Satisfiability testing for Boolean formulas using Δ-trees *

G. Gutiérrez, I.P. de Guzmán, J. Martínez, M. Ojeda-Aciego and A. Valverde
({gloriagb,guzman,javim,aciego,a_valverde}@ctima.uma.es)
Dept. Matemática Aplicada. Universidad de Málaga, P.O. Box 4114. E-29080 Málaga, Spain.

Abstract. The tree-based data structure of Δ-tree for propositional formulas is improved and optimised. The Δ-trees allow a compact representation for negation normal forms as well as for a number of reduction strategies in order to consider only those occurrences of literals which are relevant for the satisfiability of the input formula. These reduction strategies are divided into two subsets (meaning- and satisfiability-preserving transformations) and can be used to decrease the size of a negation normal form A at (at most) quadratic cost. The reduction strategies are aimed at decreasing the number of required branchings and, therefore, these strategies allow to limit the size of the search space for the SAT problem.

Keywords: Algorithms and data structures, automated deduction.

MSC2000: 68T15 (03B35 68P05)

1. Introduction

Efficient representations for negation normal form (nnfs) are necessary in order to describe and implement efficient algorithms on this kind of formulas. The ability to reason on specifications written in a language as close as possible to natural language is important for information sciences; thus, reasoning efficiently on nnf (negation normal form) is interesting because these formulas are easier to obtain from specifications given in natural language.

Formulas in cnf are usually interpreted as lists of clauses, and formulas in dnf are interpreted as lists of cubes; these interpretations allow efficient descriptions and implementations of algorithms to study satisfiability (e.g. Davis-Putnam procedures, and linear ordered resolution). In this work we use the generalization of these interpretations to nnfs given by the Δ-trees, that is, we use *trees* of clauses and cubes. Specifically, nnfs are represented as trees of clauses and cubes such that each clause-node in the tree is an implicant of the formula represented by its scope and, similarly, each cube-node is an implicate of the formula represented by its scope. The data structure of Δ-tree is so named because its nodes are built up from Δ-lists [1]. After defining the

* Partially supported by Spanish DGI project BFM2000-1054-C02-02.

notion of Δ-tree, the operators `nnf` and Δ`Tree` are introduced which, respectively, associate a nnf to each Δ-tree and vice versa. In addition, it can be shown that this correspondence preserves equivalence and, therefore, we can easily extend the concepts of validity and satisfiability to Δ-trees.

We introduce the concept of restricted Δ-tree (generalizing the well-known concept of restricted cnf in which clauses with repeated or contradictory literals are not allowed and subsumed clauses are omitted), which involves only restricted clauses and cubes in the representation and, in addition, prohibits that a single literal is both an implicant and an implicate of the same subformula. Other representations for nnf formulas have been proposed in the literature: For example, in [7] graphs are used, however, their representation does not differ substantially from the standard one, and it is only useful for the search of *links* which is the main part of the dissolution method they propose. Another widespread alternative representation are the BDDs and its variants [2, 10], but they are not useful for the study of satisfiability because although they make straightforward the testing of satisfiability, the construction of a restricted BDD for a given formula is exponential in the worst case.

Later, we describe some meaning- and satisfiability-preserving transformations in terms of Δ-trees, some of which were introduced in [1] described using the so-called $\widehat{\Delta}$-sets. This fact that $\widehat{\Delta}$-sets are no longer necessary when working with Δ-trees is extremely interesting when implementing the method, since the simple data structure of Δ-tree stores both the information about the structure of the formula and its associated $\widehat{\Delta}$-sets. Finally, the last section includes some experimental results from a Δ-trees based implementation of the method described in [1].

This paper improves the presentation of the results in [5] in that a more compact definition of Δ-trees is introduced. This results in simpler statements of the theorems and, as a consequence, the simplifying transformations are more adequately described in terms of rewrite rules. Specifically, no intermediate operators such as $\Phi_\bot$ and $\Phi_\top$ are required.

2. Preliminary Concepts and Definitions

Throughout the rest of the paper, we will work with a classical propositional language over a denumerable set of propositional variables, $\mathcal{V}$, and connectives $\{\neg, \wedge, \vee\}$, the semantics for this language being the standard one.

- An *assignment* I is an application from the set of propositional variables $\mathcal{V}$ to $\{0,1\}$; the domain of an assignment is uniquely extended to the whole language with the usual definition of the classical connectives.
- A formula A is said to be *satisfiable* if there exists an assignment I such that $I(A) = 1$; in this case I is said to be a *model* for A.
- Two formulas A and B are said to be *equisatisfiable*, denoted $A \approx B$, if A is satisfiable iff B is satisfiable.
- Two formulas A and B are said to be *equivalent*, denoted $A \equiv B$, if $I(A) = I(B)$ for all assignment I.
- We use the symbols $\top$ and $\bot$ to denote truth and falsity.

We will also use the usual notions of literal (propositional variable or the negation of a propositional variable), clause (disjunction of literals) and cube (conjunction of literals). A negation normal form (denoted nnf) is a formula in which the negations are only in the literals, it is said to be conjunctive/disjunctive if its main connective is a conjunction/disjunction. If ℓ is a literal, $\overline{\ell}$ denotes its opposite literal; if Γ is a set of literals, $\overline{\Gamma} = \{\overline{\ell}; \ell \in \Gamma\}$. The following two definitions will play an important rule in the sequel:

- A literal ℓ is an *implicant* of a formula A if $\ell \models A$.
- A literal ℓ is an *implicate* of a formula A if $A \models \ell$.

We will use the standard notions of list and tree. Finite lists are written in juxtaposition, with the standard notation, `nil`, for the empty list; if λ and λ' are lists, $\ell \in \lambda$ denotes that ℓ is an element of λ; the concatenation of two lists λ and λ' is written as either $\lambda\langle\rangle\lambda'$ or $\lambda \cup \lambda'$; the inclusion and intersection of lists are defined in the usual way.

We will work with the usual representation of nnfs as string of symbols and its representation as syntactic tree [4]; this way, an address η in the syntactic tree of a formula A will also mean, when no confusion arises, the subformula of A corresponding to the node of address η in the tree.

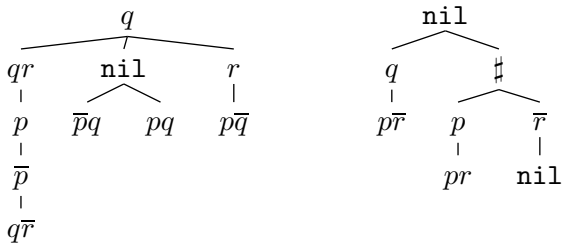
Let A and C be two formulas, let B be a subformula of A and let η be an address in the syntactic tree of A. By $A[B/C]$ we denote the result of substituting in A any occurrence of B by C and, similarly, $A[\eta/C]$ denotes the result of substituting in A the subformula rooted at η by C. If Γ is a set of literals, $A[\Gamma/\top]$ denotes the result of substituting in A any occurrence of literals in Γ by $\top$ and any occurrence of literals in $\overline{\Gamma}$ by $\bot$.

3. The Δ-trees

DEFINITION 1.

1. *A* Δ-list *is either the symbol* ♯ *or a list of literals without repeated propositional variables.*
2. *A* Δ-tree T *is a tree with labels in the set of* Δ*-lists.*

EXAMPLE 1. *The following are two examples of* Δ*-trees:*



A given Δ-tree always represents a conjunctive nnf, however, its subtrees are alternatively interpreted as either conjunctive or disjunctive nnfs, i.e. the immediate subtrees of a conjunctive Δ-tree are disjunctive, and vice versa. It is well-known the identification of the empty clause with the constant $\bot$ and the empty cube with the constant $\top$, that is, the same symbol for the empty list, `nil`, has different conjunctive and disjunctive interpretations. Similarly, we will use the same symbol, ♯, to represent the constants $\bot$ and $\top$ with the conjunctive and disjunctive interpretation respectively.

To improve the efficiency of operators defined on Δ-lists and Δ-trees it is convenient to work with lists of literals ordered wrt the lexicographic order, but the soundness of the transformations is independent from this order and it will not be considered in this theoretical development.

The nnf formula represented by a Δ-tree is determined by the operator `nnf` defined below.

