as first order predicate logic, generalises propositional logic through the introduction of the *universal* and *existencial* quantifiers

$$\forall_x \quad \equiv \quad \text{for all x} \tag{3.6.12}$$

$$\exists_x \quad \equiv \quad \text{there exists an x} \tag{3.6.13}$$

and of sets *function* symbols and *predicates* of variable arity[2] $P(x_1, \ldots, x_n)$. Predicates are the atomic sentences used to reason about individuals, which are the variables. The predicate symbol may represent an attribute associated with a variable or a relation between its variable arguments. If a predicate has arity 0, it becomes equivalent to a propositional sentence.

[1] I would like to thank Driss for lending me his TPTP Prolog converter code, which was quite helpful.

[2] Arity is defined as the number of arguments in a predicate or function. $P(x_1, \ldots, x_n)$ has arity n.

Quantifiers act upon variables. An infinite variable set $\mathcal{U} = \{x_1, x_2 \ldots\}$ is usually considered, such that

$$\forall_x P(x) \quad \equiv \quad P(x_1) \wedge P(x_2) \wedge \ldots \tag{3.6.14}$$

$$\exists_x P(x) \quad \equiv \quad P(x_1) \vee P(x_2) \vee \ldots \tag{3.6.15}$$

The NAND system can be extended to predicate logic by replacing the propositional sentences by predicates and translating the quantifiers into BNNF in the way suggested by Sandqvist [23]:

$$\forall x\ P(x) \quad \rightleftharpoons \quad [[\, x,\, P(x)\,]] \tag{3.6.16}$$

$$\exists x\ P(x) \quad \rightleftharpoons \quad [\, x\,[\, P(x)\,]\,] \tag{3.6.17}$$

This representation is based on the interpretation given in (3.6.14) and (3.6.15). The variable x appearing at the beginning of the BNNF formulas represents the quantification. If we explicitly enumerate the variables in $\mathcal{U}$ and drop the quantifiers we obtain the equivalent BNNF for (3.6.14) and (3.6.15):

$$\forall x\ P(x) \quad \rightleftharpoons \quad [[\, p(x_1),\, p(x_2) \ldots\,]] \tag{3.6.18}$$

$$\exists x\ P(x) \quad \rightleftharpoons \quad [\,[\, p(x_1)\,]\,[\, p(x_2)\,]\ \ldots\,] \tag{3.6.19}$$

Note that the BNNF quantifiers still obey the De Morgan laws

$$\neg\,(\,\forall_x P(x)\,) \quad \simeq \quad \exists_x \neg P(x) \tag{3.6.20}$$

$$\neg\,(\,\exists_x P(x)\,) \quad \simeq \quad \forall_x \neg P(x) \tag{3.6.21}$$

by virtue of bracket simplification or introduction

$$[\ [[\, x,\, P(x)\,]]\ ] \quad \simeq \quad [\ \ x\ \ [\,[\, P(x)\,]\,]\ \ ] \tag{3.6.22}$$

$$[\ \ [\, x\ \ [\, P(x)\,]\,]\ \ ] \quad \simeq \quad [[\ \ x\,[\, P(x)\,]\ \ ]] \tag{3.6.23}$$

The example below shows some first order logical formulas and their equivalent representation in BNNF.

Example 3.6.1

$$(\,\forall_x P(x)\,) \wedge (\,\exists_y Q(y)\,) \quad \rightleftharpoons \quad [[\ \ x,\ P(x)\ \ [\, y\,[\, Q(y)\,]\,]\ \ ]] \tag{3.6.24}$$

$$\exists_x\,(\,P(x) \Rightarrow \forall_y P(y)\,) \quad \rightleftharpoons \quad [\ \ x\ \ [[\, P(x)\ \ [\, y\ P(y)\,]\ ]]\ \ ] \tag{3.6.25}$$

$$\forall_{x,y}\,(\,R(x) \Rightarrow \exists_z S(y,z)\,) \quad \rightleftharpoons \quad [[\ \ x,\ y,\ [\ R(x)\,[[\, z\,[\, S(y,z)\,]\,]]\ ]\ \ ]] \tag{3.6.26}$$

$$\forall_x \exists_y P(x,y) \quad \rightleftharpoons \quad [[\ \ x\,[\, y\,[\, P(x,y)\,]\ ]\ \ ]] \tag{3.6.27}$$

Inner double brackets can be removed and allow subformulas to be merged.

