Proof Search without Backtracking using Instance Streams

Martin Giese

Institut für Logik, Komplexität und Deduktionssysteme
Universität Karlsruhe, Germany
giese@ira.uka.de

Position Paper

Most existing automated proof procedures using free-variable analytic tableaux require iterative deepening and backtracking to guarantee completeness.

The general process for closing a proof with, e.g., two branches consists in first searching an instantiation for the (rigid) free variables that permits closure of one of the branches. A corresponding substitution is then applied to the tableau, and a proof of the second branch is attempted. If the instantiation chosen for the first branch was not right, proof search on the second branch will proceed up to the current iterative deepening bound and fail. Any information about the work done on the second branch is then discarded. A second way to close the first branch is sought, and the process is repeated.

Unless restrictions, like e.g. the connectedness condition, make possible rule applications strongly dependent on the applied instantiations, a significant amount of the work done on the second branch has to be repeated again and again: Apart from instantiation of some of the free variables, the formulae on the second branch are always the same, so the possible rule applications also repeat themselves. This can mean a significant amount of duplicated work.

From an abstract point of view, proof search with backtracking is a way of enumerating proof attempts. First-order satisfiability being only semi-decidable, it is clear that proof search must rely on some sort of enumeration. But it would be nice to organize this enumeration in a way that leads to less duplicated work.

Instance Streams are a way of organizing the search for instances which close a proof. The basic idea is as follows: To close a proof with two branches, one needs to find an instantiation for the free variables, that allows to close both of them. So one first expands the first branch, until a closing instance is found. Then, one does the same for the second branch. If these two instances are *compatible*, they may be joined to give an instance that closes both branches simultaneously. Otherwise, the first two instances are remembered, and further instances closing each of the two branches are sought, by expanding them if necessary. Each instance that closes one branch is checked for compatibility with all of the instances that have been found for the other one. As soon as two of these instances are compatible, the joint instance closes both branches simultaneously. This process is applied recursively, if one of the two branches splits further. No backtracking, and no iterative deepening are needed.

The name 'Instance Stream' refers to the view that a refutation procedure takes an open branch as argument and returns a *stream* — also known as *lazy list* — of instances closing this branch.

Here is a rough sketch of a leanT^AP-like function 'refute', that takes a set M of formulae in negation normal form and returns a stream of instances, i.e. a list of substitutions for free variables occurring in M, under which M can be refuted. If the initial formula set is unsatisfiable, refute returns a non-empty stream.

$$\begin{array}{ll}
\text{refute}(\ \{\alpha_1 \wedge \alpha_2\} \cup M\ ) & = \text{refute}(\ \{\alpha_1, \alpha_2\} \cup M\ ) \\
\text{refute}(\ \{\beta_1 \vee \beta_2\} \cup M\ ) & = \text{merge}(\ \text{refute}(\ \{\beta_1\} \cup M\ ),\ \text{refute}(\ \{\beta_2\} \cup M\ )\ ) \\
\text{refute}(\ \{\forall x.\gamma_1\} \cup M\ ) & = \text{refute}(\ \{\gamma_1[x/X], \forall x.\gamma_1\} \cup M)\ |_{\text{FreeVars}(\{\forall x.\gamma_1\} \cup M)} \\
\text{refute}(\ \{L, \neg L'\} \cup M\ ) & = \text{first mgu}(L, L'),\ \text{if it exists,} \\
 & \quad\ \text{followed by refute}(\ \{L, \neg L'\} \cup M\ ) \\
 & \quad\ --\ (\text{ only once for each pair of literals })
\end{array}$$

An auxiliary function 'merge' is used, which makes a new instance stream out of two given ones, by trying to join each of the instances in one with each of the instances in the other.

merge(s_0, s_1) =
  Remember all instances from s_i in buffer P_i.
  While s_0 and s_1 are not both empty,
    fetch an instance I from s_k, $k \in \{0, 1\}$ (fairly/alternatingly).
    For all $J \in P_{1-k}$,
      If J is compatible with I
        write join(I, J) to the output stream.

The author has implemented this concept in a prototypical prover named 'PrInS'. PrInS is written in the Java Programming Language, and besides instance streams uses a first-order simplification rule with constraints described in [4]. Some promising first experimental results are also given there.

Current research concerns the adaptation of refinements such as pruning, regularity, etc. (see e.g. [3]) to the Instance Stream approach. The effectiveness of various refinements in combination with Instance Streams will have to be evaluated. Efficient data structures, e.g. for representing the buffers P_i are also a central issue. Related work like that of Baumgartner [1] and Beckert [2] will have to be compared to the presented approach.

Acknowledgements: I am grateful to Reiner Hähnle, Elmar Habermalz and Bernhard Beckert for comments on drafts of this paper.

References

1. P. Baumgartner. Fdpll – a first-order davis-putnam-logeman-loveland procedure. In D. McAllester, editor, *Automated Deduction, CADE-17*, LNAI. Springer, 2000.
2. Bernhard Beckert. *Integrating and Unifying Methods of Tableau-based Theorem Proving.* PhD thesis, University of Karlsruhe, Department of Computer Science, July 1998.
3. Bernhard Beckert and Reiner Hähnle. Analytic tableaux. In W. Bibel and P. Schmitt, editors, *Automated Deduction: A Basis for Applications*, volume I, chapter 1, pages 11–41. Kluwer, 1998.
4. Martin Giese. A first-order simplification rule with constraints. Extended Abstract, submitted to FTP 2000. Available online at `http://i11www.ira.uka.de/~giese/ftp00simp.ps.gz`.