Acta Cybernetica **17** (2005) 325–338.

HyperS Tableaux – Heuristic Hyper Tableaux

Gergely Kovásznai*

Abstract

Several syntactic methods have been constructed to automate theorem proving in first-order logic. The positive (negative) hyper-resolution and the clause tableaux were combined in a single calculus called hyper tableaux in [1]. In this paper we propose a new calculus called hyperS tableaux which overcomes substantial drawbacks of hyper tableaux. Contrast to hyper tableaux, hyperS tableaux are entirely automated and heuristic. We prove the soundness and the completeness of hyperS tableaux. HyperS tableaux are applied in the theorem prover Sofia, which additionally provides useful tools for clause set generation (based on justificational tableaux) and for tableau simplification (based on redundancy), and advantageous heuristics as well. An additional feature is the support of the so-called parametrized theorems, which makes the prover able to give compound answers.

1 Introduction

Several syntactic methods have been constructed and implemented to automate theorem proving in first-order logic. Most of these methods can be classified as either tableau-based or resolution-based. Semantic tableaux were introduced by [8] and meet the problem of term selection for instantiating γ-formulas, which was tried to be overridden by the application of the most general unifier atomic closure rule for free-variable tableaux in [2]. Resolution acts with clauses. An easy-to-implement variant is SLD-resolution on which the programming language Prolog is based. SLD-resolution restricted to the class of Horn clauses is complete. One improved variant is the positive (negative) hyper-resolution which is not restricted and resolves the entire clause head (body) in a single inference step [7]. Hyper tableaux by [1] combine the advantageous features of positive hyper-resolution and clause tableaux [3], but admit some unwanted solutions which were tried to be eliminated in [4].

In this paper, we propose an improved and heuristic variant of rigid hyper tableaux calculus defined in [4], what we call *hyperS tableaux*. The name comes from the theorem prover *Sofia* in which the proposed calculus is applied and refers to that more than one clauses can be instantiated in a single inference step. After

*Department of Computer Science, University of Debrecen, Hungary. Email: kovasz@inf.unideb.hu

introducing the necessary concepts in Section 2, we will give details of hyperS tableaux in Section 3, where the soundness and completeness of the calculus will be proven. In Section 4.1, an easy-to-use and effective tableau-based method is introduced for generating clauses. In Section 4.2, the heuristical management of hyperS tableaux is argued in order to reduce the size of the tableau constructed. In Section 5, this issue will be further analyzed by recommending some simplifications in the tableau. These simplifications are related to the concept of redundancy [1]. Last, we mention the support for parametrized theorems, which is considered in Section 4.3.

2 Preliminaries

In the followings, we assume the reader to be familiar with the basic concepts of first-order logic.

Definition 1 (Equivalence). Two formulas A and B are *logically equivalent*, denoted by $A \equiv B$, iff in any model A is true iff B is true.

Definition 2 (Literal). A formula L is a *literal* iff $L = A$ or $L = \neg A$ where A is atomic.

Definition 3 (Positive/negative literal). Let L be a literal.

(1) If L is atomic and $L \notin \{\top, \bot\}$ then L is positive.

(2) If $L = \neg A$ where A is atomic, L is positive iff A is negative.

We have given an inductive definition of positive/negative literals. The definition is incomplete since we have not defined if $\top$ and $\bot$ are positive or negative. This must also be defined for a complete definition, and can be defined on demand, as it will be in Section 3 in connection with positive and negative hyper tableaux.

Definition 4 (Complement). A literal $\overline{L}$ is a *complement* of a literal L if

- $\overline{L} = \top$ if $L = \bot$, or
- $\overline{L} = \bot$ if $L = \top$, or
- $\overline{L} = \neg A$ if $L = A$, or
- $\overline{L} = A$ if $L = \neg A$,

where A is atomic.

Definition 5 (Clause). A *clause* is a set $C = \{L_1, \ldots, L_n\}$ where $n \geq 1$ and L_i $(1 \leq i \leq n)$ is a *literal*. C can be regarded as a disjunction of its literals, namely $L_1 \vee \cdots \vee L_n$. Let $\{\bot\}$ be the *empty clause*, and be denoted by $\square$.

In the literature, a tableau is usually defined as a labelled tree. However, a tableau is commonly regarded, and used, as a set of its branches, which are defined as sets of their formulas.

Definition 6 (Branch). A *branch* is a multiset $\mathcal{B} = \{L_1, \ldots, L_n\}$ where $n \geq 1$ and L_i $(1 \leq i \leq n)$ is a literal. $\mathcal{B}$ can be regarded as a conjunction of its literals, namely $L_1 \wedge \cdots \wedge L_n$.