With the quantifiers inside the formulas simplification is not at all straightforward. To deal with these expressions a type of skolemisation must be applied. A "just in time" skolemisation can be used during the construction of a strongly analytic C^s tree, in the style of free variable tableaux [3, 25]. The BNNF first order rules are the following.

$$\begin{array}{ccc} \dfrac{[[\ x,\ F(x)\ ]]}{F(x_1)} & \dfrac{[\ x,\ F(x)\ ]}{[\ F(b)\ ]} & \dfrac{[\ x,\ F(x,y,\ldots)\ ]}{[\ F(f_1(y_1,\ldots),y_1,\ldots)\ ]} \\ (\forall\text{−E}) & (\exists\text{−E}) & (\text{Skolem Function}) \end{array} \tag{3.6.28}$$

Formulas on the top correspond to quantified immediate subformulas that can be selected from the set of ancestor nodes along a given branch. The subformulas on the bottom represent the major component of the split generated by selecting the formulas on the top. Minor components are not represented in this rules for the sake of clarity. It is implicit that they are to be extracted in the way dictated by the ground E^s extension rule that was defined in Chapter 3, Section 3.4. The F's represent any BNNF formulas. The first rule (∀−E) corresponds to the universal elimination inference rule, where the universal variable x is replaced by a fresh new variable x_1 and the quantifier is removed. Rule (∃−E) is the existential elimination rule that projects the existential quantified variable x into a constant b. b is also called a Skolem *constant*. The last rule on the right is necessary to introduce the Skolem *functions*. It applies when y is a variable that is universally quantified in an ancestor node of $[\ x,\ F(x,y,\ldots)\ ]$. The dots indicate that y may not be the only universally quantified variable in that situation, and that any other variables should also be included in the Skolem function. In practice, the Skolem function is only really used if there are predicates within with F with arity>1 that relate y and x. Each Skolem constant or function must be unique.

Furthermore, as in a normal skolemisation process, there should not be two quantifiers acting on a variable with the same name, for this generates confusion. Therefore all duplicate quantified variables should be renamed along with the predicates within their scope:

$$\begin{array}{ccccc} (\forall_x P(x)) \wedge (\exists_x Q(x)) & \rightleftharpoons & [[\ \ x,\ P(x)\ \ [x\,[Q(x)]]\ \ ]] \\ \downarrow & & \downarrow \\ (\forall_x P(x)) \wedge (\exists_y Q(y)) & \rightleftharpoons & [[\ \ x,\ P(x)\ \ [y\,[Q(y)]]\ \ ]] \end{array} \tag{3.6.29}$$

The next example shows a C^s-proof for equation (3.6.25) taken from Example 3.6.1 that applies the first order logic (FOL) BNNF rules that we have just defined.

Example 3.6.2 *The formula* $\mathcal{F} \equiv (\exists_x (P(x) \Rightarrow \forall_y P(y)))$ *is a tautology. If there is an* x *such that* $P(x)$ *is false, then we choose it and the formula holds. If there is no* x *that makes* $P(x)$ *false, then it means that* $\forall_y P(y)$ *is true, and the formula is again satisfied. Now that we are convinced of that, we shall verify that the* C^s*-tree for* $\mathrm{BNNF}(\mathcal{F})$ *is closed using the first order logic (FOL) BNNF rules in (3.6.28). This is depicted in Figure 3.16, after removal of double brackets from* $\mathrm{BNNF}(\mathcal{F})$.

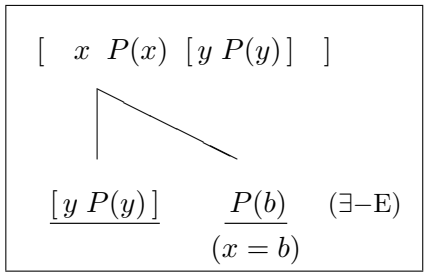


Figure 3.16: An FOL C^s-proof.

The tree is interpreted as follows. The existentially quantified formula $[y\ P(y)]$ *is selected. Application of the* $(\exists - E)$ *rule transforms it into* $[P(b)]$. *We simply keep the original formula in the major component to know for ourselves what formula has been selected. But the minor component reveals that the* $(\exists - E)$ *rule has indeed been applied. Then, the branch with node* $P(b)$ *is closed by unifying* $P(b)$ *with the universally quantified immediate subformula* $P(x)$ *at the root.*

Unification with or selection of universally quantified formulas in ancestor nodes is always made by taking a copy of the formula before the unification. This guarantees that the original formula does not get bound. This is necessary because universally quantified formulas can be selected any number of times, since the supply of variables is unlimited. Therefore, an unbound copy must always exist.

