# Restricted $\Delta$-trees in Multiple-Valued Logics*

I.P. de Guzmán, M. Ojeda-Aciego, and A. Valverde

Dept. Matemática Aplicada
Universidad de Málaga
Bvd. Louis Pasteur s/n (Campus de Teatinos)
{guzman,aciego,a_valverde}@ctima.uma.es

**Abstract.** This paper generalises the tree-based data structure of $\Delta$-tree to be applied to signed propositional formulas. The $\Delta$-trees allow a compact representation for signed 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 conversions from signed formulas to $\Delta$-trees and vice versa are described and a notion of restricted form based on this representation is introduced, allowing for a compact representation of formulas in order to consider only those occurrences of literals which are relevant for its satisfiability.

**Keywords and topics:** AI foundations, reasoning models, uncertainty management, Automated Reasoning, Many-valued logics.

**Section:** Paper track

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

# Restricted $\Delta$-trees in Multiple-Valued Logics*

I.P. de Guzmán, M. Ojeda-Aciego, and A. Valverde

Dept. Matemática Aplicada
Universidad de Málaga
{guzman,aciego,a_valverde}@ctima.uma.es

**Abstract.** This paper generalises the tree-based data structure of $\Delta$-tree to be applied to signed propositional formulas. The $\Delta$-trees allow a compact representation for signed 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 conversions from signed formulas to $\Delta$-trees and vice versa are described and a notion of restricted form based on this representation is introduced, allowing for a compact representation of formulas in order to consider only those occurrences of literals which are relevant for its satisfiability.
**Keywords.** Automated Reasoning. Knowledge Representation

## 1 Introduction

Proof methods for multiple-valued logic have developed alongside the evolution of the notions of *sign* and *signed formula*. The use of signs and signed formulas allows one to apply classical methods in the analysis of multiple-valued logics. Forgetting the set of truth-values associated with a given logic, in the metalanguage one may interpret sentences about the multiple-valued logic as being true-or-false. For example, in a 3-valued logic with truth-values $\{0, 1/2, 1\}$ and with $\{1\}$ as the designated value, the satisfiability of a formula $\varphi$ can be expressed as: *Is it possible to evaluate $\varphi$ in $\{1\}$?* In the same way, the unsatisfiability of $\varphi$ is expressed by: *Is it possible to evaluate $\varphi$ in $\{1, 1/2\}$?* These questions can be represented by the signed formulas ${}_{\{1\}}{:}\varphi$ and ${}_{\{1/2,1\}}{:}\varphi$ which are evaluated on the set $\{0, 1\}$ with the following meaning:

${}_{\{1\}}{:}\varphi$ takes the value 1 iff $\varphi$ can be evaluated in $\{1\}$

${}_{\{1/2,1\}}{:}\varphi$ takes the value 1 iff $\varphi$ can be evaluated in $\{1/2, 1\}$

In other words, the formulas in a signed logic are constructions of the form $S{:}\varphi$, where $S$ is a set of truth-values of the multiple-valued logic, called the *sign*, and $\varphi$ is a formula of that logic. The interpretations that determine the semantics of the signed logic are defined from the interpretations of the multiple-valued logic as follows:

$$I_\sigma(S{:}\varphi) = 1 \quad \text{if and only if} \quad \sigma(\varphi) \in S$$

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

The first works to provide a systematic treatment of sets of truth-values as signs were due to Hähnle in [5] and Murray and Rosenthal in [7]. There the notion of *signed formula* is formally introduced. In [5] these tools are used in the framework of truth tables, while in [7] they are used to develop another, nonclausal proof method, that of *dissolution*. As a result of these works, the use of signed formulas in the field of automated deduction has been extended, and has lead to significant advances in this method; therefore, efficient representations for signed formulas are necessary in order to describe and implement efficient algorithms on this kind of formulas.

An approach to the efficient handling of signed formulas that one can find in the literature is the clause form [6], which allow the extension of classical techniques such as resolution, or Davis-Putnam procedures. Another representation is the Multiple-Valued Decision Diagrams (MDDs) and its variants [11, 2], but they are not useful for the study of satisfiability because although they make straightforward the testing of satisfiability, the construction of a restricted MDD for a given formula is exponential in the worst case. Some specific representation approaches exist for particular tasks, such as labelled rough partitions [3] to work with multiple-valued relations.

