Incremental Closure of Free Variable Tableaux

Martin Giese

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

Abstract. This paper presents a technique for automated theorem proving with free variable tableaux that does not require backtracking. Most existing automated proof procedures using free variable tableaux require iterative deepening and backtracking over applied instantiations to guarantee completeness. If the correct instantiation is hard to find, this can lead to a significant amount of duplicated work. Incremental Closure is a way of organizing the search for closing instantiations that avoids this inefficiency.

1 Introduction

Since the 1980's, the technique of using free variables to postpone the choice of instantiations in the γ-expansions of tableau calculi for first-order logic is used in practically all implementations. These free variables have to be instantiated at some point in the proof search by unifying complementary literals on branches, and one faces the problem that doing this in a naïve way can lead to non-termination for some unsatisfiable sets of formulae, and thus to *incompleteness of the procedure.*

The most used way of regaining completeness employs backtracking and iterative deepening: A complexity limit for the proof is fixed, and a proof that does not exceed this complexity is sought for, using backtracking to explore the search space. If no proof is found, the limit is increased. Unfortunately, backtracking can lead to a large amount of duplicated work, because the prover forgets information which it might need again. On the other hand, the analytic free variable tableau calculus is *proof confluent*, meaning that any open tableaux for an unsatisfiable set of formulae may be closed by further expansion. This means that *the calculus* does not require backtracking, contrary to connection tableaux, for instance.

This is probably the main incentive to consider proof procedures that can do without backtracking. Another reason is that they are more suited for use in an integrated automated and interactive system: The user has more possibility of seeing what went wrong in a failed proof attempt, if all information about what has been tried so far is kept.

Lately, a number of tableau-based procedures has been proposed that work without backtracking. Most of these concentrate on overcoming the mentioned naïveté of simply closing a branch as soon as possible. In [Bil96] for instance, instead of instantiating free variables in the tableau, the set of input clauses is extended by instantiated variants, leading to a kind of saturation process. A similar approach based on the connection calculus is presented in [BEF99]. In [Bec00], an ordering restriction on the sequence of generated tableaux is imposed that forbids cycles.

This paper describes an approach in which the free variables are never instantiated in the tableau, but the various possibilities are effectively considered in parallel. We use an *incremental* approach to compute an instantiation of the free variables that closes all branches *simultaneously*, hence the name *Incremental Closure*.[1]

After defining a few basic notions, we shall present the basic idea of the approach in Sect. 3. We describe a number of possible refinements in Sect. 4, and some experimental results are quoted in Sect. 5.

2 Preliminaries

We assume a fixed first-order signature throughout this paper. Let terms and first-order formulae (without equality) over that signature be defined in the usual way. A ground term is a term that does not contain variables.

A formula is in *negation normal form* (NNF), iff negation signs appear only in front of atomic formulae $p(t_1, \ldots, t_n)$. By the application of de Morgan's rules, any formula can be transformed into an equivalent NNF formula. A formula is in *skolemized negation normal form* (SNNF), iff it is in NNF and does not contain existential quantifiers. Any formula F can be transformed by skolemization into a formula F' in SNNF that is satisfiable iff F is satisfiable. A formula is *closed* if all variable occurrences in it are bound by a quantifier.

Definition 1. *An* instantiation *is a mapping from the set of all variables to ground terms. Let* Sub^0 *denote the set of all instantiations.*

This differs from the usual concept of a *ground substitution*, in that we require *all*, i.e. infinitely many variables to be mapped.

Definition 2. *A* goal *is a finite set of formulae. A* tableau *is a finite tree where every node has zero, one, or two children, and each node is labeled with a goal. A* leaf *is a node with no children. The* leaf goals *of a tableau are the goals that label its leaves.*

A tableau for a finite set of SNNF formulae S *is defined inductively as follows:*

1. *The tableau consisting of the root node labeled with the goal S is a tableau for S, called the* initial tableau.
2. *If there is a tableau for S that has a leaf n with goal $\{\alpha_1 \wedge \alpha_2\} \cup G$, then the tableau obtained by adding a new child n' with goal $\{\alpha_1, \alpha_2\} \cup G$ to n is also a tableau for S. (α-expansion)*

[1] A predecessor to this approach was sketched in the Postition Paper[Gie00b] under the name of 'Instance Streams'.