Suppose now that there is a tree that does not close. In propositional logic the formulas are only selected once along each branch, which means that at some point the tree is complete and we know that the theorem is not valid. In FOL, however, the universal formulas can be continually selected. This may pose the problem of an infinite tree, which is usually associated with redundant extensions taking place. Therefore, to prevent the tree from growing forever, one must check for subsumed nodes along a branch. If a node that already exists in a branch is to be repeated by a new extension, then the extension should not be performed.

Example 3.6.3 *Consider the formula* $\mathcal{F} = (\,(\,\forall_x P(x)\,) \wedge (\,\exists_y Q(y)\,)\,)$ *with* $\mathrm{BNNF}(\mathcal{F}) = [[\quad x,\ P(x)\ \ [\,y\,[\,Q(y)\,]\,]\quad]]$. *Its open* C^s*-tree is presented in Figure 3.17*

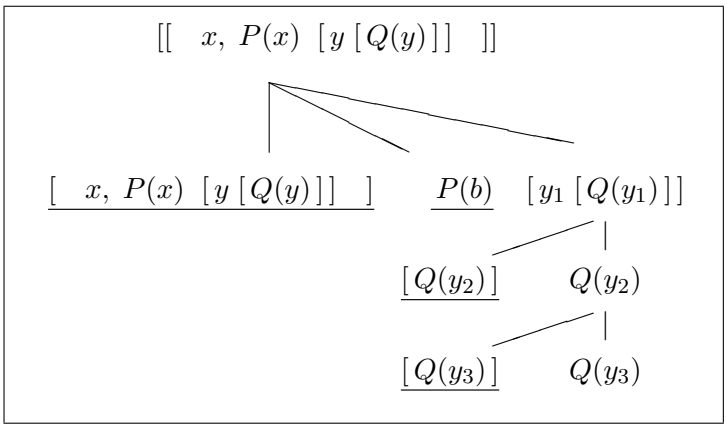


Figure 3.17: An open FOL C^s-tree.

We shall analyse Example 3.6.3. After the first existentially quantified extension, the minor component $[\,y_1\ [\,Q(y_1)\,]\,]$ is a universally quantified formula because it is preceded by the variable y. Remember that we always think of the complement of each node, so that an extra pair of outer brackets is implied around each node. So our assumption is in fact $[[\,y_1\ [\,Q(y_1)\,]\,]]$. So now $[Q(y_1)]$ can be selected any number of times, each time with a new fresh variable. But at each new extension the same node is repeated along that open branch. Although the name of the variable changes from y_2 to y_3, it still holds that $Q(y_2) = Q(y_3)$ because they are free variables. Therefore, these patterns should be detected and not allowed in the tree. Since there are no more universal formulas besides $[Q(y_1)]$ to be selected, and $[Q(y_1)]$ cannot be selected because it is redundant, the tree is considered complete and open.

Finally, the example that follows shows an application of Skolem functions.

Example 3.6.4 *Consider the contingent formula* $\mathcal{F} = \neg\,(\,\forall_x \neg\, Q(x,x)\ \wedge\ \forall_y \exists_z Q(y,z))$. *We have that* $\mathrm{BNNF}(\mathcal{F}) = [\quad x\quad [Q(x,x)]\quad y\quad [\ z\ [Q(y,z)]\quad]\quad]$. *Its open* C^s*-tree is presented in Figure 3.17*

The tree in Figure 3.18 is open and complete, in the sense that subsequent extensions are all redundant. The first split is on $[\ z\ [Q(y,z)]\ ]$, which is existentially quantified on z. However, Q depends on z and also on y, the later being a universal variable whose quantifier appears at a higher level then z, in an ancestor node (in this case the root node). Therefore, variable z has to be replaced by a function that relates it to y. After