The approach we follow in this paper is that introduced in [4], interpreting signed formulas given means of $\Delta$-trees, that is, trees of clauses and cubes. We will be mainly concerned with the metatheory of multiple-valued $\Delta$-trees, not with implementation issues; however, the results obtained for the classical case are promising. It is interesting to recall the intrinsic parallelism between the usual representation of cnfs as lists of clauses and our representation of signed formulas as lists of $\Delta$-trees.

| | | |
|---|---|---|
| Clause | $\rightsquigarrow$ | List of literals |
| Cnf | $\rightsquigarrow$ | List of clauses |
| $\Delta$-tree | $\rightsquigarrow$ | Tree of clauses/cubes |
| signed formula | $\rightsquigarrow$ | List of $\Delta$-trees |

In this multiple-valued version, we will consider clauses and cubes with *basic* literals: signed literals with singleton signs.

## 2 Reduced signed logics

The notion of *reduced signed logic* was introduced in [8] as a generalisation of previous approaches. It is developed in the general framework of propositional logics, without reference either to an initially given multiple-valued logic or to a specific algorithm, ie. the definition is completely independent of the particular application at hand. The generalisation consists in introducing a *possible truth values function* to restrict the truth values for each variable. These restrictions can be motivated by the specific application and they can be managed dynamically by the algorithms. For example, in [8] these restrictions are used to improve the efficiency of tableaux methods; in [10] are used to characterize non-monotonic reasoning systems.

The formulas in the reduced signed logics are built by using the connectives $\wedge$ and $\vee$ on the atomic formulas. The atomic formulas are the *$\omega$-signed literals*: if $\mathbf{n} = \{1, \ldots, n\}$ is a finite set of truth-values,[1] $\mathcal{V}$ is the set of propositional variables and $\omega\colon \mathcal{V} \to (2^{\mathbf{n}} \setminus \varnothing)$ is a mapping, called the *possible truth-values function*, then the set of *$\omega$-signed literals* is

$$\textsc{lit}_\omega = \{S{:}p \mid S \subseteq \omega(p), p \in \mathcal{V}\} \cup \{\bot, \top\}$$

In a literal $\ell = S{:}p$, the set $S$ is called the *sign of* $\ell$ and $p$ is the *variable of* $\ell$. The opposite of a signed literal $S{:}p$ is $(\omega(p) \setminus S){:}p$ and will be denoted $\overline{S{:}p}$.

The semantics of $\mathbf{S}_\omega$, the *signed logic valued in* $\mathbf{n}$ *by* $\omega$, is defined using the *$\omega$-assignments*. The $\omega$-assignments are mappings from the language into the set $\{0, 1\}$ that interpret $\vee$ as maximum, $\wedge$ as minimum, $\bot$ as falsity, $\top$ as truth and have the following properties:

1. For every $p$ there exists a unique $j \in S$ such that $I(\{j\}{:}p) = 1$
2. $I(S{:}p) = 1$ if and only if there exists $j \in S$ such that $I(\{j\}{:}p) = 1$

These conditions arise from the objective for which signed logics were created: the $\omega$-assignment $I$ over $S{:}p$ is 1 if the variable $p$ is assigned a value in $S$; this value must be unique for every multiple-valued assignment and thus unique for every $\omega$-assignment. This is why we some times will write $I(_{\{j\}:p}) = 1$ as $I(p) = j$.

An important operation in the sequel will be the *reduction* of a signed logic. This operation decreases the possible truth-values set for one or more propositional variables. The reduction will be forced during the application of an algorithm but it can also help us to specify a problem using signed formulas. Specifically, we will use two basic reductions: to prohibit a specific value for a given variable, $[p \neq j]$, and to force a specific value for a given variable, $[p = j]$: If $\omega$ is a possible truth-values function, then the possible truth-values functions $\omega[p \neq j]$ and $\omega[p = j]$ are defined as follows:

$$\omega[p \neq j](v) = \begin{cases} \omega(p) \setminus \{j\} & \text{if } v = p \\ \omega(v) & \text{otherwise} \end{cases} \qquad \omega[p = j](v) = \begin{cases} \{j\} & \text{if } v = p \\ \omega(v) & \text{otherwise} \end{cases}$$

If $A$ is a formula in $\mathbf{S}_\omega$, we define the following substitutions:

- $A[p \neq j]$ is a formula in $\mathbf{S}_{\omega[p \neq j]}$ obtained from $A$ by replacing $_{\{j\}:p}$ by $\bot$, $\overline{_{\{j\}:p}}$ by $\top$ and $S{:}p$ by $(S \setminus \{j\}){:}p$. In addition, the constants are deleted using the 0-1-laws.
- $A[p = j]$ is a formula in $\mathbf{S}_{\omega[p=j]}$ obtained from $A$ by replacing every literal $S{:}p$ satisfying $j \in S$ by $\top$ and every literal $S{:}p$ satisfying $j \notin S$ by $\bot$; in addition, the constants are deleted using the 0-1-laws.

An immediate consequence is the following: if $I$ is a model of $A$ in $\mathbf{S}_\omega$ and $I(p) \neq j$, then (the restriction of) $I$ is also a model of $A[p \neq j]$ in $\mathbf{S}_{\omega[p \neq j]}$; if $I$ is a model of $A$ in $\mathbf{S}_\omega$ and $I(p) = j$, then $I$ is a model of $A[p = j]$ in $\mathbf{S}_{\omega[p=j]}$.

[1] The specific elements of $\mathbf{n}$ are not important, in the examples of this work we will use $\mathbf{n} = \{1, \ldots, n\}$ as set of truth values in a $n$-valued logic.

Throughout the rest of the paper, we will use the following standard definitions. A signed formula $A$ in $\mathbf{S}_\omega$ is said to be *satisfiable* if there is an $\omega$-assignment $I$ such that $I(A) = 1$; in this case $I$ is said to be a *model* for $A$. Two signed 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 $\omega$-assignment $I$. The symbols $\top$ and $\bot$ denote truth and falsity. We will also use the usual notions of clause (disjunction of literals) and cube (conjunction of literals). A literal $\ell$ is an *implicant* of a formula $A$ if $\ell \models A$. A literal $\ell$ 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 $\lambda$ and $\lambda'$ are lists, $\ell \in \lambda$ denotes that $\ell$ is an element of $\lambda$; the concatenation of two lists $\lambda$ and $\lambda'$ is written as either $\lambda\langle\rangle\lambda'$ or $\lambda \cup \lambda'$; the inclusion and intersection of lists are defined in the usual way.

## 3 Multiple-valued $\Delta$-trees

The satisfiability algorithm we will describe is based on the structure of multiple-valued $\Delta$-trees. In the classical case, nodes in the $\Delta$-trees correspond to lists of literals; in the multiple-valued case we will exploit a duality in the representation of signed literals in terms of basic literals (whose sign is a singleton). To better understand this duality, let us consider the literal $\{1,4\}{:}p$ in the signed logic $\mathbf{S}_\omega$ where $\omega(p) = \{1, 2, 4, 5\}$, then:

$$\{1,4\}{:}p \equiv \{1\}{:}p \vee \{4\}{:}p \qquad\qquad \{1,4\}{:}p \equiv \overline{\{2\}{:}p} \wedge \overline{\{5\}{:}p}$$

This way, we have both a disjunctive and a conjunctive representation of signed literals using the literals $\{j\}{:}p$ and $\overline{\{j\}{:}p}$, which are called *basic literals*. In the sequel, we will use a simpler representation for these literals:

$$pj \stackrel{def}{=} \{j\}{:}p \qquad\qquad \overline{pj} \stackrel{def}{=} \overline{\{j\}{:}p}$$

The basic literals $pj$ are the *positive literal* and their opposites, $\overline{pj}$, are the *negative literal*. In the $\Delta$-tree representation we work with lists of positive literals.

**Definition 1.**

1. *A list/set of positive literals, $\lambda$, is* saturated *for the variable $p$ if $pj \in \lambda$ for all $j \in \omega(p)$. (This kind of lists/sets will be interpreted as logical constants.)*
2. *A $\Delta$-*list *is either the symbol $\sharp$ or a list of positive literals such that it does not have repeated literals and it is non-saturated for any propositional variable.*
3. *A $\Delta$-*tree *$T$ is a tree with labels in the set of $\Delta$-lists.*

In order to define the operator `sgf` which interprets a $\Delta$-tree as a signed formula, we should keep in mind that:

1. The empty list, `nil`, has different conjunctive and disjunctive interpretations, since it is well-known the identification of the empty clause with $\bot$ and the empty cube with $\top$; but anyway it corresponds to the neutral element for the corresponding interpretation. Similarly, we will use a unique symbol, $\sharp$, to represent the absorbent elements, $\bot$ and $\top$, under the conjunctive and disjunctive interpretation, respectively.
2. A given $\Delta$-tree will always represent a conjunctive signed formula, however, its subtrees are alternatively interpreted as either conjunctive or disjunctive signed formulas, i.e. the immediate subtrees of a conjunctive $\Delta$-tree are disjunctive, and vice versa.