Definition 7 (Tableau). A *tableau* is a multiset $\mathcal{T} = \{\mathcal{B}_1, \ldots, \mathcal{B}_n\}$ where $n \geq 1$ and $\mathcal{B}_i$ $(1 \leq i \leq n)$ is a branch.

The instantiation of a clause C in a tableau means that for some substitution σ (the literals of) $C\sigma$ is being attached to a branch of the tableau. We require to classify each parameter of a clause either as a *parametric variable* or as a *universal variable* (see Section 4.3).

Definition 8 (Brand new clause instance). A *new instance* of a clause C is $C\sigma$ where σ is a renaming substitution such that $Dom(\sigma)$ contains all universal variables in C, as defined in [3]. By a *brand new instance* of C, we mean a new instance $C\sigma$ where $Range(\sigma)$ consists of only "brand new" variables, i.e., variables that have not occured in the given derivation yet (neither in the clause set being refuted nor in the tableau being constructed). We say that a variable is *rigid* iff it was introduced by a brand new clause instance.

Definition 9 (Renamed variants). Two formula F_1 and F_2 are *renamed variants* iff there is a renaming substitution σ such that $F_1 = F_2\sigma$.

The concepts "unifiable" and "unifier" are well-known. We define the following concept:

Definition 10 (Complementary unifier). Two literals L_1 and L_2 are *complementary unifiable* iff there is a substitution σ such that $L_1\sigma$ is a complement of $L_2\sigma$, and σ is called a *complementary unifier* of L_1 and L_2.

3 HyperS Tableaux

Let $\mathcal{C}$ be a clause set. We assume that no $C \in \mathcal{C}$ contains any parametric variable.[1] The effects and advantage of eliminating this restriction will be explored in Section 4.3.

Definition 11 (Extension). An *extension* is a tuple (E, θ) where E is a clause and θ is a substitution.

Definition 12 (Extension application). Applying extension (E, θ) to a branch $\mathcal{B}$ in a tableau $\mathcal{T}$ means executing the following steps:

(1) $\mathcal{T} := \mathcal{T} \backslash \mathcal{B} \cup \{\mathcal{B} \cup \{L\} \mid L \in E\}$;

(2) $\mathcal{T} := \mathcal{T}\theta$.

[1] By this we refer to closed clauses, as it is known in the literature.

The following definitions (Definition 13, Definition 14, and Definition 15) and theorem (Theorem 19) concern *positive hyperS tableaux*. For negative hyperS tableaux, we can easily formulate the dual forms of these definitions and theorem only by switching all the adjectives "positive" to "negative", and vice versa. This is why we postponed the completion of Definition 3 to this section, which is done by the following definition.

Definition 13. Let $\top$ be negative, and $\bot$ be positive.

The following two definitions are for computing the possible extensions for a branch and a clause set.

Definition 14. Let C be a clause and let $\{L_1^{\ominus}, \ldots, L_n^{\ominus}\}$ $(n \geq 0)$ be the set of all negative literals in C. Furthermore, let $\{C_1, \ldots, C_n\}$ be a clause multiset and let $\{L_1^{\oplus}, \ldots, L_n^{\oplus}\}$ be a multiset of positive literals such that $L_i^{\oplus} \in C_i$ $(1 \leq i \leq n)$. The extension $\mathcal{E}$ for C, $\{C_1, \ldots, C_n\}$, and $\{L_1^{\oplus}, \ldots, L_n^{\oplus}\}$ is computed by executing the following steps:

(1) $E := \Box$ and $\theta := \epsilon$;

(2) for all i $(1 \leq i \leq n)$:

  (a) let σ be the most general complementary unifier of $L_i^{\ominus}\theta$ and $L_i^{\oplus}\theta$;

  (b) if σ does not exist then $\mathcal{E}$ does not exist (halt);

  (c) $E := E \cup C_i'$ where $C_i' = C_i \backslash \{L_i^{\oplus}\}$;

  (d) $\theta := \theta\sigma$;

(3) $E := E \cup C'$ where $C' = C \backslash \{L_1^{\ominus}, \ldots, L_n^{\ominus}\}$;

(4) $\mathcal{E}$ exists and is (E, θ).