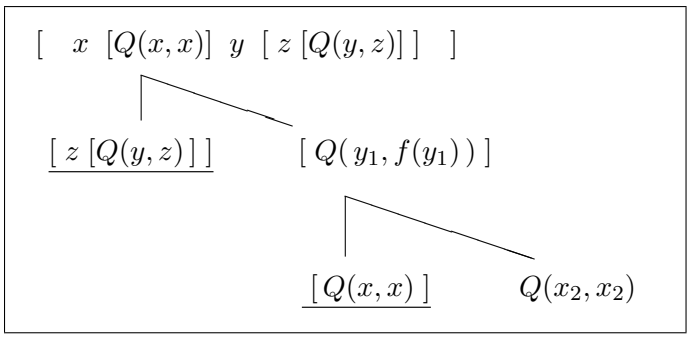


Figure 3.18: Extension with a Skolem function.

making a copy of y into y_1, we replace z with the Skolem function $f(y_1)$. When we try to unify $Q(x_2, x_2)$ with $Q(\, y_1, f(y_1)\,)$ to close the branch, the *occurs check* makes it fail. To unify these terms we would have to unify both arguments by making $x_2 = y_1$ and $x_2 = f(y_1)$. But this means that $y_1 = f(y_1)$ and that $y_1 = f(f(f \ldots f(y_1)))$, indefinitely. Since unification fails, the branch does not close, as it should not. Without the Skolem function the unification would have succeeded. If $f(y_1)$ was instead a Skolem constant b, then we could set $x_2 = y_1$ and $x_2 = b$, yielding $y_1 = b$ as well. The unification would mistakenly close the branch.

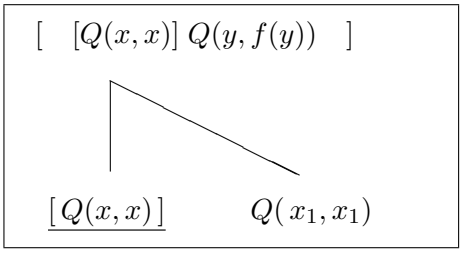


Figure 3.19: Beforehand skolemisation.

Tree expansion in FOL can also be performed with a beforehand skolemisation. In this case, if we want to prove a theorem $\mathcal{T}$, what we should do is the following:

1. Negate the theorem $\mathcal{T}$ into $\neg\,\mathcal{T}$;
2. Skolemise it into $\mathcal{S} = \text{Skolem}\,(\neg\,\mathcal{T})$;
3. Compute $\mathcal{B} = \text{BNNF}\,(\neg\,\mathcal{S})$;
4. Derive a C^s-tree for $\mathcal{B}$.

Doing it this way for the previous example would yield

$$\neg\,\mathcal{F} \;=\; \forall_x \neg\, Q(x,x) \;\wedge\; \forall_y \exists_z\, Q(y,z) \tag{3.6.30}$$

with skolemisation $\mathcal{S} = \text{Skolem}(\neg\,\mathcal{F})$

$$\mathcal{S} \;=\; \neg\, Q(x,x) \wedge Q(y,f(y)) \tag{3.6.31}$$

such that

$$BNNF(\neg\,\mathcal{S}) \;=\; [\,[\,Q(x,x)\,]\,Q(y,f(y))] \tag{3.6.32}$$

The tree expansion is shown in Figure 3.19. Again, $Q(x_1,x_1)$ cannot be unified with $Q(y,f(y))$.

Chapter 4

Conclusions and Perspectives

We have described a proof system that can handle satisfiability testing more efficiently than the analytic tableau method. This has been shown upon comparison with the LeanTap algorithm over a set of propositional problems. This is due to the a set of simplification rules incorporated in the NAND system that accelerate the proofs by providing branch reductions. The propositional NAND is proved to be sound and complete. The first order case is not expected to perform better because the simplifications cannot be freely applied due to variable binding. A possibility for improvements on the first order NAND may reside in a relatively recent idea by Martin Giese [10] that uses delayed unification of variables in order to prevent backtracking.

A few examples of how NAND could be used as a knowledge compilation system have also been presented. They couple the tree expansion method with lemmas and branch reduction. NAND has many features closely related to compilation systems. Future studies could be made on new formula selection strategies, which is always a delicate problem in knowledge compilation.

Besides of its immediate applications, the NAND system has also shown to be a versatile way of representing logical expressions thanks to its BNNF representation. A next interesting step could be to try to incorporate the equivalence operator into the NAND syntax, as it is done with negated form.

Appendix

The NAND Interface

The implementation of the NAND system is written in Prolog. The accepted symbols and predicates for using the program are listed below. Unless stated otherwise, they apply to propositional expressions.