DEFINITION 2. *The operators* `nnf` *and* `dnnf` *over the set of* Δ*-trees are defined as follows:*

1. $\texttt{nnf}(\texttt{nil}) = \top$
2. $\texttt{nnf}(\sharp) = \bot$
3. $\texttt{nnf}\left(\begin{array}{c}\ell_1 \ldots \ell_n \\ \hline T_1 \quad \ldots \quad T_m\end{array}\right) = \ell_1 \wedge \cdots \wedge \ell_n \wedge \texttt{dnnf}(T_1) \wedge \cdots \wedge \texttt{dnnf}(T_m)$
4. $\texttt{dnnf}(\texttt{nil}) = \bot$
5. $\texttt{dnnf}(\sharp) = \top$

6. $\mathtt{dnnf}\left(\begin{array}{c}\ell_1 \ldots \ell_n \\ \diagup \quad \diagdown \\ T_1 \quad \ldots \quad T_m\end{array}\right) = \ell_1 \vee \cdots \vee \ell_n \vee \mathtt{nnf}(T_1) \vee \cdots \vee \mathtt{nnf}(T_m)$

EXAMPLE 2. *The Δ-trees in the previous example are interpreted as nnfs as follows:*

$$\mathtt{nnf}\left(\begin{array}{c} q \\ \diagup \quad \diagdown \\ qr \quad\quad \mathtt{nil} \quad\quad r \\ | \quad\quad \diagup \quad \diagdown \quad\quad | \\ p \quad\quad \overline{p}q \quad\quad pq \quad\quad p\overline{q} \\ | \\ \overline{p} \\ | \\ q\overline{r} \end{array}\right) =$$

$$= q \wedge (q \vee r \vee (p \wedge (\overline{p} \vee (q \wedge \overline{r})))) \wedge (\bot \vee (\overline{p} \wedge q) \vee (p \wedge q)) \wedge (r \vee (p \wedge \overline{q}))$$

$$\mathtt{nnf}\left(\begin{array}{c} \mathtt{nil} \\ \diagup \quad \diagdown \\ q \quad\quad \sharp \\ | \quad\quad \diagup \quad \diagdown \\ p\overline{r} \quad\quad p \quad\quad \overline{r} \\ \quad\quad | \quad\quad | \\ \quad\quad pr \quad\quad \mathtt{nil} \end{array}\right) =$$

$$= \top \wedge (q \vee (p \wedge \overline{r})) \wedge (\top \vee (p \wedge (p \vee r)) \vee (\overline{r} \wedge \bot))$$

Obviously, the logical constants introduced are simplified by using the 0-1 laws.

REMARK 1. *In the rest of the work we will use a simpler notation for the nnfs constructed from the previous operators:*

$$\hat{T} = \mathtt{nnf}(T) \qquad\qquad \check{T} = \mathtt{dnnf}(T)$$

In particular, if $T = \lambda = \ell_1 \ldots \ell_n$:

$$\hat{\lambda} = \ell_1 \wedge \cdots \wedge \ell_n \qquad \textit{and} \qquad \check{\lambda} = \ell_1 \vee \cdots \vee \ell_n$$

The notions of validity, satisfiability, equivalence, equisatisfiability or model are defined by means of the `nnf` operator; for example, a Δ-tree, T is satisfiable if and only if $\mathtt{nnf}(T)$ is satisfiable and the models of T are just the models of $\mathtt{nnf}(T)$.

DEFINITION 3. *The operators* `Union` *and* `Inters` *are defined on the set of Δ-lists as follows. If $\lambda_1, \ldots, \lambda_n$ are Δ-lists then:*

1. $\mathtt{Union}(\lambda_1, \ldots, \lambda_n) = \sharp$ *if either there exists i such that $\lambda_i = \sharp$ or there exists i, j and a literal ℓ such that $\ell \in \lambda_i$ and $\overline{\ell} \in \lambda_j$.*

   $\mathtt{Union}(\lambda_1, \ldots, \lambda_n) = \bigcup_{i=1}^{n} \lambda_i$ *otherwise.*

2. $\ell \in \texttt{Inters}(\lambda_1, \ldots, \lambda_n)$ *if and only if* $\ell \in \lambda_i$ *for all* $\lambda_i \neq \sharp$.

Now we need to prove that any nnf can be represented by a Δ-tree. To do that we associate to each nnf A a pair of Δ-lists denoted $\Delta_0(A)$ and $\Delta_1(A)$, the associated Δ-lists of A. In a nutshell, $\Delta_0(A)$ and $\Delta_1(A)$ are, respectively, lists of implicates and implicants of A.

DEFINITION 4. *Given a nnf* A*, the* Δ*-lists* $\Delta_0(A)$ *and* $\Delta_1(A)$ *are defined recursively as follows:*

$$\begin{array}{ll}
\Delta_0(\ell) = \ell & \Delta_1(\ell) = \ell \\
\Delta_0(\bot) = \sharp & \Delta_1(\bot) = \texttt{nil} \\
\Delta_0(\top) = \texttt{nil} & \Delta_1(\top) = \sharp \\
\Delta_0\left(\bigwedge_{i=1}^{n} A_i\right) = \texttt{Union}(\Delta_0(A_1), \ldots, \Delta_0(A_n)) & \\
\Delta_1\left(\bigwedge_{i=1}^{n} A_i\right) = \texttt{Inters}(\Delta_1(A_1), \ldots, \Delta_1(A_n)) & \\
\Delta_0\left(\bigvee_{i=1}^{n} A_i\right) = \texttt{Inters}(\Delta_0(A_1), \ldots, \Delta_0(A_n)) & \\
\Delta_1\left(\bigvee_{i=1}^{n} A_i\right) = \texttt{Union}(\Delta_1(A_1), \ldots, \Delta_1(A_n)) &
\end{array}$$

THEOREM 1. *Let* A *be a nnf and* ℓ *be a literal in* A *then:*

1. *If* $\ell \in \Delta_0(A)$*, then* $A \models \ell$ *and, equivalently,* $A \equiv \ell \wedge A$.
   If $\Delta_0(A) = \sharp$*, then* $A \equiv \bot$.
2. *If* $\ell \in \Delta_1(A)$*, then* $\ell \models A$ *and, equivalently,* $A \equiv \ell \vee A$.
   If $\Delta_1(A) = \sharp$*, then* $A \equiv \top$.

Proof. We only prove the first item, the second follows similarly.
By induction. For literals and logical constants the result is trivial.

- If $A = A_1 \vee \cdots \vee A_n$ and $\ell \in \Delta_0(A)$, then $\ell \in A_i$ for all A_i with $\Delta_0(A_i) \neq \sharp$. Thus, by induction hypothesis, for every i, either $A_i \equiv \bot$ or $\ell \in \Delta_0(A_i)$ and therefore $A \models \ell$.
- If $A = A_1 \wedge \cdots \wedge A_n$ and $\ell \in \Delta_0(A)$, then there is i such that $\ell \in \Delta_0(A_i)$ and $A \models A_i \models \ell$.
  If $\Delta_0(A) = \sharp$ then either there is i such that $\Delta_0(A_i) = \sharp$ and then $A_i \equiv \bot$ and $A \equiv \bot$ or there are i and j such that $\ell \in \Delta_0(A_i)$ and $\overline{\ell} \in \Delta_0(A_j)$, for some ℓ, and then $A \models A_i \models \ell$, $A \models A_j \models \overline{\ell}$ and $A \equiv \bot$.

■

The following proposition, which is the key to the construction of the Δ-tree associated to a nnf, follows trivially from the definition of the operator `nnf`.

PROPOSITION 1. *If λ is the root of a Δ-tree T then*

$$\Delta_0(\mathtt{nnf}(T)) \supset \lambda \qquad \Delta_1(\mathtt{dnnf}(T)) \supset \lambda$$

Given a nnf A, the operator $\Delta\mathtt{Tree}$ generates a list of Δ-trees; the nodes are the Δ-lists associated to subformulas of A. Note that the $\Delta\mathtt{Tree}$ operator treats differently clauses and cubes, in that the Δ-tree of a cube is considered to be a singleton list whose label is the whole list $\Delta_0(A)$ and the Δ-tree of a clause is considered to be the list of literals understood as a list of Δ-trees (each one being a single literal).

DEFINITION 5. *Let A be a nnf, we define the operators $\Delta\mathtt{Tree}$ and $\mathtt{d}\Delta\mathtt{Tree}$ as follows:*

1. *If $A \neq \top$ is a clause, then $\Delta\mathtt{Tree}(A) = \mathtt{d}\Delta\mathtt{Tree}(A) = \Delta_1(A)$.*
2. *If $A \neq \bot$ is a cube, then*

$$\Delta\mathtt{Tree}(A) = [\Delta_0(A)] \quad \textit{and} \quad \mathtt{d}\Delta\mathtt{Tree}(A) = \Delta_0(A)$$

3. *If A is a disjunctive nnf, and $A_1, \ldots, A_n$ ($n \geq 1$) are the non-literal disjuncts of A, then*

$$\Delta\mathtt{Tree}(A) = \Delta_1(A)\langle\rangle[\Delta\mathtt{Tree}(A_1), \ldots, \Delta\mathtt{Tree}(A_n)]$$

$$\mathtt{d}\Delta\mathtt{Tree}(A) = \overbrace{\Delta\mathtt{Tree}(A_1) \quad \ldots \quad \Delta\mathtt{Tree}(A_n)}^{\Delta_1(A)}$$

4. *Let A be a conjunctive nnf, and let $A_1, \ldots, A_n$ ($n \geq 1$) be the non-literal conjuncts of A, then*

$$\Delta\mathtt{Tree}(A) = \overbrace{\mathtt{d}\Delta\mathtt{Tree}(A_1) \quad \ldots \quad \mathtt{d}\Delta\mathtt{Tree}(A_n)}^{\Delta_0(A)}$$

The reason why it is interesting to consider lists of Δ-trees is that the study of satisfiability of disjunctive formulas leads to a parallel study of the satisfiability of each disjunct. On the other hand, it is interesting to recall the intrinsic parallelism between the usual representation of cnfs as lists of clauses and our representation of nnfs as lists of Δ-trees.