**Definition 2.** *The operator* `sgf` *over the set of $\Delta$-trees is defined as follows:*

1. $\mathtt{sgf}(\mathtt{nil}) = \top$, $\mathtt{sgf}(\sharp) = \bot$, $\mathtt{sgf}(\ell_1 \dots \ell_n) = \overline{\ell_1} \wedge \dots \wedge \overline{\ell_n}$
2. $\mathtt{sgf}\left(\begin{array}{c}\lambda\\ \diagup\ \ \diagdown\\ T_1\ \ \dots\ \ T_m\end{array}\right) = \mathtt{sgf}(\lambda) \wedge \mathtt{dsgf}(T_1) \wedge \dots \wedge \mathtt{dsgf}(T_m)$

*where the auxiliary operator* `dsgf` *is defined as follow:*

1. $\mathtt{dsgf}(\mathtt{nil}) = \bot$, $\mathtt{dsgf}(\sharp) = \top$, $\mathtt{dsgf}(\ell_1 \dots \ell_n) = \ell_1 \vee \dots \vee \ell_n$
2. $\mathtt{dsgf}\left(\begin{array}{c}\lambda\\ \diagup\ \ \diagdown\\ T_1\ \ \dots\ \ T_m\end{array}\right) = \mathtt{dsgf}(\lambda) \vee \mathtt{sgf}(T_1) \vee \dots \vee \mathtt{sgf}(T_m)$

*In short, we will write $\hat{T} = \mathtt{sgf}(T)$ and $\check{T} = \mathtt{dsgf}(T)$; in particular, if $T = \lambda = \ell_1 \dots \ell_n$ we have: $\hat{\lambda} = \ell_1 \wedge \dots \wedge \ell_n$ and $\check{\lambda} = \ell_1 \vee \dots \vee \ell_n$.*

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

In the next definition we introduce an operator to make the converse translation, that is, to define the $\Delta$-tree associated to a signed formula. To begin with, we will introduce the representation of clauses and cubes in terms of basic literals.

**Definition 3.**

1. *Given $A = S_1{:}p_1 \vee \dots \vee S_n{:}p_n$, consider the following set of positive literals:*

$$\mathcal{A} = \{ps \mid p = p_i \text{ for some } i \text{ and } s \in S_i\}$$

   *Then the $\Delta$-list $\mathtt{d}\Delta\mathtt{List}(A)$ is $\sharp$ if $\mathcal{A}$ is saturated for some $p_i$, otherwise it is the list of the elements of $\mathcal{A}$.*
2. *Given $A = S_1{:}p_1 \wedge \dots \wedge S_n{:}p_n$, consider the following set of positive literals:*

$$\mathcal{B} = \{ps \mid p = p_i \text{ for some } i \text{ and } s \in \omega(p) \smallsetminus S_i\}$$

   *Then the $\Delta$-list $\mathtt{c}\Delta\mathtt{List}(A)$ is $\sharp$ if $\mathcal{B}$ is saturated for some $p_i$, and it is the list of the elements of $\mathcal{B}$ otherwise.*

*Example 1.* In the logic $\mathbf{S}_\omega$ with $\omega(p) = \{1,2,4,5\}$, $\omega(q) = \{1,2,3\}$, $\omega(r) = \{2,5\}$.

- $\mathtt{d}\Delta\mathtt{List}(\{1,4\}{:}p \vee \{1,2\}{:}q) = p1\, p4\, q1\, q2$
- $\mathtt{c}\Delta\mathtt{List}(\{1,4\}{:}p \wedge \{1,2\}{:}q) = p2\, p5\, q3$
- $\mathtt{d}\Delta\mathtt{List}(\{1,4\}{:}p \vee \{2\}{:}r \vee \{2,4,5\}{:}p) = \sharp$, for $\{p1, p2, p4, p5, r2\}$ is saturated for $p$.
- $\mathtt{d}\Delta\mathtt{List}(\{1\}{:}q \wedge \{1,2,4\}{:}p \wedge \{2\}{:}q) = \sharp$, for $\{p5, q1, q2, q3\}$ is saturated for $q$.