3. *If there is a tableau for S that has a leaf n with goal $\{\beta_1 \vee \beta_2\} \cup G$, then the tableau obtained by adding two new children n', resp. n'' with goals $\{\beta_1\} \cup G$, resp. $\{\beta_2\} \cup G$ to n is also a tableau for S. (β-expansion)*
4. *If there is a tableau for S that has a leaf n with goal $\{\forall x.\gamma_1\} \cup G$, then the tableau obtained by adding a new child n' with goal $\{[x/X]\gamma_1, \forall x.\gamma_1\} \cup G$ to n, where X did not previously occur in the tableau, is also a tableau for S. (γ-expansion)*

A complementary pair *is a pair ϕ, $\neg\psi$, where ϕ and ψ are unifiable atomic formulae. A goal G is* closed under an instantiation *σ, iff there is a complementary pair $\{\phi, \neg\psi\} \subseteq G$ with $\sigma(\phi) = \sigma(\psi)$. A tableau T is* closed under an instantiation *σ, iff each leaf goal of T is closed under σ. A tableau is* closable *iff it is closed under some instantiation.*

We use this somewhat unusual formulation of tableaux labeled with sets of formulae (Smullyan [Smu68] calls them *block tableaux*) because it helps in describing the procedure. Note that in an implementation, it is sufficient to keep the leaf goals in memory; they correspond to the branches in the usual definition.

Another deviation from the usual formulations of free variable tableau calculi in that there is no rule that instantiates the free variables introduced by a γ-rule. Instead, an instantiation that closes all branches simultaneously has to be found, to decide that a tableau is closable. This is an important aspect of the incremental closure technique. It is obvious, that the usual correctness and completeness proofs for free variable tableaux are also applicable to this formulation.

Proposition 1. *Let S be a set of closed formulae in SNNF. S is unsatisfiable iff there is a closable tableau for S.*

3 Incremental Closure

From Prop. 1, it is easy to derive a complete proof procedure:

```
T := initial tableau for S
while ( not closable(T) ) do
   if expandable(T) then
      select possible expansion of T
      expand T
   else
      answer 'satisfiable'
   end
end
answer 'unsatisfiable'
```

This is a complete proof procedure, provided the selection of tableau expansions is *fair*. Being fair means that if the procedure does not terminate, any extension step possible on a goal will at some point be applied on that goal or one of its descendants. In particular, in a non-terminating run, infinitely many instances of each γ-formula will ultimately be produced on each branch.

The main problem with this proof procedure is the test $\mathsf{closable(T)}$: In general, the right combination of complementary literals has to be found in the leaf goals to compute a simultaneous unifier. This is NP-complete in the size of the leaf goals.[2]

The problem of finding the right complementary pairs has to be solved in any free variable tableau proof procedure, backtracking or not. But although the worst-case complexity cannot be reduced, a speedup can be achieved by tuning the procedure to perform well in practical cases.

The approach presented here makes the procedure more efficient by computing $\mathsf{closable(T)}$ in an incremental fashion, based on the following observations:

– If a pair of complementary literals is unifiable, it will stay unifiable after any extension. This should make an incremental algorithm worthwhile.
– An instantiation has to be found for the free variables introduced by the γ-rule. These only occur in the proof tree below the corresponding γ formula. To take advantage of this locality, the algorithm should exploit the structure of the proof tree.

3.1 An Abstract Description

In this section we shall abstract away from concrete representations of instantiations, and assume that we can perform calculations on (potentially infinite) sets of instantiations. How to represent these in an actual implementation is discussed in Sect. 3.2.

Let

$$\mathrm{unif}(\phi, \psi) := \{\sigma \in \mathrm{Sub}^0 \mid \sigma(\phi) = \sigma(\psi)\}$$

be the set of instantiations that unify two atomic formulae. We define

$$\mathrm{cl}(G) := \bigcup_{\phi, \neg\psi \in G} \mathrm{unif}(\phi, \psi)$$

to be the set of instantiations under which a goal G is closed. For a node n of a tableau, let $lg(n)$ be the set of leaf goals associated with the leaves that are descendants of n. Use this to define

$$\mathrm{cl}(n) := \bigcap_{G \in lg(n)} \mathrm{cl}(G)$$

[2] Unifiability can be decided in linear time [PW78], so with indeterministic selection of complementary pairs, $\mathsf{closable(T)}$ is in NP. On the other hand, SAT for propositional clauses can be reduced to this problem by translating each clause to one leaf goal, mapping propositional symbols to free variables, such that a goal is closable under an instantiation to $\{0, 1\}$ iff the clause is satisfied by the corresponding interpretation. E.g., translate $A \vee \neg B$ to $\{p_A(0), \neg p_A(1), p_B(0), \neg p_B(1), p_A(A), \neg p_B(B)\}$.