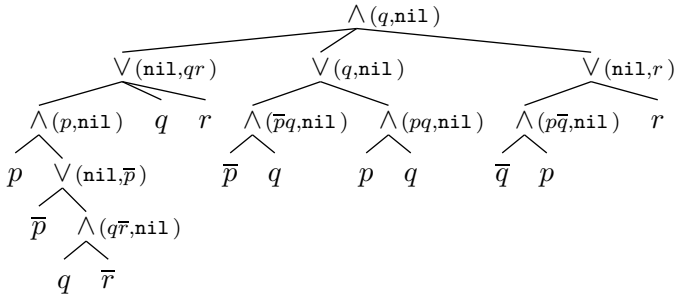
Clause $\rightsquigarrow$	List of literals	Cnf $\rightsquigarrow$	List of clauses
Δ-tree $\rightsquigarrow$	Tree of clauses/cubes	Nnf $\rightsquigarrow$	List of Δ-trees

The next example shows some subtleties of the definition of the Δ-trees operator.

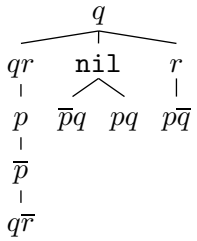
EXAMPLE 3.

1. $\Delta\texttt{Tree}(p \wedge q) = [pq]$.
2. $\Delta\texttt{Tree}(p \vee q) = [p, q]$.
3. $\Delta\texttt{Tree}((p \wedge q) \vee (\overline{q} \wedge (r \vee \overline{p}))) = \left[pq,\ \begin{array}{c}\overline{q}\\ |\\ r\overline{p}\end{array}\right]$.

EXAMPLE 4. *Consider* $A = ((p \wedge (\overline{p} \vee (q \wedge \overline{r}))) \vee q \vee r) \wedge ((\overline{p} \wedge q) \vee (p \wedge q)) \wedge ((\overline{q} \wedge p) \vee r)$, *where every node* η *has associated the pair* $(\Delta_0(\eta), \Delta_1(\eta))$



For the formula A *above we have that* $\Delta\texttt{Tree}(A)$ *is:*



Note that for the previous example $\texttt{nnf}(\Delta\texttt{Tree}(A))$ is *not* equal to A, for a new literal q is attached as an immediate successor of the root node, making explicit that q is an implicate of the formula.

The next theorem shows that the operators `nnf` and $\Delta\texttt{Tree}$ are inverse, up to equivalence.

THEOREM 2.

1. *Let* A *be a nnf and* $\Delta\texttt{Tree}(A) = [T_1, \ldots, T_n]$. *Then* $A \equiv \hat{\wedge}T_1 \vee \cdots \vee \hat{\wedge}T_n$.
2. *If* A *is disjunctive, then* $\texttt{dnnf}(\texttt{d}\Delta\texttt{Tree}(A)) \equiv A$.

Proof. We prove simultaneously the two items by induction on the degree n of A.

i) The basic cases $n = 0$ and $n = 1$ correspond to logical constants, clauses and cubes and thus the proofs are trivial.

ii) Consider that the result is true for $1 \leq n < k$ with $k \geq 1$, and let us prove it for $n = k$.

a) Let A be a disjunctive nnf; we can assume that $A = \ell_1 \vee \ldots \vee \ell_n \vee A_1 \vee \ldots \vee A_m$ with A_i conjunctive and non-literal nnf. In this case, $\Delta\mathtt{Tree}(A) = \Delta_1(A)\langle\rangle[T_1, \ldots T_n]$, where $T_i = \Delta\mathtt{Tree}(A_i)$. By the induction hypothesis $A_i \equiv \mathtt{nnf}(T_i)$ for $1 \leq i \leq m$; in this case, the two items say the same thing, whose proof is:

$$\begin{aligned} \mathtt{dnnf}(\mathtt{d}\Delta\mathtt{Tree}(T)) &= \mathtt{dnnf}(\Delta_1(A)) \vee \hat{T_1} \vee \cdots \vee \hat{T_n} \\ &\equiv \mathtt{dnnf}(\Delta_1(A)) \vee A_1 \vee \cdots \vee A_n \\ &\overset{(1)}{\equiv} \mathtt{dnnf}(\Delta_1(A)) \vee A \overset{(2)}{\equiv} A \end{aligned}$$

The equivalence (1) follows from the fact that the literal successors of A are elements of $\Delta_1(A)$; and the equivalence (2) follows by Theorem 1.

b) Let A be a conjunctive nnf; we can assume that $A = \ell_1 \wedge \ldots \wedge \ell_n \wedge A_1 \wedge \ldots \wedge A_m$ with A_i conjunctive and non-literal nnf. In this case,

$$\Delta\mathtt{Tree}(A) = \overbrace{\mathtt{d}\Delta\mathtt{Tree}(A_1) \quad \ldots \quad \mathtt{d}\Delta\mathtt{Tree}(A_n)}^{\Delta_0(A)}$$

By the induction hypothesis $A_i \equiv \mathtt{dnnf}(\mathtt{d}\Delta\mathtt{Tree}(A_i))$ for $1 \leq i \leq m$, and thus

$$\begin{aligned} &\mathtt{nnf}(\Delta_0(A)) \wedge \mathtt{dnnf}(\mathtt{d}\Delta\mathtt{Tree}(A_1)) \wedge \cdots \wedge \mathtt{dnnf}(\mathtt{d}\Delta\mathtt{Tree}(A_n)) = \\ &\quad \equiv \mathtt{nnf}(\Delta_0(A)) \wedge A_1 \wedge \cdots \wedge A_n \overset{(1)}{\equiv} \mathtt{nnf}(\Delta_0(A)) \wedge A \overset{(2)}{\equiv} A \end{aligned}$$

■

From this result we have that, in some sense, the structure of Δ-tree allows to substitute reasoning with literals by reasoning with clauses and cubes.

The next corollary states some simple conditions for the satisfiability of the nnf represented by a Δ-tree T.

COROLLARY 1. *Consider $A = \hat{T_1} \vee \cdots \vee \hat{T_n}$. If T_i is a tree-leaf, λ, then A is satisfiable and a model for A is given by any assignment I such that $I(\ell) = 1$ for all $\ell \in \lambda$.*

In fact the test for satisfiability given in the last section works by transforming the list of Δ-trees until one of them (if any) gets reduced to a single leaf, otherwise the input formula is unsatisfiable.

4. Restricted Δ-trees

The aim of this section is to generalise the well-known definition of restricted clauses, in which opposite literals and logical constants are not allowed. We can say that restricted Δ-trees are Δ-trees without *trivially* redundant information.

Conclusive nodes

DEFINITION 6. *A node of a Δ-tree T is said to be conclusive if it satisfies any of the following conditions:*

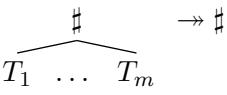
- *It is labelled with $\sharp$, provided that $T \neq \sharp$.*
- *It is either a leaf or a monary node labelled with* `nil`*, provided that it is not the root node.*
- *It is labelled with λ, it has an immediate successor λ' which is a leaf and $\overline{\lambda'} \subseteq \lambda$.*
- *It is labelled with λ and* $\texttt{Inters}(\lambda, \lambda') \neq \texttt{nil}$*, where λ' is the label of its predecessor.*

Intuitively, the previous definition detects those nodes in the Δ-tree which, in some sense, can be substituted by either $\bot$ or $\top$ without affecting the meaning. The effective deletion of those nodes is made by the rewriting rules introduced in Theorem 3 below.

Note that the rewriting rules have a double meaning; since they needn't apply to the root node, the interpretation can be either conjunctive or disjunctive. This is just another efficiency-related feature of Δ-trees: duality of connectives $\wedge$ and $\vee$ gets subsumed in the structure and it is not necessary to determine the conjunctive/disjunctive character to decide the transformation to be applied.