In the following definition, we will work with lists of $\Delta$-trees. To help the reading, we will write these lists with the elements separated by commas and using square brackets as delimiters. This way, for example, $p_1 s_1 \dots p_n s_n$ is a $\Delta$-list, and $[p_1 s_1, \dots, p_n s_n]$ is a list of $\Delta$-trees (in which each $\Delta$-tree is a leaf, which turns out to be a singleton $\Delta$-list).

**Definition 4.** *Let $A$ be a signed formula, $\Delta\mathtt{Tree}(A)$ is a list of $\Delta$-trees defined recursively as follow:*

1. *If $A$ is a disjunctive signed formula, and the disjunction of its literals disjuncts is $A_0 = S_1{:}p_1 \vee \dots \vee S_k{:}p_k$, and $A_1, \dots, A_n$ are the non-literal disjuncts of $A$, then*
   (a) *If $k = 0$ then $\Delta\mathtt{Tree}(A) = [\mathtt{c}\Delta\mathtt{Tree}(A_1), \dots, \mathtt{c}\Delta\mathtt{Tree}(A_n)]$ (in this case, necessarily $n \neq 0$)*
   (b) *If $\mathtt{d}\Delta\mathtt{List}(A_0) = \sharp$, then $\Delta\mathtt{Tree}(A) = [\mathtt{nil}]$ (that is, a list with just one $\Delta$-tree, the leaf $\mathtt{nil}$.)*
   (c) $\Delta\mathtt{Tree}(A) = [\mathtt{c}\Delta\mathtt{List}(S_1{:}p_1), \dots, \mathtt{c}\Delta\mathtt{List}(S_k{:}p_k), \mathtt{c}\Delta\mathtt{Tree}(A_1), \dots, \mathtt{c}\Delta\mathtt{Tree}(A_n)]$ *otherwise*
2. *If $A$ is a conjunctive signed formula, then $\Delta\mathtt{Tree}(A) = [\mathtt{c}\Delta\mathtt{Tree}(A)]$ (that is, a list with just one $\Delta$-tree)*

*The auxiliary operators $\mathtt{d}\Delta\mathtt{Tree}$ and $\mathtt{c}\Delta\mathtt{Tree}$ are defined as follows:*

- *Let $A$ be a conjunctive signed formula, let $A_0 = S_1{:}p_1 \wedge \dots \wedge S_k{:}p_k$ be the conjunction of its literal conjuncts, and let $A_1, \dots, A_n$ be the non-literal conjuncts of $A$. If $\mathtt{c}\Delta\mathtt{List}(A_0) = \sharp$, then $\mathtt{c}\Delta\mathtt{Tree}(A) = \sharp$; if $\mathtt{c}\Delta\mathtt{List}(A_0) \neq \sharp$, then*
$$\mathtt{c}\Delta\mathtt{Tree}(A) = \overbrace{\mathtt{c}\Delta\mathtt{List}(A_0)}^{\mathtt{d}\Delta\mathtt{Tree}(A_1) \quad \dots \quad \mathtt{d}\Delta\mathtt{Tree}(A_n)}$$
- *Let $A$ be a disjunctive signed formula, let $A_0 = S_1{:}p_1 \vee \dots \vee S_k{:}p_k$ be the disjunction of its literals disjuncts, and let $A_1, \dots, A_n$ be the non-literal disjuncts of $A$. If $\mathtt{d}\Delta\mathtt{List}(A_0) = \sharp$, then $\mathtt{d}\Delta\mathtt{Tree}(A) = \sharp$; if $\mathtt{d}\Delta\mathtt{List}(A_0) \neq \sharp$, then*
$$\mathtt{d}\Delta\mathtt{Tree}(A) = \overbrace{\mathtt{d}\Delta\mathtt{List}(A_0)}^{\mathtt{c}\Delta\mathtt{Tree}(A_1) \quad \dots \quad \mathtt{c}\Delta\mathtt{Tree}(A_n)}$$

A $\Delta$-tree will always be interpreted as a conjunctive signed formula. To work with arbitrary signed formulas, we will use lists of $\Delta$-trees; this way, the study of satisfiability can be performed in parallel with the elements of the list.