Definition 15 (Extension for branch and clause set). An extension for a branch $\mathcal{B}$ and the clause set $\mathcal{C}$ is an extension for

(1) any $\underline{C}$ where $C \in \mathcal{B}' \cup \mathcal{C}$ and $\mathcal{B}'$ is the set of the negative literals in $\mathcal{B}$,

(2) any multiset $\{\underline{C}_1, \ldots, \underline{C}_n\}$ where n is the number of negative literals in C and $C_i \in \mathcal{B} \cup \mathcal{C}$ $(1 \leq i \leq n)$, and

(3) any multiset $\{L_1, \ldots, L_n\}$ where L_i is a positive literal in $\underline{C}_i$ $(1 \leq i \leq n)$,

where for any clause D

$$\underline{D} = \left\{ \begin{array}{ll} \text{a brand new instance of } D & \text{, if } D \in \mathcal{C} \\ D & \text{, otherwise} \end{array} \right\}$$

Definition 16 (HyperS tableaux). For a clause set $\mathcal{C}$, a *hyperS tableau* is constructed as follows:

(1) (Initialization rule) A tableau consisting of a single branch $\{\top\}$ is a hyperS tableau for $\mathcal{C}$.

(2) (Extension rule) Let $\mathcal{T}$ be a hyperS tableau for $\mathcal{C}$, let the branch $\mathcal{B} \in \mathcal{T}$, and let $\mathcal{E}$ be an extension for $\mathcal{B}$ and $\mathcal{C}$. The tableau constructed by the application of $\mathcal{E}$ to $\mathcal{B}$ in $\mathcal{T}$ is a hyperS tableau for $\mathcal{C}$.

For proving hyperS tableaux to be sound and complete, we will use the soundness and completeness of *clause tableaux* introduced in [3].

Definition 17 (Clause tableaux). For the clause set $\mathcal{C}$, a clause tableau is constructed with the following rules:

(1) (Initialization rule) A tableau consisting of a single branch $\{\top\}$ is a clause tableau for $\mathcal{C}$.

(2) (Extension rule) Let $\mathcal{T}$ be a clause tableau for $\mathcal{C}$, let $\mathcal{B} \in \mathcal{T}$, and let $C \in \mathcal{C}$. Let $\{\underline{L}_1, \ldots, \underline{L}_n\}$ be a brand new instance of C. $\mathcal{T} \backslash \mathcal{B} \cup \big\{\mathcal{B} \cup \{\underline{L}_i\} \mid 1 \leq i \leq n\big\}$ is a clause tableau for $\mathcal{C}$.

(3) (Closure rule) Let $\mathcal{T}$ be a clause tableau for $\mathcal{C}$, let $\mathcal{B} \in \mathcal{T}$, and let the literals $L_1, L_2 \in \mathcal{B}$. If L_1 and L_2 are complementarily unifiable with the most general unifier σ, then $\mathcal{T}\sigma$ is a clause tableau for $\mathcal{C}$.

Lemma 18. *Clause tableaux are sound and complete as proven in [3].*

Theorem 19 (Soundness and completeness). *HyperS tableaux are sound and complete.*

Proof. It is sufficient to prove that the application of each rule of clause tableaux can be simulated by the application of the rules of hyperS tableaux, and vice versa. In the proof, we do not distinguish two tableaux if they contain some closed branches additionally as compared to each other. Such a distinction is superfluous in terms of derivation.

The initialization rule of the two calculi is the same. For the rest:

(I) For clause tableaux (notation is from Definition 17):

  (1) (Extension rule) Let C' denote $C \cup \{\bot\}$. Notice that $\top \in \mathcal{B}$ and $C' \equiv C$. Consider the extension for $\{\top\}$, $\{\underline{C}'\}$, and $\{\bot\}$, namely $(\underline{C}', \epsilon)$ where $\underline{C}'$ is a brand new instance of C'.

  (2) (Closure rule) Assume that L_1 is negative. Consider the extension for $\{L_1\}$, $\big\{\{L_2\}\big\}$, and $\{L_2\}$, which is actually $(\square, \sigma)$.

(II) For hyperS tableaux, the proof for the extension rule follows (notation is from Definition 14 and Definition 16).

  Apply the extension rule of clause tableaux for $\mathcal{T}$ and for

(1) $\mathcal{B}$ and C_1;

(2) $\mathcal{B} \cup \{L_1^{\oplus}\}$ and C_2;

(3) $\mathcal{B} \cup \{L_1^{\oplus}, L_2^{\oplus}\}$ and C_3;

$\vdots$

(n) $\mathcal{B} \cup \{L_1^{\oplus}, \ldots, L_{n-1}^{\oplus}\}$ and C_n;

(n+1) $\mathcal{B} \cup \{L_1^{\oplus}, \ldots, L_n^{\oplus}\}$ and C

one after the other.