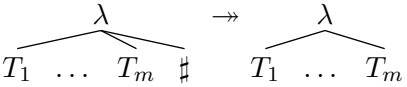
THEOREM 3. *The following rewriting rules (up to the order of the successors) allows to delete the conclusive nodes of a Δ-tree.*

Rule C1

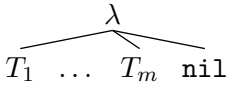


$\twoheadrightarrow \sharp$

Rule C2

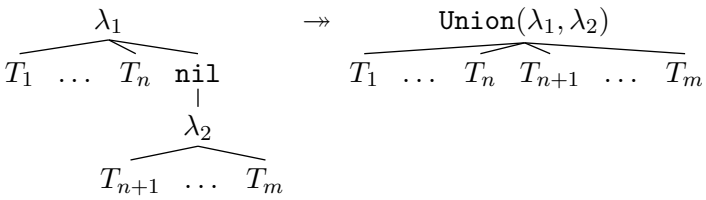


Rule C3

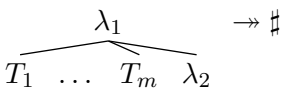


$\twoheadrightarrow \sharp$

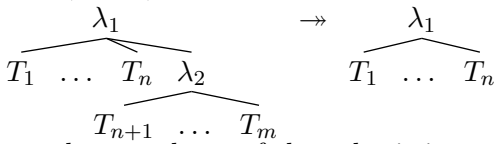
Rule C4



Rule C5 *If* $\overline{\lambda_2} \subseteq \lambda_1$ *then*



Rule C6 *If* $\texttt{Inters}(\lambda_1, \lambda_2) \neq \texttt{nil}$ *then*



Proof. To prove the soundness of the rules it is enough to establish the semantics of each one of them:

C1 $\texttt{nnf}\left(\begin{array}{c}\sharp\\ T_1 \ \ldots \ T_m\end{array}\right) = \bot \wedge \overset{\vee}{T_1} \wedge \cdots \wedge \overset{\vee}{T_n} \equiv \bot = \texttt{nnf}(\sharp)$

$\texttt{dnnf}\left(\begin{array}{c}\sharp\\ T_1 \ \ldots \ T_m\end{array}\right) = \top \vee \overset{\wedge}{T_1} \vee \cdots \vee \overset{\wedge}{T_n} \equiv \bot = \texttt{dnnf}(\sharp)$

C2 $\texttt{nnf}\left(\begin{array}{c}\lambda\\ T_1 \ \ldots \ T_m \ \sharp\end{array}\right) = \overset{\wedge}{\lambda} \wedge \overset{\vee}{T_1} \wedge \cdots \wedge \overset{\vee}{T_n} \wedge \top$

$\equiv \overset{\wedge}{\lambda} \wedge \overset{\vee}{T_1} \wedge \cdots \wedge \overset{\vee}{T_n} = \texttt{nnf}\left(\begin{array}{c}\lambda\\ T_1 \ \ldots \ T_m\end{array}\right)$

$\texttt{dnnf}\left(\begin{array}{c}\lambda\\ T_1 \ \ldots \ T_m \ \sharp\end{array}\right) = \overset{\vee}{\lambda} \vee \overset{\wedge}{T_1} \wedge \cdots \vee \overset{\wedge}{T_n} \vee \bot$

$\equiv \overset{\vee}{\lambda} \vee \overset{\wedge}{T_1} \vee \cdots \vee \overset{\wedge}{T_n} = \texttt{dnnf}\left(\begin{array}{c}\lambda\\ T_1 \ \ldots \ T_m\end{array}\right)$

In the rest of the proof we will not write the evaluation of the `nnf` and `dnnf` operators, that is the first and last members of the equality will be not shown.

C3 $\overset{\wedge}{\lambda} \wedge \overset{\vee}{T_1} \wedge \cdots \wedge \overset{\vee}{T_n} \wedge \bot \equiv \bot$

$\overset{\vee}{\lambda} \vee \overset{\wedge}{T_1} \vee \cdots \vee \overset{\wedge}{T_n} \vee \top \equiv \top$

C4 $\overset{\wedge}{\lambda_1} \wedge \overset{\vee}{T_1} \wedge \cdots \wedge \overset{\vee}{T_n} \wedge (\bot \vee (\overset{\wedge}{\lambda_2} \wedge \overset{\vee}{T_{n+1}} \wedge \cdots \wedge \overset{\vee}{T_m}))$

$\equiv \overset{\wedge}{\lambda_1} \wedge \overset{\vee}{T_1} \wedge \cdots \wedge \overset{\vee}{T_n} \wedge \overset{\wedge}{\lambda_2} \wedge \overset{\vee}{T_{n+1}} \wedge \cdots \wedge \overset{\vee}{T_m}$

$\equiv \overset{\wedge}{\lambda_1} \wedge \overset{\wedge}{\lambda_2} \wedge \overset{\vee}{T_1} \wedge \cdots \wedge \overset{\vee}{T_n} \wedge \overset{\vee}{T_{n+1}} \wedge \cdots \wedge \overset{\vee}{T_m}$

$\equiv \texttt{nnf}(\texttt{Union}(\lambda_1, \lambda_2)) \wedge \overset{\vee}{T_1} \wedge \cdots \wedge \overset{\vee}{T_n} \wedge \overset{\vee}{T_{n+1}} \wedge \cdots \wedge \overset{\vee}{T_m}$

$\overset{\vee}{\lambda_1} \vee \overset{\wedge}{T_1} \vee \cdots \vee \overset{\wedge}{T_n} \vee (\top \wedge (\overset{\vee}{\lambda_2} \vee \overset{\wedge}{T_{n+1}} \vee \cdots \vee \overset{\wedge}{T_m}))$

$$\equiv \overset{\vee}{\lambda}_1 \vee \hat{T}_1 \vee \cdots \vee \hat{T}_n \vee \overset{\vee}{\lambda}_2 \vee \hat{T}_{n+1} \vee \cdots \vee \hat{T}_m$$
$$\equiv \overset{\vee}{\lambda}_1 \vee \overset{\vee}{\lambda}_2 \vee \hat{T}_1 \vee \cdots \vee \hat{T}_n \vee \hat{T}_{n+1} \vee \cdots \vee \hat{T}_m$$
$$\equiv \mathtt{dnnf}(\mathtt{Union}(\lambda_1, \lambda_2)) \vee \hat{T}_1 \vee \cdots \vee \hat{T}_n \vee \hat{T}_{n+1} \vee \cdots \vee \hat{T}_m$$

C5 We can assume that $\lambda_1 = \ell_1 \dots \ell_s \ell_{s+1} \dots \ell_r$ and $\lambda_2 = \bar{\ell}_1 \dots \bar{\ell}_s$; then:

$$\ell_1 \wedge \cdots \wedge \ell_s \wedge \ell_{s+1} \wedge \cdots \wedge \ell_r \wedge \overset{\vee}{T}_1 \wedge \cdots \wedge \overset{\vee}{T}_n \wedge (\bar{\ell}_1 \vee \cdots \vee \bar{\ell}_s)$$
$$\equiv (\ell_1 \wedge \cdots \wedge \ell_s) \wedge \ell_{s+1} \wedge \cdots \wedge \ell_r \wedge \overset{\vee}{T}_1 \wedge \cdots \wedge \overset{\vee}{T}_n \wedge \neg(\ell_1 \wedge \cdots \wedge \ell_s)$$
$$\equiv \bot$$
$$\ell_1 \vee \cdots \vee \ell_s \vee \ell_{s+1} \vee \cdots \vee \ell_r \vee \hat{T}_1 \vee \cdots \vee \hat{T}_n \vee (\bar{\ell}_1 \wedge \cdots \wedge \bar{\ell}_s)$$
$$\equiv (\ell_1 \vee \cdots \vee \ell_s) \vee \ell_{s+1} \vee \cdots \vee \ell_r \vee \hat{T}_1 \vee \cdots \vee \hat{T}_n \vee \neg(\ell_1 \vee \cdots \vee \ell_s)$$
$$\equiv \top$$