*Example 2.* The following examples are from $\mathbf{S}_3$.

$$\Delta\texttt{Tree}(\texttt{(}\{1,2\}\texttt{:}p \vee \{2\}\texttt{:}q) \wedge (\{2,3\}\texttt{:}p \vee \{1,3\}\texttt{:}r) = \left[\begin{array}{c} \texttt{nil} \\ \diagup\quad\diagdown \\ p1p2q3 \quad p2p3r1r3 \end{array}\right]$$

$$\Delta\texttt{Tree}(\{2,3\}\texttt{:}q \vee (\{1,2\}\texttt{:}p \wedge (\{1,2\}\texttt{:}q \vee \{2,3\}\texttt{:}p) \wedge \{3\}\texttt{:}q \vee \{1\}\texttt{:}p))) = \left[q1, \begin{array}{c} p3 \\ \diagup\quad\diagdown \\ p2p3q1q2 \quad p1q3 \end{array}\right]$$

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

**Theorem 1.** *Let $A$ be a signed formula*

1. *If $A$ is disjunctive, then* $\texttt{dsgf}(\texttt{d}\Delta\texttt{Tree}(A)) \equiv A$
2. *If $A$ is conjunctive, then* $\texttt{sgf}(\texttt{c}\Delta\texttt{Tree}(A)) \equiv A$
3. *If* $\Delta\texttt{Tree}(A) = [T_1, \ldots, T_n]$, *then* $A \equiv \hat{T_1} \vee \cdots \vee \hat{T_n}$. *In particular, if* $n = 0$, *then* $A \equiv \bot$.

From this result we have that, in some sense, the structure of $\Delta$-tree allows to substitute reasoning with literals by reasoning with clauses and cubes. Other important consequence is that the structure of $\Delta$-tree gives us a means to calculate implicants and implicates, which will be used in the reduction transformations below.

**Proposition 1.** *If $T$ is rooted with $\lambda$ and $pj \in \lambda$, then:*

$$\texttt{sgf}(T) \models \overline{pj} \qquad \textit{and} \qquad pj \models \texttt{dsgf}(T)$$

## 4 Restricted $\Delta$-trees

In multiple-valued logic there is not a notion which captures the well-known definition of restricted clauses of classical logic, in which opposite literals and logical constants are not allowed. We can say that restricted $\Delta$-trees are $\Delta$-trees without *trivially* redundant information. The aim of this section is to give a suitable generalisation built on the notion of restricted multiple-valued $\Delta$-tree which is built from its classical counterpart [4].

To begin with, we need the technical definitions given below and in the subsequent sections:

**Definition 5.** *The operators* `Uni` *and* `Int` *are defined on the set of $\Delta$-lists as follows. If $\lambda_1, \ldots, \lambda_n$ are $\Delta$-lists then:*

1. $\texttt{Uni}(\lambda_1, \ldots, \lambda_n) = \sharp$ *if either there exists $i$ such that $\lambda_i = \sharp$ or $\bigcup_{i=1}^{n} \lambda_i$ is saturated for some variable $p$. Otherwise,* $\texttt{Uni}(\lambda_1, \ldots, \lambda_n) = \bigcup_{i=1}^{n} \lambda_i$.
2. $\texttt{Int}(\lambda_1, \ldots, \lambda_n) = \sharp$ *if $\lambda_i = \sharp$ for all $i$.*
   *Otherwise,* $\texttt{Int}(\lambda_1, \ldots, \lambda_n) = \bigcap_{\lambda_i \neq \sharp} \lambda_i$.