Then for each i $(1 \leq i \leq n)$, consider the branch containing $L_i^{\oplus}$ and $L_i^{\ominus}$, to which the closure rule is applied, i.e., σ_i is applied to the tableau where σ_i is the most general complementary unifier of $L_i^{\oplus}$ and $L_i^{\ominus}$. Notice that $\theta = \sigma_1 \sigma_2 \ldots \sigma_n$.

□

As usual, a branch $\mathcal{B}$ is said to be closed if $\mathcal{B}$ contains complementary literals. In hyperS tableaux, it would be sufficient to define the *closeness* of $\mathcal{B}$ as follows: $\mathcal{B}$ contains both $\top$ and $\bot$. What is more, since a branch always contains $\top$, it would be completely sufficient to monitor whether $\bot$ has occurred in $\mathcal{B}$. This latter method is very similar to that is used in resolution, namely monitoring whether $\square$ has occurred.

Compared to hyper tableaux, hyperS tableaux do not require "*purifying substitutions*", which moreover were generated by guessing in [1]. This is handled by using rigid variables, similarly to [4]. However, we have made improvements to the method written there in order to avoid other unwanted solutions like "clause copies" and "factoring". Three evident improvements, as compared to [4], have been made as follows.

In Definition 14, $\{C_1, \ldots, C_n\}$ is a multiset, i.e., a clause may occur more than once. This means that a clause can be instantiated more than once during a single extension application. This is why "*clause copies*" are not needed in hyperS tableaux. For example, consider the clause set consisting of $E_1 = \{P(x), Q(x)\}$ and $E_2 = \{\neg P(a()), \neg P(f(x)), R(y)\}$. E_1 should be instantiated twice in order to instantiate E_2 (see [4] for details). In hyperS tableaux, an extension can be constructed for E_2, $\{E_1, E_1\}$, and $\{P(x), P(x)\}$ (we are passing over brand new instances this time).

In Definition 15, (1), C can be chosen from $\mathcal{B}' \cup \mathcal{C}$, i.e., not only from the clause set, but also from the branch. This results in the elimination of "*factoring.*" For example, consider the clause set consisting of $E_1 = \{P(x), P(y)\}$ and $E_2 = \{\neg P(z)\}$. The tableau, after the instantiation of E_1, should be factored [5], otherwise an infinite tableau will be constructed. In hyperS tableaux, this is avoided by applying both the extension for E_2, $\{E_1\}$, and $\{P(x)\}$, and the extension for E_2, $\{P(y)\}$, and $\{P(y)\}$ (we are passing over brand new instances again). We want to emphasize that although we have eliminated factoring, it could be performed as a

simplification (i.e., tableau reduction) on demand. In Section 5, we will propose other simplifications in connection with redundancy.

In Definition 15, (2), C_i $(1 \leq i \leq n)$ can be from $\mathcal{B} \cup \mathcal{C}$, i.e., not only from the branch, but also from the clause set. Such a permissiveness was motivated by the observation that [1] and [4] took into consideration only the "past" (i.e., what literals have been attached to the branch), but the "future" (i.e., what literals may be attached to the branch further on) not at all. By such a *lookahead*, we intend to reduce the size of the constructed tableau (and so the execution time). This is one of the reasons why we use *heuristics* in ranking extensions for the branch and the clause set (see Section 4.2).

4 Theorem Prover Sofia

The theorem prover Sofia applies hyperS tableaux for theorem proving. Besides the use of the calculus, Sofia provides automated clause set generation (Section 4.1) and the heuristic management of hyperS tableaux (Section 4.2), and can give compound answers by using parametric variables (Section 4.3).

4.1 Clause Set Generation by Justificational Tableaux

In this section, we propose an easy-to-use and effective method for generating a clause set for a formula, i.e., a clause set which is satisfiable iff the given formula is satisfiable. The method is based on the so-called justificational tableaux. Justificational tableaux are similar to refutational tableaux except for the nature of the questions that can be answered. While refutational tableaux are for investigating whether a formula is unsatisfiable, justificational tableaux can answer whether a formula is valid. This difference manifests in the form of the tableau expansion rules. Justificational tableau expansion rules are the ones in Figure 1, where we use α, β, γ, and δ formulas as introduced in the unifying notation by [8].

$$\frac{\alpha}{\alpha_1 \mid \alpha_2} \qquad \frac{\beta}{\begin{array}{l}\beta_1\\ \beta_2\end{array}} \qquad \frac{\gamma}{\gamma(x)} \quad x \text{ is a new parameter} \qquad \frac{\delta}{\delta(f(x_1,\ldots,x_n))} \quad \begin{array}{l}f \text{ is a new function symbol}\\ \text{and } FV(\delta) = \{x_1,\ldots,x_n\}\end{array}$$

Figure 1: Justificational Tableau Expansion Rules

Every branch of a justificational tableau can be regarded as a disjunction of formulas, and a justificational tableau is a conjunction of its branches. The closeness of a branch means that the branch, as a disjunction, is valid, and the closeness of a tableau corresponds to the fact that the tableau, as a conjunction, is valid.