C6 Let ℓ be a literal such that $\ell \in \mathtt{Inters}(\lambda_1, \lambda_2)$; then $\overset{\wedge}{\lambda}_1 \equiv \ell \wedge C_1$, $\overset{\wedge}{\lambda}_2 \equiv \ell \wedge C_2$, $\overset{\vee}{\lambda}_1 \equiv \ell \vee D_1$ and $\overset{\vee}{\lambda}_2 \equiv \ell \vee D_2$ and, by the absorption rules we have:

$$\overset{\wedge}{\lambda}_1 \wedge \overset{\vee}{T}_1 \wedge \cdots \wedge \overset{\vee}{T}_n \wedge (\overset{\vee}{\lambda}_2 \vee \hat{T}_{n+1} \vee \cdots \vee \hat{T}_m)$$
$$\equiv \ell \wedge C_1 \wedge \overset{\vee}{T}_1 \wedge \cdots \wedge \overset{\vee}{T}_n \wedge (\ell \vee D_2 \vee \hat{T}_{n+1} \vee \cdots \vee \hat{T}_m)$$
$$\equiv \overset{\wedge}{\lambda}_1 \wedge \overset{\vee}{T}_1 \wedge \cdots \wedge \overset{\vee}{T}_n$$
$$\overset{\vee}{\lambda}_1 \vee \hat{T}_1 \vee \cdots \vee \hat{T}_n \wedge (\overset{\wedge}{\lambda}_2 \wedge \overset{\vee}{T}_{n+1} \wedge \cdots \wedge \overset{\vee}{T}_m)$$
$$\equiv \ell \vee D_1 \vee \hat{T}_1 \vee \cdots \vee \hat{T}_n \vee (\ell \wedge C_2 \wedge \overset{\vee}{T}_{n+1} \wedge \cdots \wedge \overset{\vee}{T}_m)$$
$$\equiv \overset{\vee}{\lambda}_1 \vee \hat{T}_1 \vee \cdots \vee \hat{T}_n$$

Note that, for each rule, the resulting Δ-tree is strictly smaller than the input, therefore the application of these rules always terminates. ∎

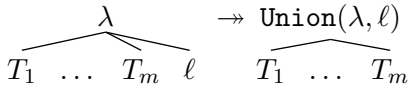
Simple leaves

As we are considering a tree of lists of literals, it might happen that some of the lists of literals in the leaves are singletons. In this case, those leaves are redundant, for they are not proper clauses or cubes, but literals.

DEFINITION 7. *Let T be a non-leaf Δ-tree, a leaf in T is said to be* simple *if it is labelled with just one literal.*

THEOREM 4. *The following rewriting rule (up to the order of successors) delete the simple leaves of a Δ-tree.*

Rule S



Proof. The interpretations of the transformation is stated below:

$$\begin{aligned}
\hat{\lambda} \wedge \check{T}_1 \wedge \cdots \wedge \check{T}_m \wedge \ell &= \hat{\lambda} \wedge \ell \wedge \check{T}_1 \wedge \cdots \wedge \check{T}_m \\
&= \mathtt{nnf}(\mathtt{Union}(\lambda, \ell)) \wedge \check{T}_1 \wedge \cdots \wedge \check{T}_m \\
\check{\lambda} \vee \hat{T}_1 \vee \cdots \vee \hat{T}_m \vee \ell &= \check{\lambda} \vee \ell \vee \hat{T}_1 \vee \cdots \vee \hat{T}_m \\
&= \mathtt{dnnf}(\mathtt{Union}(\lambda, \ell)) \vee \hat{T}_1 \vee \cdots \vee \hat{T}_m
\end{aligned}$$

■

Note that some situations in the rewrite rules (actually, C1–C4 and S) are not possible for a well-formed input formula, but may well arise after applying some rules.

Updatable nodes

A third source for redundant information can be stated in terms of the relationship between the common information in consecutive Δ-lists in a single branch.

DEFINITION 8. *Let T be a Δ-tree, and λ be the label of a node of T. Let λ' be the label of one immediate successor of λ and let $\lambda_1, \ldots, \lambda_n$ be the labels of the immediate successors of λ'. We say that λ* can be updated *if it satisfies some of the next conditions:*

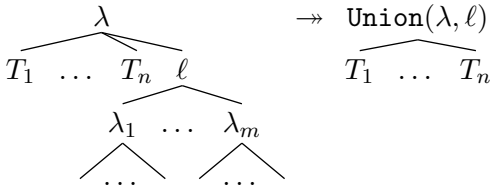
1. *$\lambda' = nil$ and $\mathtt{Inters}(\lambda_1, \ldots, \lambda_m) \not\subset \lambda$.*
2. *$\lambda' = \ell$, $\ell \notin \lambda$ and $\ell \in \mathtt{Inters}(\lambda_1, \ldots, \lambda_m)$.*

We say that T is updated *if it has no nodes that can be updated.*

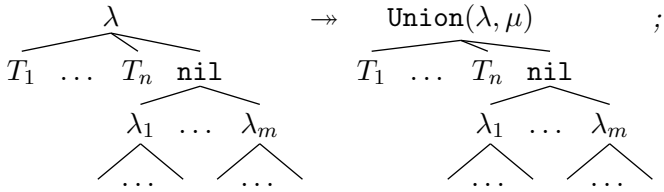
In order to obtain an updated Δ-tree, we have to drive upwards all those literals that can be generated by intersections; this operation is done by the rewriting rules introduced in the following theorem.

THEOREM 5. *The following rewriting rules (up to the order of successors) update every non-updated node of a Δ-tree.*

Rule U1 *If $\ell \in \mathtt{Inters}(\lambda_1, \ldots, \lambda_m)$,*



Rule U2 *If* $\mu = \texttt{Inters}(\lambda_1, \ldots, \lambda_m) \neq \texttt{nil}$.



;

Proof. The soundness of each rule is proved below. We will simply prove the conjunctive interpretation of the rules, the other follow easily by duality.

U1 In this case it is enough to show that the right-most subtree of the input tree is equivalent to the single literal ℓ.
$\ell \vee ((\ell \wedge A_1) \vee \cdots \vee (\ell \wedge A_m)) \equiv \ell \vee (\ell \wedge (A_1 \vee \cdots \vee A_m)) \equiv \ell$

U2 For this rule the two sides of the rule are proved to be equivalent

$$\begin{aligned}
&\hat{\lambda} \wedge \check{T}_1 \wedge \cdots \wedge \check{T}_n \wedge ((\hat{\mu} \wedge A_1) \vee \cdots \vee (\hat{\mu} \wedge A_m)) \\
&\quad \equiv \hat{\lambda} \wedge \check{T}_1 \wedge \cdots \wedge \check{T}_n \wedge ((\hat{\mu} \wedge \hat{\mu} \wedge A_1) \vee \cdots \vee (\hat{\mu} \wedge \hat{\mu} \wedge A_m)) \\
&\quad \equiv \hat{\lambda} \wedge \check{T}_1 \wedge \cdots \wedge \check{T}_n \wedge \hat{\mu} \wedge ((\hat{\mu} \wedge A_1) \vee \cdots \vee (\hat{\mu} \wedge A_m)) \\
&\quad \equiv \texttt{nnf}(\texttt{Union}(\lambda, \mu)) \wedge \check{T}_1 \wedge \cdots \wedge \check{T}_n \wedge ((\hat{\mu} \wedge A_1) \vee \cdots \vee (\hat{\mu} \wedge A_m))
\end{aligned}$$

■

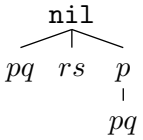
Restricted Δ-trees

DEFINITION 9. *Let T be a Δ-tree. If T is updated and it has neither conclusive nodes nor simple leaves, then it is said to be* restricted.

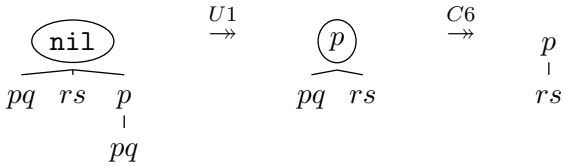
THEOREM 6. *If T is a Δ-tree, there exists a list of restricted Δ-trees, $[T_1, \ldots, T_n]$, such that* $\texttt{nnf}(T) \equiv \hat{T}_1 \vee \cdots \vee \hat{T}_n$. *Specifically, if T' is the Δ-tree obtained from T by exhaustively applying the rules C1, C2, C3, C4, C5, C6, S, U1 and U2 till no one of them can be applied any more, then the list of restricted Δ-trees is:*

$$\texttt{Restrict}(T) = \begin{cases} \lambda\langle\rangle[T_1, \ldots, T_n] & \text{if } T' = \begin{array}{c} \texttt{nil} \\ | \\ \lambda \\ T_1 \ \ldots \ T_n \end{array} \\ [T'] & \text{otherwise} \end{cases}$$

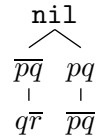
EXAMPLE 5. *Let us consider the Δ-tree*



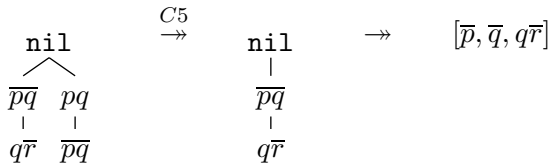
Its restricted form is obtained as follows (the circled nodes are the nodes to be rewritten):



EXAMPLE 6. *Let us consider the* Δ*-tree*



Its restricted form is obtained as follows (the circled nodes are the nodes to be rewritten):



In a restricted Δ-tree we have that the labels contain all the information provided by the Δ_0 and Δ_1 operators. Formally, the following proposition, whose proof is straightforward, states that the Δ-lists in a Δ-tree are the sets of implicants and implicates.