to be the set of instantiations under which all leaves below n are closed. Obviously, $\mathrm{cl}(\mathit{root})$, where *root* is the root node, is the set of instantiations that close the whole tableau.

We can take advantage of the tableau structure by expressing $\mathrm{cl}(n)$ recursively: If a node n has only one child n', $\mathrm{cl}(n) = \mathrm{cl}(n')$, for two children n', n'', $\mathrm{cl}(n) = \mathrm{cl}(n') \cap \mathrm{cl}(n'')$. For a leaf n labeled with goal G, $\mathrm{cl}(n) = \mathrm{cl}(G)$.

We shall clarify these notions using the following tableau:

$$\begin{tikzcd}
 & n_1 : \forall x.(qx \vee \neg px), \forall y.qy, \neg qb, pa \arrow[d, no head] & \\
 & n_2 : \underline{qX \vee \neg pX}, \forall y.qy, \neg qb, pa, \forall x.(\ldots) \arrow[dl, no head] \arrow[dr, no head] & \\
n_3 : \underline{qX}, \forall y.qy, \neg qb, pa, \forall x.(\ldots) & & n_4 : \underline{\neg pX}, \forall y.qy, \neg qb, pa, \forall x.(\ldots)
\end{tikzcd}$$

n_2 was constructed by applying a γ-expansion at n_1, and n_3, n_4 were introduced by a β-expansion at n_2. The newly introduced formulae are underlined in each goal. The goal at n_3 contains one complementary pair $qX, \neg qb$. So $\mathrm{cl}(n_3) = \mathrm{unif}(qX, qb) = \{\sigma \in \mathrm{Sub}^0 | \sigma(X) = b\}$, the set of instantiations that map X to b. Similarly, $\mathrm{cl}(n_4) = \{\sigma \in \mathrm{Sub}^0 | \sigma(X) = a\}$, because of the complementary pair $\neg pX, pa$. For $\mathrm{cl}(n_2)$ we have to find instantiations that close both leaf goals, $\mathrm{cl}(n_2) = \mathrm{cl}(n_3) \cap \mathrm{cl}(n_4)$. There are obviously no such instantiations, $\mathrm{cl}(n_2) = \emptyset$. The same holds for the root n_1, of course.

To get an incremental algorithm, we shall examine the values of cl change when a tableau expansion produces new complementary pairs. In general, one expansion step might lead to several new complementary pairs in one goal, or there might be two new goals, each of which can contain new complementary pairs. We shall examine the changes to cl induced by *one* new complementary pair $\phi, \neg\psi$ at *one* leaf l, called the *selected leaf.* If there are several new complementary pairs, these changes may be applied consecutively for each of them.

Let cl_0 denote the value of cl before taking into account $\phi, \neg\psi$, while cl is the updated value. Possible closing instantiations are never destroyed by an expansion step, so the sets cl can only grow when the tableau is expanded, i.e. $\mathrm{cl}(n) \supseteq \mathrm{cl}_0(n)$ for all nodes of the tableau. Define $\delta(n) := \mathrm{cl}(n) \setminus \mathrm{cl}_0(n)$ to be the set of *new* closing instantiations. Obviously, $\mathrm{cl}(n) = \mathrm{cl}_0(n)$, so $\delta(n) = \emptyset$, if the selected leaf l is not a descendant of n. In other words, $\delta(n)$ is non-empty only for nodes n on the path between l and the root of the tableau. For the selected leaf l, δ is given by

$$\delta(l) = \mathrm{unif}(\phi, \psi) \setminus \mathrm{cl}_0(n) \quad .$$

Using the recursive expression for $\mathrm{cl}(n)$, we can 'propagate' this change up the branch towards the root. We obtain $\delta(n) = \delta(n')$ for all nodes n with one child n'. For a node n with two children n' and n'', we assume that l lies below n'. This implies that $\mathrm{cl}(n'') = \mathrm{cl}_0(n'')$, so we have