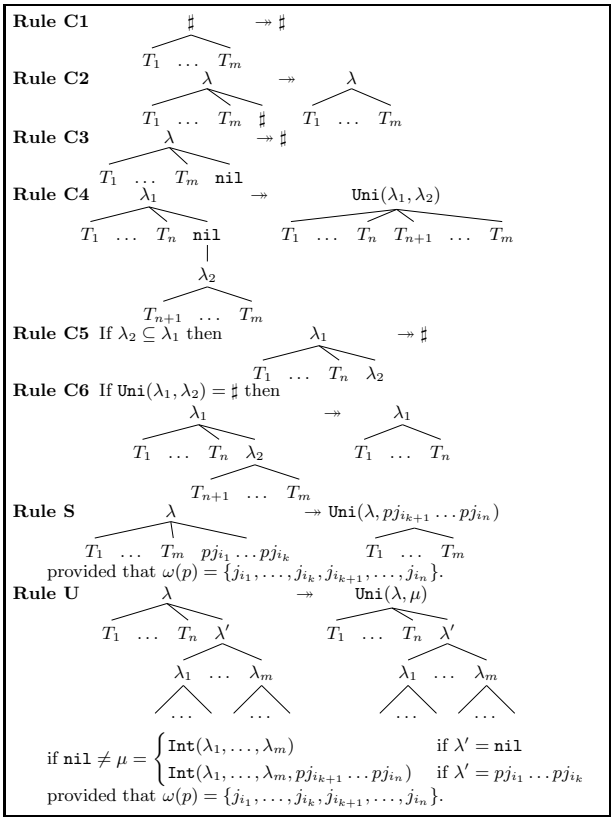



**Fig. 1.** Rewriting rules to obtain the restricted form

The following definition gathers the specific situations that will not be allowed in a restricted form: nodes in the $\Delta$-tree which, in some sense, can be substituted by either $\bot$ or $\top$ without affecting the meaning and leaves with only one propositional variable; in addition, our restricted trees must have explicitly the implicants and implicates of every subtree in order to perform the reductions based in these objects (see [9]).

**Definition 6.** *Let $T$ be a $\Delta$-tree.*

1. *A node of $T$ 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 $\lambda$, it has an immediate successor $\lambda'$ which is a leaf and $\lambda' \subseteq \lambda$.*

- *It is labelled with* $\lambda$ *and* $\texttt{Uni}(\lambda, \lambda') = \sharp$*, where* $\lambda'$ *is the label of its predecessor.*

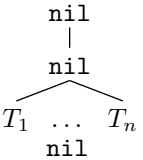
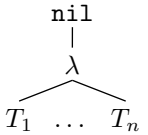
2. *A leaf in* $T$ *is said to be* simple *if the literals in its label share a common propositional variable.*
3. *Let* $\lambda$ *be the label of a node of* $T$*; let* $\lambda'$ *be the label of one immediate successor of* $\lambda$ *and let* $\lambda_1, \ldots, \lambda_n$ *be the labels of the immediate successors of* $\lambda'$*. We say that* $\lambda$ can be updated *if it satisfies some of the following conditions:*
   - $\lambda' = \texttt{nil}$ *and* $\texttt{Int}(\lambda_1, \ldots, \lambda_m) \not\subset \lambda$.
   - $\lambda' = pj_{i_1} \ldots pj_{i_k}$ *and* $\texttt{Int}(\lambda_1, \ldots, \lambda_m, pj_{i_{k+1}} \ldots pj_{i_n}) \not\subset \lambda$*, provided that* $\omega(p) = \{j_{i_1}, \ldots, j_{i_k}, j_{i_{k+1}}, \ldots, j_{i_n}\}$.

   *We say that* $T$ *is* updated *if it has no nodes that can be updated.*
4. *If* $T$ *is updated and it has neither conclusive nodes nor simple leaves, then it is said to be* restricted.

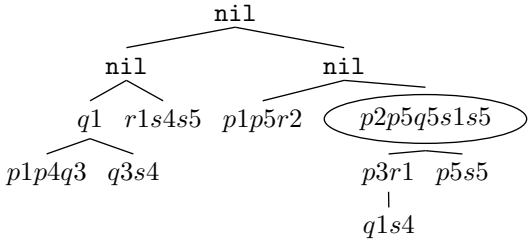
The rewriting rules (up to the order of the successors) in figure 1 allow to delete the conclusive nodes and simple leafes of a $\Delta$-tree and in addition, to update the updatable nodes. Note that the rewriting rules have a double meaning; since they need not apply to the root node, the interpretation can be either conjunctive or disjunctive. This is just another efficiency-related feature of $\Delta$-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 2.** *If* $T$ *is a* $\Delta$*-tree, there exists a list of restricted* $\Delta$*-trees,* $[T_1, \ldots, T_n]$*, such that* $\texttt{sgf}(T) \equiv \hat{T_1} \vee \cdots \vee \hat{T_n}$.