PROPOSITION 2. *If* T *is a* Δ*-tree and* λ *is the label of the root of* $\texttt{Restrict}(T)$*, then*

$$\Delta_0(\texttt{nnf}(T)) = \Delta_1(\texttt{dnnf}(T)) = \lambda$$

5. Reduction of Δ-trees

In this section we introduce the reductions used by the TAS algorithm to be given in the last section. The two first reductions are based on the following theorem.

THEOREM 7. *Let* A *be a nnf and let* η *be an arbitrary node*[1] *of* A*.*

[1] Here, η denotes a node in the syntactic tree of A. In this way, the theorem states properties of the substitution of just an occurrence of a subformula of A.

1. *If* $\ell \in \Delta_0(A)$, *then*
   a) $A \equiv A[\eta/\eta \wedge \ell]$
   b) $A \equiv \ell \wedge A[\eta/\eta \vee \overline{\ell}]$
2. *If* $\ell \in \Delta_1(A)$, *then*
   a) $A \equiv A[\eta/\eta \vee \ell]$
   b) $A \equiv \ell \vee A[\eta/\eta \wedge \overline{\ell}]$

Proof. We will only prove the first item, the proof of the second is similar.

1.a) Let I be an arbitrary assignment,

- If $I(\ell) = 1$, then $I(\eta) = I(\eta \wedge \ell)$ and therefore, $I(A) = I(A[\eta/\eta \wedge \ell])$
- If $I(\ell) = 0$, then by Theorem 1 we have $I(A) = 0$; in addition, $I(\eta) \geq I(\eta \wedge \ell) = 0$ and, by monotonicity of $\wedge$ and $\vee$, we have that $I(A[\eta/\eta \wedge \ell]) \leq I(A) = 0$. Therefore,

$$I(A) = 0 = I(A[\eta/\eta \wedge \ell])$$

1.b) Let I be an arbitrary assignment,

- If $I(\ell) = 1$, then $I(\eta) = I(\eta \vee \overline{\ell})$ and therefore,

$$I(A) = I(A[\eta/\eta \vee \overline{\ell}]) = I(A[\eta/\eta \vee \overline{\ell}] \wedge \ell)$$

- If $I(\ell) = 0$, then by Theorem 1 we have $I(A) = 0$ and therefore,

$$I(A[\eta/\eta \vee \overline{\ell}] \wedge \ell) = 0 = I(A)$$

■

As an easy consequence of the previous theorem we get the corollaries below which will be used to prove the soundness of the reductions defined in the following sections.

COROLLARY 2. *Let* A *be a nnf and* ℓ *a literal in* A. *Then:*

1. *If* $\ell \in \Delta_0(A)$, *then* $A \equiv \ell \wedge A[\ell/\top]$.
2. *If* $\ell \in \Delta_1(A)$, *then* $A \equiv \ell \vee A[\overline{\ell}/\top]$.

COROLLARY 3. *Let* A *be a nnf; if* $\ell \in \Delta_0(A)$, *then* $A \approx A[\ell/\top]$.

Now, to apply the results above, we need to define the substitutions of literals by constants in a Δ-tree. Specifically, we are going to define the substitution of a literal by $\top$, that is, the syntactical partial evaluation of a Δ-tree.

DEFINITION 10. *If $\mu \neq \sharp$ is a Δ-list, we define the operator $[\mu]$ over the set of Δ-trees, as follows:*

$$[\mu]\left(\begin{array}{c}\lambda\\ \diagup\ \ \diagdown\\ T_1\ \ \ldots\ \ T_n\end{array}\right) = \begin{cases} \sharp & \text{if } \texttt{Inters}(\mu, \overline{\lambda}) \neq \texttt{nil} \\ \begin{array}{c}\lambda \smallsetminus \mu\\ \diagup\ \ \diagdown\\ [\overline{\mu}]T_1\ \ \ldots\ \ [\overline{\mu}]T_n\end{array} & \text{otherwise} \end{cases}$$

where $\lambda \smallsetminus \mu$ denotes the list obtained by deleting in λ all the literals in μ.

The following easy-to-prove lemma states that the definition we have just given coincides with the usual meaning of substitution in formulas.

LEMMA 1. *For every Δ-list μ and every Δ-tree T we have:*

$$\texttt{nnf}([\mu]T) \equiv \texttt{nnf}(T)[\mu/\top]$$

5.1. Subreduction

All the transformations performed by the operator `Restrict` only use the information of a node and its immediate successors. The next transformation uses the information in a node to simplify *all* its descendants.

DEFINITION 11. *Operators* `SubReduce` *and* `dSubReduce` *are defined on the set of restricted Δ-trees as follows:*

1. $\texttt{SubReduce}(\lambda) = \texttt{dSubReduce}(\lambda) = \lambda$ *for all Δ-list λ.*
2. $$\texttt{SubReduce}\left(\begin{array}{c}\lambda\\ \diagup\ \ \diagdown\\ T_1\ \ \ldots\ \ T_n\end{array}\right) = \begin{array}{c}\lambda\\ \diagup\ \ \diagdown\\ \texttt{dSubReduce}([\lambda]T_1)\ \ \ldots\ \ \texttt{dSubReduce}([\lambda]T_n)\end{array}$$
3. $$\texttt{dSubReduce}\left(\begin{array}{c}\lambda\\ \diagup\ \ \diagdown\\ T_1\ \ \ldots\ \ T_n\end{array}\right) = \begin{array}{c}\lambda\\ \diagup\ \ \diagdown\\ \texttt{SubReduce}([\overline{\lambda}]T_1)\ \ \ldots\ \ \texttt{SubReduce}([\overline{\lambda}]T_n)\end{array}$$

THEOREM 8. *Let T be a Δ-tree. Then* $\texttt{SubReduce}(T) \equiv T$.

Proof. The result is a consequence of the following equivalences, which are an immediate consequence of Corollary 2:

$$\texttt{nnf}\left(\begin{array}{c}\lambda\\ T_1 \quad \ldots \quad T_n\end{array}\right) \equiv \begin{array}{c}\lambda\\ [\lambda]T_1 \quad \ldots \quad [\lambda]T_n\end{array}$$

$$\texttt{dnnf}\left(\begin{array}{c}\lambda\\ T_1 \quad \ldots \quad T_n\end{array}\right) \equiv \begin{array}{c}\lambda\\ [\overline{\lambda}]T_1 \quad \ldots \quad [\overline{\lambda}]T_n\end{array} \qquad \blacksquare$$

The following proposition, which follows easily from the definition of subreduction, states that only the dominant occurrences of literals are present in a subreduced Δ-tree.

PROPOSITION 3. *Let T be a Δ-tree. In every branch of* $\texttt{SubReduce}(T)$ *there is at most one occurrence of each propositional variable. In particular, if ℓ is a literal in* $\texttt{SubReduce}(T)$*, then there is no occurrence of $\overline{\ell}$ under ℓ.*

5.2. Complete reduction

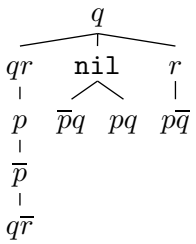
In this section we introduce a satisfiability-preserving transformation which, in essence, is a refinement of the subreduction of the Δ-list of the root.

DEFINITION 12. *A Δ-tree with non-empty root is said to be completely reducible.*

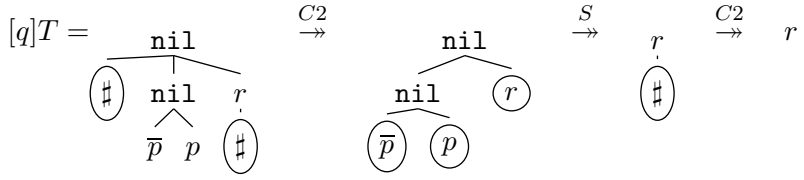
The following results is the Δ-tree formulation of the corollary 3.

THEOREM 9. *If $\mu \neq \texttt{nil}$ is the root of T, then $T \approx [\mu]T$. Moreover, if I is a model of $[\mu]T$, the extension defined as $I(\ell) = 1$ if $\ell \in \mu$ is a model of T.*

EXAMPLE 7. *Given the first Δ-tree of Example 1*



we have that this Δ-tree, T, is equisatisfiable to $[q]T$. The restricted form of $[q]T$ is obtained as follows:



As $[q]T \equiv r$ *is satisfiable and* $I(r) = 1$ *is a model; we have that,* $I(r) = 1 = I(q)$ *is a model of* T*.*

5.3. Pure literals

The concept of pure literal for nnfs in [9] can be immediately extended for Δ-trees, by using Theorem 2. In addition, by means of the subreduction we can define a more general concept of purity for Δ-trees.