$$\begin{aligned}\delta(n) &= \mathrm{cl}(n) \setminus \mathrm{cl}_0(n)\\ &= (\mathrm{cl}(n') \cap \mathrm{cl}(n'')) \setminus (\mathrm{cl}_0(n') \cap \mathrm{cl}_0(n''))\\ &= (\mathrm{cl}(n') \cap \mathrm{cl}_0(n'')) \setminus (\mathrm{cl}_0(n') \cap \mathrm{cl}_0(n''))\\ &= (\mathrm{cl}(n') \setminus \mathrm{cl}_0(n')) \cap \mathrm{cl}_0(n'')\\ &= \delta(n') \cap \mathrm{cl}_0(n'')\end{aligned}$$

The case where l lies below n'' is of course symmetrical.

The central idea of the incremental closure procedure is to keep track of the sets $\mathrm{cl}(n)$ and update them by propagating the additional closures $\delta(n)$ up the branch using this equation. As soon as $\delta(\mathit{root}) \neq \emptyset$, the tableau must be closable.

We shall continue the example from above to demonstrate the propagation of δ values.

$$\begin{tikzcd}[column sep=tiny]
 & n_1 : \ \forall x.(qx \vee \neg px), \forall y.qy, \neg qb, pa \arrow[d, no head] & \\
 & n_2 : \ \underline{qX \vee \neg pX}, \forall y.qy, \neg qb, pa, \forall x.(\ldots) \arrow[dl, no head] \arrow[dr, no head] & \\
n_3 : \ \underline{qX}, \forall y.qy, \neg qb, pa, \forall x.(\ldots) \arrow[d, no head] & & n_4 : \ \underline{\neg pX}, \forall y.qy, \neg qb, pa, \forall x.(\ldots) \\
n_5 : \ \underline{qY}, qX, \neg qb, pa, \forall x.(\ldots), \forall y.qy & &
\end{tikzcd}$$

There is a new node n_5 stemming from a γ-expansion at n_3. This leads to the new complementary pair $qY, \neg qb$. Not taking this into account leads to: $\mathrm{cl}_0(n_3) = \mathrm{cl}_0(n_5) = \{\sigma \in \mathrm{Sub}^0 | \sigma(X) = b\}$, $\mathrm{cl}_0(n_4) = \{\sigma \in \mathrm{Sub}^0 | \sigma(X) = a\}$, and $\mathrm{cl}_0(n_1) = \mathrm{cl}_0(n_2) = \emptyset$. These are the values we derived for cl on the previous page. Now, including $qY, \neg qb$, we get

$$\delta(n_5) = \mathrm{unif}(qY, qb) \setminus \mathrm{cl}_0(n_5) = \{\sigma \in \mathrm{Sub}^0 | \sigma(Y) = b \text{ and } \sigma(X) \neq b\}$$

This allows us to calculate δ for all nodes between n_5 and the root: After $\delta(n_3) = \delta(n_5)$, we have

$$\delta(n_2) = \delta(n_3) \cap \mathrm{cl}_0(n_4) = \{\sigma \in \mathrm{Sub}^0 | \sigma(Y) = b \text{ and } \sigma(X) = a\}$$

This in turn leads to $\delta(n_1) = \delta(n_2) \neq \emptyset$, so the proof is closable, namely by any instantiation that maps X to a and Y to b.

Still assuming we could calculate with infinite sets of instantiations, we shall now show how the computation and propagation of the δ values is organized. The procedure shall be described in a state-based way, but it turns out that operations on the state will typically be local. For that reason, we shall take an object oriented view.

Every leaf goal has an associated Sink object. A sink is an object capable of receiving a set of instantiations and performing some computation on it. This is realized by giving a put method to the Sink objects that takes a set of instantiations as parameter. The proof procedure will call this method after every expansion step with any set $\delta(n)$ of new closing instantiations coming from a new complementary pair i.e.

```
goal.sink.put($\mathrm{unif}(\phi, \psi) \setminus \mathrm{cl}_0(n)$)
```

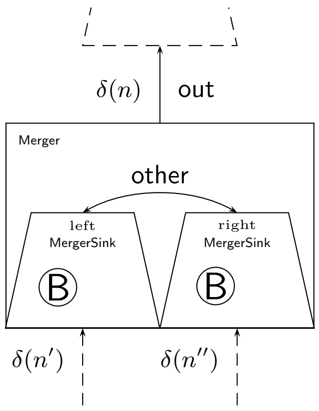
We shall see further down how $\mathrm{cl}_0(n)$ is extracted from the data structures.