Symbols

Boolean Operators The boolean connectives that can be used to write logical expressions in the NAND program are '&' (and), '#' (*or*), '⇒' (*implication*), '⇔' (*equivalence*) and '∼' (*negation*)

Constant Symbols The constant symbols 'tt' (*true*) and 'ff' (*false*) are allowed in logical expressions.

Propositions Propositional letters must be lower case.

Quantifiers The quantified expressions $\forall_X p(X)$ and $\exists_X p(X)$ are represented by $all(X, p(X))$ and $exists(X, p(X))$, respectively. Variables must be upper case letters and predicates and functions must be lower case.

Higher level predicates

prove(+Theorem, −Model) Proves a Theorem in the form of a logical expression. Returns the counter models for the Theorem or an empty model (the empty list) if the Theorem is valid. Calls **itrans/2**, **ord_simplify/2**, **model/2** and **show/2**.

prove_sentence(+FilePath, −Model) Same as **prove/2** but the theorem is read from a file. The theorem must be stored as a dot ended logical sentence.

verbose/0 The command "**assert(verbose).**" sets the option to view the evolution of the proof. Cancel this option with "**retract(verbose).**".

lemma/0 The command "**assert(lemma).**" introduces the option of using lemmas in the tree expansion. The option is canceled with "**retract(lemma).**".

Translation Predicates

itrans(+Logical Expression, −Nand Expression) Translates a boolean logical expression into a NAND expression. This predicate calls **trans/3**.

itransimp(+Logical Expression, −Nand Expression) Similar to **itrans/2**, but also performs partial simplification of the expression based on the $\top - \bot$ reduction rules.

show(+Nand Expression, −Logical Expression) Translates a NAND expression into a boolean logical expression.

trans(+Logical Expression, −Nand Expression, ?DiffListTail) This predicate is - called by **itrans/2** to handle the inner level subformulas. Double bracket elimination on subformulas is done using difference lists, which is the *DiffListTail* argument of the predicate. The first time it is called $DiffListTail = [\ ]$. Bracket absorption is also performed.

Simplification Predicates

simplify(+Nand Expression, −Nand Simplified) Simplifies a NAND expression using the $\top - \bot$ reduction rules at the literal level. The simplification is recursive over all depth levels and is performed in several passes by calling the predicate **isimp/2**. It does not order nor removes duplicates.

ord_simplify(+Nand Expression, −Nand Simplified) Similar to **simplify/2** but performs recursive ordering of subformulas to remove duplicates.

isimp(+Nand Expression, −Nand Simplified) Performs a one-pass recursive simplification of a Nand Expression over literals.

compound_simplify(+Nand Expression, −Nand Simplified) Similar to **ord-simplify/2**, but applies the $\top - \bot$ reduction rules on compound subformulas as well as literals. Calls **compound_isimp/2**.

compound_isimp(+Nand Expression, −Nand Simplified) Similar to **isimp/2** but performs the simplification on compound subformulas as well as literals.

shanon(+Nand Expression, +Atom, −SE) Performs the shanon expansion (SE) of the Nand expression over the atom Atom.

Tree Expansion Predicates

model(+Nand Expression, −Counter Models) Returns the counter models of a NAND expression using the tree expansion method.

First Order Logic Predicates

prove_fol(+Theorem, −Proof) Proves a skolemised negated theorem. If the returned Proof is the empty list, the theorem is valid.

prove_fol_sentence(+FilePath, −Proof) Same as **prove_fol/2** but the theorem is read from a file. The theorem must be stored as a dot ended logical sentence.

model(+NandFOL, −Proof) Generates a tree expansion proof for a skolemised NAND expression. If Proof is the empty list the expression is valid.

itrans(+FOL, −Nand) Translates a first order logic expression (FOL) into NAND syntax.

skolemise(+FOL, −Skolem) Skolemises a first order logic (FOL) expression.

Bibliography

[1] D'Agostino, M., Mondadori, M. *The Taming of the Cut.* Journal of Logic and Computation, **4** (3) (1994), p.285—319.

[2] D'Agostino, M., Gabbay, D.M., Hähnle, R., and Possega, J. *Handbook of Tableau Methods.* Kluwer Academic Publishers, 1999.

[3] Broda, Krysia. *Automated Reasoning Teaching Materials.* Imperial College London (2006).

[4] Chang, C.L. & Lee, R.T. *Symbolic Logic and Mechanical Theorem Proving.* Academic Press (1971).