DEFINITION 13. *Let* T *be a* Δ*-tree. We say that* ℓ *is a* Δ-pure *literal in* T *if every occurrence of* $\overline{\ell}$ *in* T *is under an occurrence of* ℓ*.*

THEOREM 10. *If* ℓ *is a* Δ*-pure literal in* T*, then* T *is satisfiable iff* $[\ell]T$ *is satisfiable. Moreover, if* I *is a model of* $[\ell]T$*, then the extension defined as* $I(\ell) = 1$ *is a model of* T*.*

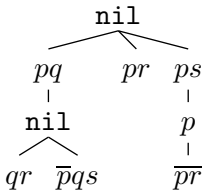
Proof. It ℓ is Δ-pure in T, then, by proposition 3, ℓ is pure in $\mathtt{SubReduce}(T)$; therefore:

$$\mathtt{nnf}(T) \equiv \mathtt{nnf}(\mathtt{SubReduce}(T)) \approx \mathtt{nnf}([\ell]\,\mathtt{SubReduce}(T)) \equiv \mathtt{nnf}([\ell]T)$$

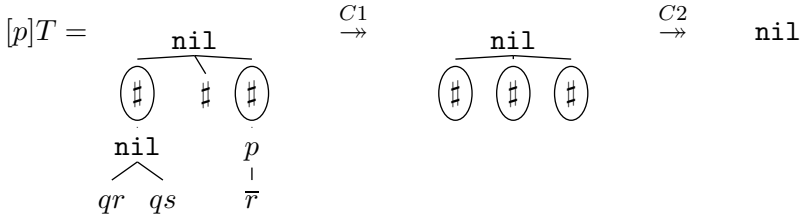
■

Note that, although the soundness of the Δ-pure literals deletion has been proved by means of the subreduction, its applications will be performed without this intermediate transformation.

EXAMPLE 8. *Let us consider the following* Δ*-tree,* T*:*



Literal p *is non pure in* T *but it is* Δ*-pure; then* T *is satisfiable if and only if* $[p]T$ *is satisfiable:*



This Δ-tree is valid and then T is satisfiable and $I(p) = 1$ is a model.

6. The TAS satisfiability algorithm

1. The information flow of the algorithm is a list of pairs such as $[(T_1, \mu_1), \ldots, (T_m, \mu_m)]$, where the T_i are Δ-trees and μ_i are lists of literals which, as we will see, can be used to build a countermodel (if any).
   The initial list to study the satisfiability of a single formula A is
   $$[(T_1, \mathtt{nil}), \ldots, (T_n, \mathtt{nil})]$$
   where
   a) $[T_1, \ldots, T_n] = \Delta\mathtt{Tree}(A)$ if the satisfiability of A is to be tested.
   b) $[T_1, \ldots, T_n] = \Delta\mathtt{Tree}(\neg A)$ if the validity of A is to be tested.
   c) $[T_1, \ldots, T_n] = \Delta\mathtt{Tree}(A_1 \wedge \cdots \wedge A_n \wedge \neg A)$ if the validity of the logical consequence $A_1, \ldots, A_n \models A$ is to be tested.

   In the rest of the items we will always refer to satisfiability testing of a formula.
2. If $[(T_1, \mu_1), \ldots, (T_m, \mu_m)]$ is the flow in some instant during the execution of the algorithm, then the initial Δ-tree is unsatisfiable if and only if every T_i is unsatisfiable, that is $T_i = \sharp$ for all i. The actual search done by test for satisfiability is to obtain an element in the list of tasks which is equal to one the following possibilities:
   a) $(\mathtt{nil}, \mu)$, in this case, the input formula is satisfiable and a model is given by assigning 1 to any element in μ.
   b) (λ, μ), in this case, the input formula is satisfiable and a model is given by assigning 1 to any element in $\lambda \cup \mu$.
3. Updating: After each reduction, the Δ-trees are converted to restricted form (this step also applies to the initial list):
   Given $[(T_1, \mu_1), \ldots, (T_m, \mu_m)]$, its updated form is
   $$@(\mathtt{Restrict}(T_1), \mu_1)\langle\rangle \cdots \langle\rangle @(\mathtt{Restrict}(T_m), \mu_m)$$
   where the operator $@(T :: list, \mu)$ is defined as $(T, \mu) :: @(list, \mu)$.

4. COMPLETE REDUCTION: if some of the elements of the list of tasks is completely reducible, then the corresponding transformation is applied:

$$[\ldots, \left(\overset{\lambda}{\underset{T_1 \quad \ldots \quad T_n}{\diagup\!\diagdown}}, \mu \right), \ldots] \twoheadrightarrow [\ldots, \left(\overset{\texttt{nil}}{\underset{[\lambda]T_1 \quad \ldots \quad [\lambda]T_n}{\diagup\!\diagdown}}, \mu \cup \lambda \right), \ldots]$$

5. Δ-PURE LITERALS: If (T, μ) is an element of the list of tasks and π is a (non-empty) list of Δ-pure literals of T, then the following reduction is applied:

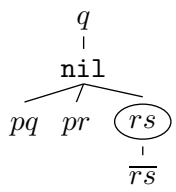
$$[\ldots, (T, \mu), \ldots] \qquad \twoheadrightarrow \qquad [\ldots, ([\pi]T, \mu \cup \pi), \ldots]$$

6. SUBREDUCTION: If no task is either completely reducible or has pure literals, then the subreduction transformation is applied, and the result is updated (obviously, only if some modification has been made).

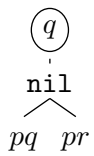
7. QUINE: Finally, if no transformation applies to the list of tasks, then a random task is chosen together with a literal ℓ to branch on, and the following transformation is applied:

$$[\ldots, (T, \mu), \ldots] \qquad \twoheadrightarrow \qquad [\ldots, ([\ell]T, \mu \cup \{\ell\}), ([\overline{\ell}]T, \mu \cup \{\overline{\ell}\}), \ldots]$$

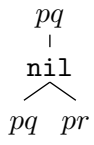
EXAMPLE 9. *Let us study the satisfiability of the formula* $A = ((p \wedge q) \vee (p \wedge r) \vee ((\overline{r} \vee \overline{s}) \wedge r \wedge s)) \wedge q$, *whose associated* Δ*-tree is*



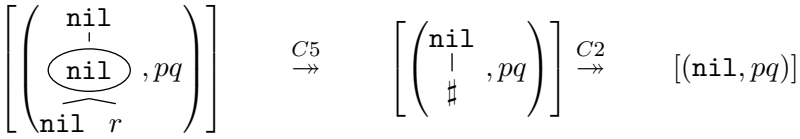
Using rule C5 on the circled node we get the Δ*-tree*



The root can be updated, by using rule U2, obtaining a completely reducible Δ*-tree*

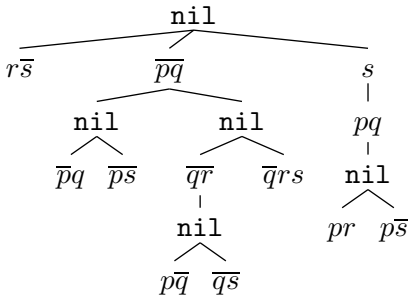


Applying complete reduction we get

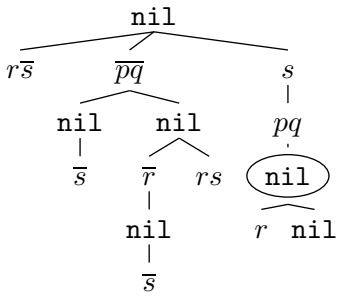


As a result we get that the input formula is satisfiable, and a model is given by any assignment I such that $I(p) = I(q) = 1$.