There are two kinds of objects that act as sinks. One is the RootSink, which will receive $\delta(root)$. This contains a flag closable that records whether a non-empty set of instantiations has yet been received:

```
RootSink::put(S) **is**
  **if** S nonempty **then**
    closable := **true**
  **end**
**end**
```

The other kind of sink is provided by Merger objects which correspond to the splits in the tableau and are responsible for calculating the intersections $\delta(n) = \delta(n') \cap \mathrm{cl}_0(n'')$.

The structure of a Merger is shown in the following diagram:



It consists of two MergerSink objects, one to receive $\delta(n')$ and one for $\delta(n'')$. The current set $\mathrm{cl}(n')$, resp. $\mathrm{cl}(n'')$ is stored in a buffer B in the corresponding input sink. Furthermore there is a reference out to an output sink, to which $\delta(n)$ will be passed on. The two sinks are mutually connected by an association other, so they can access each others buffers via other.B. Accordingly, the put method of the MergerSink object works as follows:

```
MergerSink::put(S) **is**
  J := S ∩ other.B    // $\delta(n) = \delta(n') \cap \mathrm{cl}_0(n'')$
  B := B ∪            // $\mathrm{cl}(n) = \mathrm{cl}_0(n) \cap \delta(n)$
  out.put(J)
**end**
```

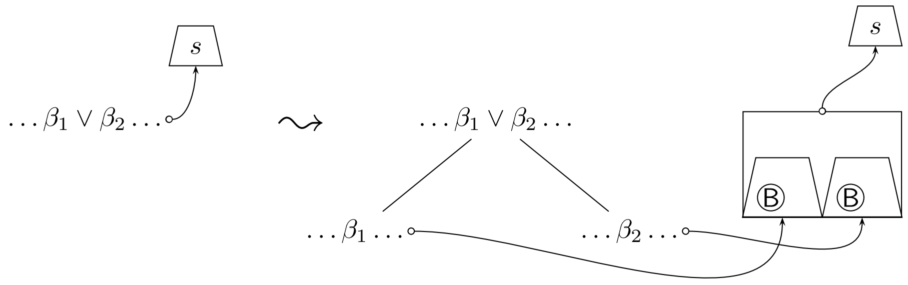
It only remains to see how $\mathrm{cl}_0(n)$ can be computed to determine $\delta(n) = \mathrm{unif}(\phi, \psi) \setminus \mathrm{cl}_0(n)$ for a new closure. There are two cases: If the goal is associated with the RootSink, $\mathrm{cl}_0(n)$ must be empty, because the proof would otherwise be closed already. If it is associated with a MergerSink, this sink contains the current value of $\mathrm{cl}_0(n)$ in its buffer B.

The proof procedure is now changed as follows:

```
T := initial tableau for S
associate RootSink r with goal of T
while ( not r.closable ) do
  if expandable(T) then
    select possible expansion of T
    expand T
    possibly generate new Merger
    handle new complementary pair
  else
    answer 'satisfiable'
  end
end
answer 'unsatisfiable'
```

At the initialization, a RootSink object is associated with the single goal of the tableau.

In the case of a β-expansion, i.e. a new split in the tableau, the step ‘possibly generate new Merger’ creates a new Merger object. The buffers B are initialized with the current value of cl_0 of the parent node. The output of the merger object is sent to the sink s of the parent node, and the new child nodes are associated with the input sinks of the merger, as shown in the following diagram:



s
$\ldots\beta_1 \vee \beta_2\ldots$
$\leadsto$
s
$\ldots\beta_1 \vee \beta_2\ldots$
B
B
$\ldots\beta_1\ldots$
$\ldots\beta_2\ldots$

After the sinks have been updated, the procedure checks for new complementary pairs introduced by the expansion, calculates $\delta(n) = \mathrm{unif}(\phi, \psi) \setminus \mathrm{cl}_0(n)$ for each of them, and sends $\delta(n)$ into the associated sink of the goal.