Justificational tableau expansion rules for α, β, and γ formulas are based on the same facts as refutational tableau expansion rules. Namely: in any model

(1) α is true iff both α_1 and α_2 are true;

(2) β is true iff any of β_1 and β_2 is true;

(3) γ is true iff $\gamma(x)$ is true.

The justification tableau expansion rule for δ formulas is based on the following fact: for a formula F, F is satisfiable iff F^{SK} is satisfiable where F^{SK} is a Skolem formula for F [9]. Our treatment of δ formulas results in performing skolemization "on the fly", i.e., there is no need to skolemize F before constructing a tableau for F.

We consider a tableau finished iff each of its formulas either is a literal or has already been used for applying a tableau expansion rule. Since we want to use justificational tableaux only for clause set generation, we do not allow reusing any formulas[2]. Consequently, a finished justificational tableau for a formula can be constructed in finitely many steps.

Note that if only literals are considered then a finished justificational tableau for a formula F is a clause set for F. Hence, it is reasonable to let Sofia operate with two tableaux: one finished justificational tableau $\mathcal{T}$ for F and one hyperS tableau for (the set of the branches of) $\mathcal{T}$. In Figure 2, a clause set $\mathcal{C}$ is generated for the formula set of problem No. 31 in [6], as an example.

4.2 Heuristics

In this section, we argue what heuristic is worth to use for ranking the extensions for a branch and a clause set. Let $\mathcal{E} = (E, \theta)$ be an extension for a branch $\mathcal{B}$ of a tableau $\mathcal{T}$ and for a clause set $\mathcal{C}$. The heuristic is reckoned as a total order over the set of extensions. When defining the heuristic, we primarily consider the number of new branches in $\mathcal{T}$ after applying $\mathcal{E}$ to $\mathcal{B}$, which is exactly $|E|$. We secondarily consider the number of rigid variables substituted by the application of $\mathcal{E}$.

Definition 20 (Heuristic). For two extensions (E_1, θ_1) and (E_2, θ_2), $(E_1, \theta_1) \leq (E_2, \theta_2)$ iff

(1) $|E_1| < |E_2|$ or

(2) $|E_1| = |E_2|$ and $|R_1| \leq |R_2|$
where $R_i = \{x | x \in Dom(\theta_i) \cap FV(\mathcal{T})\}$, $i \in \{1, 2\}$.

Sofia applies the minimal extension to $\mathcal{B}$ in $\mathcal{T}$. Of course, the total order over the set of extensions can be defined differently.

In Figure 3, a derivation for the clause set $\mathcal{C} = \{C_1, \ldots, C_6\}$ in Figure 2 can be seen. The derivation consists of three hyperS tableaux shown on the left in Figure 3. On the right, the extensions generated for the single open branch of the given tableau and $\mathcal{C}$ are enumerated, ordered by the proposed heuristic. To the (open branch of the) first tableau $\mathcal{E}_1$ is applied. To the (open branch of the) second tableau $\mathcal{E}_3$ is applied, which only appends $\bot$ to the open branch, thus a closed tableau gets constructed. Hence, $\mathcal{C}$ is unsatisfiable.

[2] As it is known, reusing γ formulas must be allowed for a complete proof procedure [2].

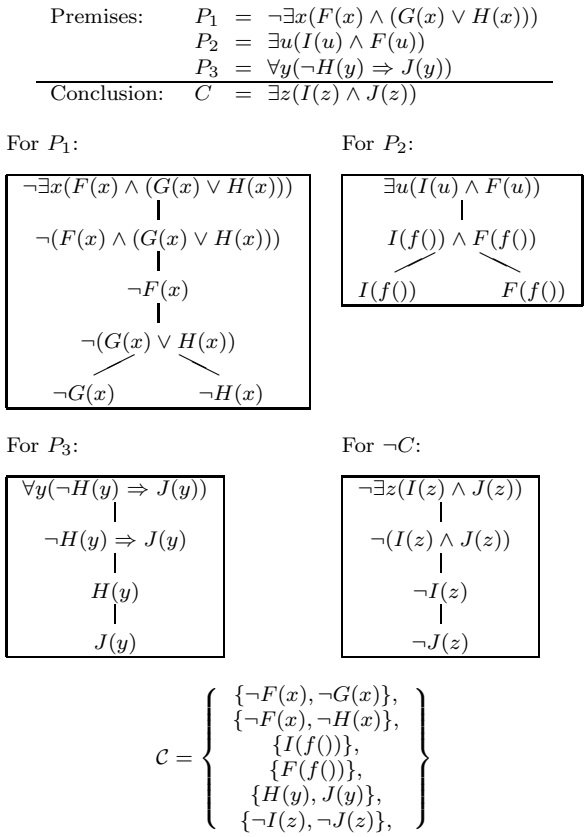

Figure 2: Generating a clause set.

