Proof Confluent Tableau Calculi

Reiner Hähnle and Bernhard Beckert

Dept. of Computer Science, University of Karlsruhe
76128 Karlsruhe, Germany, {reiner,beckert}@ira.uka.de

1 Introduction

A tableau calculus is proof confluent if every partial tableau proof for an unsatisfiable formula can be extended to a closed tableau. A rule application may be redundant but it can never prevent the construction of a proof; there are no "dead ends" in the proof search. Proof confluence is a prerequisite of (a) backtracking-free proof search and (b) the generation of counter examples to non-theorems.

In this tutorial we discuss the rôle and perspectives of proof confluent calculi in tableau-based theorem proving. For the sake of simplicity the discussion focuses on clause tableaux.

2 Tableaux with Selection Function

Among the more effective resolution refinements are those based on selection functions such as hyperresolution and semantic resolution. A number of calculi related to these concepts were also introduced into the world of semantic tableaux in form of various proof confluent refinements.

The emphasis so far were tableau calculi corresponding to positive hyperresolution and binary resolution with selection function. In this tutorial, more general calculi based on arbitrary selection functions with hyper extension steps are discussed. For those selection functions that correspond to Herbrand interpretations one obtains a semantic tableau analogue of semantic resolution. All introduced calculi are based on a simple, generic saturation principle leading to brief and schematic completeness proofs.

It is shown that just as model generation theorem proving (MGTP) is an instance of hyper tableaux, constraint MGTP turns out to be an instance of hyper tableaux with selection function. This gives a formal justification why constraint MGTP is a complete procedure for many applications such as quasigroup problems.

3 Proof Confluence and Strong Completeness

For practical purposes, a completeness theorem merely stating the existence of a tableau proof is not sufficient. A stronger result is needed giving the guarantee that a concrete tableau proof *search procedure* will find a closed tableau if there exists one. Let us call this (as usual) the *strong completeness* problem.

This problem can easily be solved if the calculus is *non-destructive*, i.e., if all tableaux that can be constructed from a given tableau contain that tableau as an initial subtree. In that case, one can simply arrange input clauses in a queue (on each branch) and thus ensure that enough instances of each clause are used on each branch to obtain a proof. Examples of non-destructive tableau calculi are Smullyan tableaux and Fitting's delayed instantiation rule.

Unfortunately, the standard version of clause tableaux is a *destructive* calculus. The culprit is the closure rule, which allows to instantiate free variables.

The standard solution for proof search in all destructive calculi is *depth-first iterative deepening* search, it was pioneered by Stickel, and is used, for example, in the provers Setheo, $_3T^AP$, and KoMeT. One enumerates all tableaux up to a fixed size via backtracking over possible closure rule applications. Completeness is achieved by iterative increase of the bound on tableau size.

But how, besides backtracking, can be dealt with the strong completeness problem in case the calculus is destructive but proof confluent? A strongly complete procedure performing a depth-first proof search has several advantages. The information represented by the constructed tableaux increases at each proof step; no information is lost since there is no backtracking. In addition, considering similar tableaux or sequences of tableaux in different paths of the search tree is avoided.

The problem of constructing a strongly complete proof procedure without backtracking is discussed in the last part of the tutorial. A possible solution is presented that is based on a notion of *regularity* to make sure that there are no "cycles" in the search (it is not possible to deduce the same literals, clauses, or sub-tableau again and again). In addition, each literal is assigned a "weight" in such a way that there are only finitely many different literals (up to variable renaming) of a certain weight; thus, since literals with lesser weight are deduced first, sooner or later each possible conclusion is added to all branches containing its premiss, i.e., the strategy is *fair*. To handle the destructiveness of clause tableaux, the strategy employs *reconstruction steps*. Immediately after a rule application that instantiates free variables, the expansion steps that are needed to recreate the destroyed part of the tableau are executed.

Further Information

There is a Web page for this tutorial, where slides, references, and related papers are available; the URL is `i12www.ira.uka.de/tab99-tutorial`.