[5] Davis, M. and Putnam, H. *A Computing Procedure for Quantification Theory.* J.ACM **7** (1), p.201—215, 1960.

[6] Davis, M., Logemann, G. and Loveland, D. *A Machine Program for Theorem Proving.* Communications of the ACM, **5** (7), p.394—397, 1962.

[7] Darwiche, Adnan. *Compiling Knowledge into Decomposable Negation Normal Form.* Proceedings of the Sixteenth International Joint Conference on Artificial Intelligence, p.284—289, 1999.

[8] Darwiche, Adnan. *Decomposable negation normal form.* Journal of the ACM (JACM), **48** 4, p.608—647, July 2001.

[9] Fitting, Melvin. *First-Order Logic and Automated Theorem Proving.* 2^{nd} Edition. Springer-Verlag, New York (1996).

[10] Giese, M. *Incremental Closure of Free Variable Tableaux.* Proceedings Intl. Joint Conf. on Automated Reasoning, Italy, 2001.

[11] Hähnle R., Murray N.V and Rosenthal E. *Normal Forms for Knowledge Compilation.*Lecture Notes in Computer Science, Springer Berlin / Heidelberg, Vol. 3488, p.304—313, 2005

[12] Hähnle R., Murray N.V and Rosenthal E. *Linearity and Regularity with Negation Normal Form.* Theoretical Computer Science **328** (2004) p.325—354.

[13] McCune, William *OTTER 3.3 Reference Manual.* Tech. Memo ANL/MCS-TM-263. Mathematics and Computer Science Division. Argonne National Laboratory. Argonne, IL (2003). (http://www-unix.mcs.anl.gov/AR/otter/otter33.pdf)

[14] McCune, William *MACE 2.0 Reference Manual and Guide.* Tech. Memo ANL/MCS-TM-249. Mathematics and Computer Science Division. Argonne National Laboratory. Argonne, IL (2001). (http://www-unix.mcs.anl.gov/AR/mace2/mace2.pdf)

[15] Moore, J. Strother *Introduction to the OBDD Algorithm for the ATP Community.* Journal of Automated Reasoning **12**, p.33—45, 1994. Kluwer Academic Press.

[16] Murray, Neil V. and Rosenthal, Erik *Dissolution: Making Paths Vanish.* Journal of the Association for Computer Machinery, **40** (3) (1993), p.504—535.

[17] Murray, Neil V. and Rosenthal, Erik *Tableaux, Path Dissolution, and Decomposable Negation Normal Form for Knowledge Compilation.* Lecture Notes in Computer Science, Springer Berlin / Heidelberg, Vol. 2796, p.165—180. Automated Reasoning with Analytic Tableaux and Related Methods: International Conference, TABLEAUX 2003 Rome, Italy, September 9—12, 2003 Proceedings.

[18] Otten, Jens *On the Advantage of a Non-Clausal Davis-Putnam Procedure.* Technical Report, AIDA-97-1, TH Darmstadt, Intellektik, 1997. (http://www.leancop.de/ncdp/)

[19] Paulson, Lawrence C. *Logic and Proof.* Computer Science Tripos Part IB, Michaelmas Term. University of Cambridge (1999).

[20] Ramesh, A., Beckert, B., Hähnle, R. Murray, Neil V. *Fast Subsumption Checks Using Anti-Links.* Journal of Automated Reasoning **18**, p.47—83, 1997.

[21] Robinson, J.C., Voronkov, A. *Handbook of Automated Reasoning.* Vols I-II. Elsevier and MIT Press 2001.

[22] Robinson, J.C. *A Machine-Oriented Logic Based on the Resolution Principle*. J. ACM **12**(1): 23-41 (1965).

[23] Sandqvist, Tor *Logica Chameleonica.* Uppsala Prints and Preprints in Philosophy (2001:1).

[24] Sheffer, H.M. *A set of five independent postulates for Boolean algebras, with application to logical constants.* Transactions of the American Mathematical Society **14** 481—488, 1913.

[25] Smullyan, Raymond M. *First-Order Logic.* Springer-Verlag, New York (1968).

[26] Yang, Lin *Theorem Proof By Using NAND Expressions.* MSc Dissertation. Imperial College, University of London (2003).

[27] http://i12www.ira.uka.de/leantap/

[28] http://www.cs.miami.edu/∼tptp/