4.3 Parametrized Theorems

As mentioned in Section 3, we remove the restriction on the closeness of clauses in a clause set $\mathcal{C}$ being refuted. Thus we let any clause in $\mathcal{C}$ contain parametric variables. For a user who wants the theorem prover to answer on the theoremhood of a formula F, the facility of permitting parametric variables in F provides exciting opportunities. The prover can give not only a yes/no answer but can tell the substitution θ to the parametric variables which makes $F\theta$ a theorem. Notice that F being a theorem is a special case of $F\theta$ being a theorem, when $\theta = \epsilon$.

As it is well-known, SLD-resolution (and hence, Prolog) allows the use of parametric variables. Actually, Prolog does not distinguish universal and parametric variables. The effect of this fact is the need of backtracking. The hyper tableaux calculus (and hence, the hyperS tableaux calculus) is a straightforward method (see [1]) since it prohibits the use of parametric variables. The explanation for this is that a substitution to (a rigid variable substituted to) a universal variable does

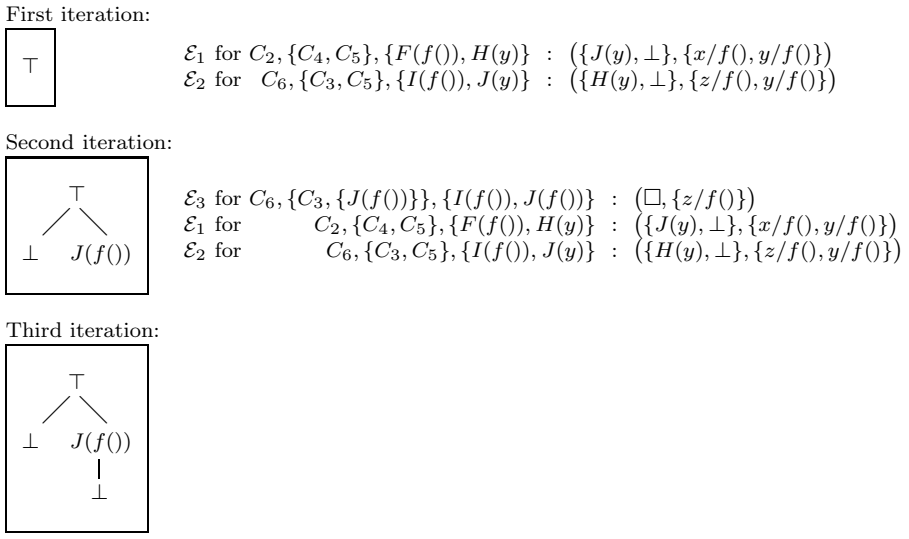

Figure 3: HyperS tableaux derivation.

not exclude the other substitutions to (a rigid variable substituted to) the same universal variable to be applied later, since new instances of the clauses in $\mathcal{C}$ can be attached to the tableau several times. Contrarily, a substitution to a parametric variable is final.

On the basis of the preceding discussion, the use of parametric variables in hyperS tableaux requires "rollback points" to be declared in a derivation whenever a parametric variable gets substituted, and to try to construct another derivation starting from the latter "rollback point" whenever a derivation ends in an open tableau. Thus, Sofia becomes a compromised solution: it substitutes universal variables in a straightforward way and it supports the use of parametric variables by backtracking.

5 Tableau Simplifications Based on Redundancy

In this section, we propose simplifications (or reductions) of the tableau being constructed, based on redundancy. [1] introduced the redundancy of a clause in a branch. We define the concept of redundancy by the following definition:

Definition 21 (Redundancy). A formula A is redundant in a formula B w.r.t. a logical connective $\circ$ iff $A \circ B \equiv B$.

In Section 5.1, we define the redundancy of an extension in a branch, based on [1] and [4], and then a simplification called the Redundancy Check in order to avoid repetitions along a branch.

In Section 5.2, we propose two other simplifications based on the redundancy of clauses.