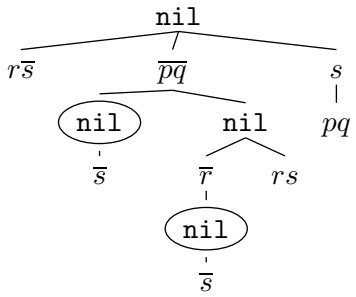
EXAMPLE 10. *Let us study the satisfiability of the following nnf* $A = (r \vee \overline{s}) \wedge (((\overline{p} \vee q) \wedge (\overline{p} \vee \overline{s})) \vee ((\overline{r} \vee ((\overline{q} \vee p) \wedge (\overline{s} \vee \overline{q}))) \wedge (\overline{q} \vee s \vee r))) \wedge (((p \wedge r) \vee (p \wedge \overline{s})) \wedge q) \vee s)$. *Its associated* Δ*-tree is*



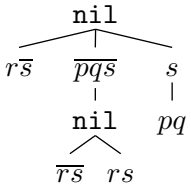
It is not difficult to check that this is a restricted Δ*-tree, it is not completely reducible and it has not* Δ*-pure literals. The first applied transformation is that given by the operator* `SubReduce`*, which outputs the following* Δ*-tree*



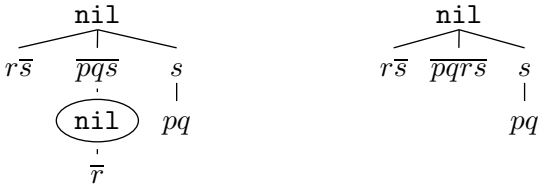
Rule C5 is applied to the circled node, obtaining:



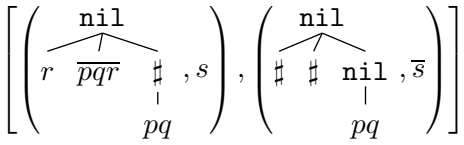
Now, Rule C4 is applied to the circled nodes, obtaining:



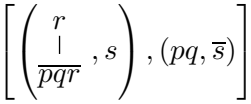
As this Δ-tree is neither completely reducible nor has Δ-pure literals, operator `SubReduce` *can be used to obtain the Δ-tree on the left and finally, rule C5 is applied once again on the circled node to get the Δ-tree on the right:*



Here we have to branch the Δ-tree. By choosing literal s we get the two following tasks



Updating the tasks we obtain



The second task allows us to deduce the satisfiability of the input formula, and any assignment I such that $I(p) = I(q) = I(\overline{s}) = 1$ is a model.

Table I. TAS vs Beatrix and Isabelle.

Prob.	Isab.	Bea.	TAS	Prob.	Isab.	Bea.	TAS
ex2	1.3	0.0	0.00	mul	130.9	0.2	0.07
transp	0.2	0.0	0.00	rip02	1.6	0.0	0.03
risc	9.8	0.6	0.05	rip04	994.5	0.5	0.38
counter	68.8	0.1	0.13	rip06	-	3.0	2.75
hostint1	96.5	0.2	0.10	rip08	-	18.2	17.18

7. Experimental results

We have written a straightforward implementation for the Macintosh port of the interpreter of Objective CAML (an ML-like functional language) in order to obtain a rapid prototype of a theorem prover. Δ-trees have been used to implement the reductions just described, together with a naive branching rule based on the Davis-Putnam procedure; namely, a formula A is split into two subformulas $A[p/\top]$ and $A[p/\bot]$, where p is the first variable occurring in A.

As our method is specially focused on non-cnf formulas we have run the prover, named TAS, on the IFIP benchmarks for hardware verification [3]. The results obtained, using a Power Macintosh G3 with 64 Mb of memory and 233 Mhz, are compared with those obtained in [6], for he also uses there a reduction-like strategy (which he calls *simplification*), in his experiments he used a Sun SuperSPARK. In Table I, we compare our implementation with the results obtained by Isabelle [8] (a well-known interactive prover, written in Standard ML) and Beatrix (a `sicstus Prolog` implementation in the spirit of lean tableau theorem proving). As several strategies were used in the cited work, in fairness to Isabelle and Beatrix, we compare our running time with their best absolute results no matter the strategy used.

It is important to remark that the results obtained are by far much better than those of Isabelle, showing that not only the scaling factor in problems such as `rip0n` can be reduced but also that absolute run time values are comparable to those obtained by Beatrix, which shortens the gap between lean theorem proving in Prolog and standard theorem proving in ML-like languages. In Table II some more results are com-

Table II. Run time (seconds) on other IFIP benchmarks.

Problem	Beatrix	TAS	Problem	Beatrix	TAS
d3 (satisf.)	0.1	0.17	vg2	7.0	2.82
misg	0.7	0.35	alu	7.1	3.98
ztwaalf1	0.8	0.80	x1dn	7.2	3.37
mp2d	1.1	1.03	z9sym	9.8	4.07
dk27	2.2	0.07	sqn	11.2	0.43
z4	2.3	1.53	add1	12.2	1.20
rom2	2.5	3.03	dc2	12.5	0.40
table	2.8	2.72	mul03	20.1	1.03
dk17	3.0	0.38	rd73	30.4	1.27
z5xpl	4.1	0.38	root	33.7	0.67
f51m	5.7	0.48	alupla20	618.1	31.72
pitch	5.7	2.55			

pared with the run time of Beatrix, where an important speed-up when using TAS can be noticed.

To make the comparison more interesting we also chose to run TAS on the Random 3-Sat benchmark, although TAS has not been neither designed nor optimised for cnf formulas. Table III shows the results for the standard random distribution of 3-SAT, where `3_sat(V,C)` means that samples had C clauses, with 3 literals selected uniformly among V variables and each literal negated with probability 0.5.

We show our results together with the results of two different flavours of Beatrix, the 'standard' one (in which the usual β-rule is used) and the 'lemmaizing' version (an asymmetric rule for a limited form of cut).

$$\frac{\mathcal{S}, \beta_1 \quad \mathcal{S}, \beta_2}{\mathcal{S}, \beta}\ \text{Std} \qquad\qquad \frac{\mathcal{S}, \beta_1 \quad \mathcal{S}, \overline{\beta_1}, \beta_2}{\mathcal{S}, \beta}\ \text{Lem}$$

One can easily see that, although our implementation has been run on a interpreter (as far as we know no compiler for CAML is still available for Macs) the performance of TAS is in between the two flavours of Beatrix.

Table III. TAS vs Beatrix on Random 3-SAT.

C/V	Problem	Beatrix	Bea-Lem	TAS
3	3_sat(32,96)	0.3	0.2	0.80
4	3_sat(32,128)	3.9	1.2	2.07
4.25	3_sat(32,136)	6.1	1.8	3.03
4.5	3_sat(32,144)	6.9	2.1	3.53
5	3_sat(32,160)	8.2	2.4	3.90
6	3_sat(32,192)	7.7	2.6	3.71
3	3_sat(64,192)	1.4	1.0	7.55
4	3_sat(64,256)	334.6	38.4	98.31
4.25	3_sat(64,272)	554.3	56.4	188.81
4.5	3_sat(64,288)	1,050.9	72.0	216.64
5	3_sat(64,320)	568.6	60.0	141.72
6	3_sat(64,384)	240.3	39.4	90.88

The speedup factor of TAS w.r.t. the standard version of Beatrix is about 2 for formulas with 32 variables and about 3.5 for formulas with 64 variables, whereas the better performance of the lemmaizing version of Beatrix averages 1.63 for 32 variables and 2.72 for 64 variables.

These results are neither surprising, for the standard version of Beatrix is just a tableau system improved with a particular case of our reductions, nor discouraging, for the branching rule we have implemented is just a raw DPLL-like procedure.

It is worth to note that, although the computational pay-off of the reductions implemented in TAS results in poor runtimes for the formulas in the first row of the table, the negative effect disappears as the size of the formulas is increased.

8. Conclusions

We have introduced Δ-trees for propositional formulas. This representation allows a compact representation for well-formed formulas as well as for a number of reduction strategies in order to consider only those occurrences of literals which are relevant for the satisfiability of the input formula. The reduction strategies have been implemented and tests are reported which show the relative good performance of our implementation of the techniques introduced.

References

1. G. Aguilera, I. P. de Guzmán, M. Ojeda-Aciego, and A. Valverde. Reductions for non-clausal theorem proving. *Theoretical Computer Science*, 2001. To appear. Available at `http://www.satd.uma.es/aciego/TR/tas-tcs.pdf`.
2. R.E. Bryant. Graph-based algorithms for Boolean function manipulation. *IEEE Transactions on Computers*, C-35(8):677–691, 1986.
3. L.J. Claesen, editor. *Formal VLSI correctness verification—VLSI design methods*, volume 2. Elsevier, 1990.
4. Jean H. Gallier. *Logic for Computer Science: Foundations for Automatic Theorem Proving*. Wiley & Sons, 1987.
5. G. Gutiérrez, I. P. de Guzmán, J. Martínez, M. Ojeda-Aciego, and A. Valverde. Reduction theorems for Boolean formulas using Δ-trees. In *Proc. of JELIA 2000*, pages 179–192. Lect. Notes in Artif. Intelligence 1919, 2000.
6. Fabio Massacci. Simplification: a general constraint propagation technique for propositional and modal tableaux. In *Proceedings of Tableaux'98*. Lect. Notes in Artificial Intelligence, 1998.
7. N.V. Murray and E. Rosenthal. Dissolution: Making paths vanish. *Journal of the ACM*, 40(3):504–535, 1993.
8. L.C. Paulson. Isabelle: a generic theorem prover. Lect. Notes in Comp. Sci. 828, 1994.
9. P. W. Purdom, Jr. Average time for the full pure literal rule. *is*, 78:269–291, 1994.
10. B. Yang, Y.A. Chen, R.E. Bryant, and D.R. O'Hallaron. Space- and time-efficient BDD construction via working set control. In *Proceedings of Asian-Pacific Design Automation Conference ASPDAC '98*, pages 423–432, 1998.