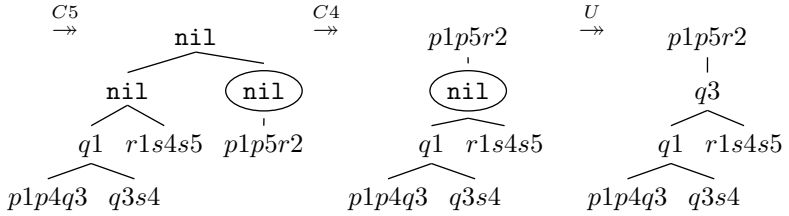
The proof of the theorem allows to specify a procedure to obtain $[T_1, \ldots, T_n]$. Let $T'$ be the $\Delta$-tree obtained from $T$ by exhaustively applying the rules C1, C2, C3, C4, C5, C6, S, and U till no one of them can be applied any more, then the list of restricted $\Delta$-trees $[T_1, \ldots, T_n]$, denoted by $\texttt{Restrict}(T)$, is defined as:

1. If $T' =$  then $\texttt{Restrict}(T) = [T_1, \ldots, T_n]$
2. If $T' =$  , and $\texttt{dsgf}(\lambda) = S_1{:}p_1 \vee \cdots \vee S_k{:}p_k$ with $p_i \neq p_j$ for every $i \neq j$, then $\texttt{Restrict}(T) = [\texttt{c}\Delta\texttt{List}(S_1{:}p_1), \ldots, \texttt{c}\Delta\texttt{List}(S_k{:}p_k), T_1, \ldots, T_n]$
3. Otherwise, $\texttt{Restrict}(T) = [T']$.

*Example 3.* Let us obtain the restricted form of the following $\Delta$-tree in $\mathbf{S}_\omega$ with $\omega(p) = \mathbf{5}$, $\omega(q) = \{1, 3, 5\}$, $\omega(r) = \{1, 2\}$, $\omega(s) = \{1, 4, 5\}$.




Rule C5 can be applied on the circled node because it contains its right successor: the subtree is substituted by ♯ and it is deleted by rule C2 obtaining the leftmost $\Delta$-tree in the figure below. The restricted form is obtained by applying rules C4 and U on the corresponding circled nodes.




## References


1. G. Aguilera, I. P. de Guzmán, M. Ojeda-Aciego, and A. Valverde. Reductions for non-clausal theorem proving. *Theoretical Computer Science*, 266(1), 2001.
2. C. Files, R. Drechsler, and M. Perkowski. Functional decomposition of MVL functions using multi-valued decision diagrams. In *Proc. ISMVL'97*, pp. 7–32. IEEE Press, 1997.
3. S. Grygiel and M. Perkowski, Labeled rough partitions—a new general purpose representation for multiple-valued functions and relations, *Journal of Systems Architecture*, 47(1):29–59, 2001.
4. G. Gutiérrez, I. P. de Guzmán, J. Martínez, M. Ojeda-Aciego, and A. Valverde. Satisfiability testing for Boolean formulas using $\Delta$-trees. *Studia Logica*, 2002. To appear.
5. R. Hähnle. Uniform notation of tableaux rules for multiple-valued logics. In *Proc. ISMVL'91*, pp. 238–245. IEEE Press, 1991.
6. R. Hähnle. Short conjunctive normal forms in finitely valued logics. *Journal of Logic and Computation*, 4(6):905–927, 1994.
7. N. V. Murray and E. Rosenthal. Improving tableau deductions in multiple-valued logics. In *Proc. ISMVL'91*, pp. 230–237. IEEE Press, 1991.
8. M. Ojeda-Aciego, I. P. de Guzmán, and A. Valverde. Multiple-Valued Tableaux with $\Delta$-reductions. In *Proc. of IC-AI'99*, pp. 177–183. C.S.R.E.A Press, 1999.
9. I. P. de Guzmán, M. Ojeda-Aciego, and A. Valverde. Reducing signed propositional formulas. *Soft Computing*, 2(4):157–166, 1999.
10. D. Pearce, I. P. de Guzmán, and A. Valverde. *Computing Equilibrium Models using Signed Formulas*. In *Proc. of CL'2000*, London, UK, 2000. Springer-Verlag.
11. A. Srinivasan, T. Karn, S. Malik, and R. Brayton. Algorithms for discrete function manipulation. In *Proc. Intl. Conf. on CAD*, pages 92–95, 1990.