5.1 Redundancy Check

Definition 22 (Redundancy in conjunction). A literal L is redundant in a conjunction of the literals $L_1, \ldots, L_n$ iff for some i $(1 \leq i \leq n)$ there is a substitution σ such that $L = L_i\sigma$.

Theorem 23. *If a literal L is redundant in a conjunction of the literals $L_1, \ldots, L_n$ then L is redundant in $L_1 \wedge \cdots \wedge L_n$ w.r.t. $\wedge$.*

Proof. By Definition 22, L is redundant in a conjunction of the literals $L_1, \ldots, L_n$ iff for some i $(1 \leq i \leq n)$ there is a substitution σ such that $L = L_i\sigma$.

By Definition 21, L is redundant in $L_1 \wedge \cdots \wedge L_n$ w.r.t. $\wedge$ iff $L_1 \wedge \cdots \wedge L_n$ and $L_1 \wedge \cdots \wedge L_n \wedge L$ are logically equivalent, i.e., in any model $L_1 \wedge \cdots \wedge L_n$ is true iff $L_1 \wedge \cdots \wedge L_n \wedge L$ is true. This latter equivalence must be proven.

Right-to-left is evident. For the reverse: in a model, $L_1 \wedge \cdots \wedge L_n$ is true iff L_j is true for all j $(1 \leq j \leq n)$. Hence, L_i is true. Since $L = L_i\sigma$, L is true. Hence, $L_1 \wedge \cdots \wedge L_n \wedge L$ is true. □

Definition 24 (Redundancy of clause in branch). A clause $\{L_1, \ldots, L_n\}$ is redundant in a branch $\mathcal{B}$ iff for some i $(1 \leq i \leq n)$ L_i is redundant in $\mathcal{B}$ as a conjunction.

Definition 25 (Redundancy of extension). An extension (E, θ) is redundant in a branch $\mathcal{B}$ iff $E\theta$ is redundant in $\mathcal{B}\theta$.

By using the redundancy of an extension, a simplification called the *Redundancy Check* can be introduced: do not apply an extension $\mathcal{E}$ to a branch B if $\mathcal{E}$ is redundant in $\mathcal{B}$.

5.2 Clause Simplifications

Definition 26 (Redundancy in disjunction). A literal L is redundant in a disjunction of the literals $L_1, \ldots, L_n$ iff for some i $(1 \leq i \leq n)$ there is a substitution σ such that $L\sigma = L_i$.

Theorem 27. *If a literal L is redundant in a disjunction of the literals $L_1, \ldots, L_n$ then L is redundant in $L_1 \vee \cdots \vee L_n$ w.r.t. $\vee$.*

Proof. By Definition 26, L is redundant in a disjunction of the literals $L_1, \ldots, L_n$ iff for some i $(1 \leq i \leq n)$ there is a substitution σ such that $L\sigma = L_i$.

By Definition 21, L is redundant in $L_1 \vee \cdots \vee L_n$ w.r.t. $\vee$ iff $L_1 \vee \cdots \vee L_n$ and $L_1 \vee \cdots \vee L_n \vee L$ are logically equivalent, i.e., in any model $L_1 \vee \cdots \vee L_n$ is false iff $L_1 \vee \cdots \vee L_n \vee L$ is false. This latter equivalence must be proven.

Right-to-left is evident. For the reverse: in a model, $L_1 \vee \cdots \vee L_n$ is false iff L_j is false for all j $(1 \leq j \leq n)$. Hence, L_i is false. Since $L_i = L\sigma$, L is false. Hence, $L_1 \vee \cdots \vee L_n \vee L$ is false. □

Definition 28 (Redundant clause). A clause C is redundant iff L is redundant in $C \backslash L$ as a disjunction for some $L \in C$.

The next definition and theorem are for simplifying a redundant clause to a non-redundant one.

Definition 29 (Non-redundant variant). A clause C' is a non-redundant variant of a clause C iff $C' \subseteq C$, $C' \equiv C$, and C' is not redundant.

Theorem 30.

(I) Of any clause, a non-redundant variant exists.

(II) If there are more than one such variants then they are renamed variants of each other.

Proof. Let C be a clause $\{L_1, \ldots, L_n\}$. Let us define the following function on clauses:

$$r(D) = \big|\{L \mid L \in D \text{ and } L \text{ is redundant in } D\backslash L \text{ as a disjunction}\}\big|.$$

(I) The proof is inductive on $r(C)$.

(1) If $r(C) = 0$ then C itself is a non-redundant variant of C since $C \subseteq C$, $C \equiv C$, and C is not redundant.

(2) Assume that $r(C) \geq 1$, and there is a non-redundant variant of any clause D if $r(D) < r(C)$. Prove that there is a non-redundant variant of C.

By Definition 28, there is an $L \in C$ such that L is redundant in $C' = C\backslash L$ as a disjunction. Since $r(C') < r(C)$, there is a non-redundant variant C'' of C', i.e., $C'' \subseteq C'$, $C'' \equiv C'$, and C'' is not redundant. By Theorem 27 and Definition 21, $C' \equiv C$. Since $C'' \subseteq C' \subseteq C$, $C'' \equiv C' \equiv C$, and C'' is not redundant, C'' is a non-redundant variant of C.

(II) In the proof for (I), an inductive method for computing non-redundant variants is used, where literals are eliminated in C one by one until a non-redundant clause gets computed. Let us prove that it does not matter in what order they are eliminated, i.e., the possible resulting clauses are renamed variants.

It is sufficient to prove that the elimination of two literals results either in one single clause or in renamed variants. That is, the case when $r(C) \geq 2$ is focused. Without loss of generality, it can be assumed that L_1 is redundant in the disjunction of $\{L_2, \ldots, L_n\}$, and L_2 is redundant in the disjunction of $\{L_1, L_3, \ldots, L_n\}$. By Definition 26, $L_1\sigma = L_i$ for some $i \in \{2, \ldots, n\}$ and some σ, and $L_2\theta = L_j$ for some $j \in \{1, 3, \ldots, n\}$ and some θ. There are three distinct cases:

(1) $i, j \notin \{1, 2\}$: both L_1 and L_2 are eliminated, thus the resulting clause is $\{L_3, \ldots, L_n\}$.

(2) $i \notin \{1, 2\}$ and $j = 1$: $L_2\theta = L_1$, hence $L_2\theta\sigma = L_i$, which leads to the previous case.

(3) $i = 2$ and $j = 1$: $L_1\sigma = L_2$ and $L_2\theta = L_1$, hence $L_1\sigma\theta = L_1$. This holds iff $\sigma\theta = \epsilon$, which holds iff σ and θ are renaming substitutions. Thus the possible resulting clauses $\{L_2, L_3, \ldots, L_n\}$ and $\{L_1, L_3, \ldots, L_n\}$ are renamed variants.

□

Based on Theorem 30, we propose two simplifications:

(1) As an initializing step, simplify all the clauses in the clause set $\mathcal{C}$ you want to refute, i.e., try to refute the clause set that consists of non-redundant variants, exactly one of each clause in $\mathcal{C}$.

(2) As a simplification of an extension (E, θ), replace E with one of its non-redundant variants.

6 Conclusion

In this paper, we proposed a new calculus called hyperS tableaux, which is a refinement of the hyper tableaux calculus. HyperS tableaux eliminate the non-automatable and unwanted solutions in hyper tableaux, automate the clause set generation, and perform a kind of heuristic lookahead. We proved the soundness and completeness of this calculus. We proposed several facilities based on redundancy for tableau reduction. Furthermore, we discussed the requirements for using parametric variables, which are necessary for a theorem prover to be able to give compound answers.

References

[1] P. Baumgartner, U. Furbach, I. Niemelä, "Hyper Tableaux". *Proc. JELIA '96*, Vol. 1226 of LNAI, Springer, 1996.

[2] M. Fitting, "First-Order Logic and Automated Theorem Proving". Springer-Verlag, 1996.

[3] R. Hähnle, "Tableaux and Related Methods", *Handbook of Automated Reasoning*, by J. A. Robinson and A. Voronkov, Vol. 1, Chapter 3, p. 100-178, Elsevier and MIT Press, 2001.

[4] M. Kühn, "Rigid Hypertableaux", *Proc. KI '97: Advances in Artificial Intelligence*, Vol. 1303 of LNAI, Springer, 1997.

[5] R. Letz, K. Mayr, C. Goller, "Controlled Intergration of the Cut Rule into Connection Tableau Calculi", *Journal of Automated Reasoning*, Vol. 13, p. 297-328, 1994.

[6] F. J. Pelletier, “Seventy-Five Problems for Testing Automatic Theorem Provers”, *Journal of Automated Reasoning*. Vol. 2, p. 191-216, 1986.

[7] J. A. Rosinson, “Automated Deduction with Hyper-Resolution”, *International Journal of Computer Mathematics*. Vol. 1, p. 227-234, 1965.

[8] R. M. Smullyan, “First-Order Logic”. Springer-Verlag, 1968.

[9] K. Pásztorné Varga, M. Várterész, “A matematikai logika alkalmazásszemléletû tárgyalása”. Panem, 2003.