After all new closing instantiations have been passed to the sinks, the tableau is closable, if the closable flag of the root sink has been set.

3.2 Representation of Instantiation Sets

We have so far assumed that we can compute with infinite sets of instantiations. To get closer to a concrete implementation, we have to show how these may be represented with finite data structures. We shall briefly describe the representations used in the prototypical prover PrInS. (Prover with Instance Streams—referring to the streams of closing instantiations passed between the Sink objects.)

We use *syntactic equality constraints* to denote sets of instantiations: These are first-order formulae with equality as only predicate symbol, which are interpreted over the free term algebra. A constraint represents the set of instantiations that satisfy it. E.g. $\mathrm{unif}(p(X,b),p(a,Y))$ yields a constraint $X \equiv a \& Y \equiv b$, that is satisfied by all instantiations that map X to a and Y to b. As usual for unification, these constraints are kept in a 'solved form', that makes it easy to determine their satisfiability. The intersection of sets of instantiations required in the Mergers corresponds to the conjunction of constraints. In the updates of the MergerSinks' buffers B, set union is required which could be represented as disjunction of constraints. There is however no need to handle arbitrary disjunctive constraints: The buffers B can be implemented as lists of conjunctive constraints. The put method then looks as follows:

```
MergerSink::put(C) is
  foreach D in other.B do
    J := C & D
    if J satisfiable then
      out.put(J)
    end
  end
  add C to B
end
```

The constraints passed into the put methods are then purely conjunctive.

Finally, the set difference operation can be modeled either by taking negation into the constraint language, or by introducing subsumption checks at various places. We will not elaborate this here. See e.g. [Com91] for a survey on syntactic constraint solving methods.

4 Refinements

The Incremental Closure approach has the desirable property that it can easily be refined in a number of ways. We stress this point, because incremental closure is surely not the answer to all problems in automated theorem proving. It is therefore important to see how this new technique can be combined with successful existing approaches.

This section presents a number of possible refinements. While some of them are particular to the incremental closure technique, many are adaptations of refinements known from backtracking procedures.

4.1 Restriction of Instantiation Domains

On the abstract level, instead of passing instantiations for all free variables through the sink structure, it is possible to define the method to work with instantiations of only the free variables actually present at a certain tableau node.

For instance, in the following tableau:[3]

$$\begin{tikzcd}[arrows=dash]
 & \vdots & \\
 & \forall u, v.p(X, u, v) \lor q(u, v, Y) \arrow[d] & \\
 & p(X, U, V) \lor q(U, V, Y) \arrow[dl] \arrow[dr] & \\
p(X, U, V) & & q(U, V, Y)
\end{tikzcd}$$

assume that the left branch may be closed for instantiations satisfying $X \equiv U$, and the right branch for $U \equiv Y$. The Merger corresponding to the split will find that both branches are closable with $X \equiv U \& U \equiv Y$. But U does not exist in the tableau above the γ-expansion, so this sub-tableau can be considered closable for all instantiations satisfying $X \equiv Y$.

In terms of constraints, the restriction to a subset of occurring variables corresponds to existential quantification: $X \equiv Y$ is equivalent to $\exists U, V.X \equiv U \,\&\, U \equiv Y$. This variation may be implemented by introducing a new kind of Sink object at every γ-expansion that computes the domain restriction.

This modification has several advantages:

- As in the example above, the existentially quantified constraints can often be simplified. Thus, they consume less space in the buffers B.
- Assume that a new combination of complementary literals leads to $X \equiv V \,\&\, V \equiv Y$ in the example. This would have to be handled separately in the original version, but domain restriction leads to $X \equiv Y$ as above. A subsumption check can be used to avoid further redundant computations.

4.2 Delete Propositionally Closable Branches

It occasionally happens that a sub-tableau is closable under *any* instantiation. This is the case in proofs of propositional formulae, where no free variables are required at all, but it can also happen with first-order formulae if there is a complementary pair that is unifiable without any further instantiation. It is useless to expand that part of the tableau any further, because no more closing instantiations can be found. The sub-tableau is called *propositionally closable.*

In the implementation using constraints, this corresponds to a constraint (equivalent to) *true* being passed through the sink structure. If this is detected, the corresponding goals and the Sink structure may be deleted to reduce memory consumption.

4.3 Using Buffers for Goal Selection

So far, the Sink structure built during a proof has only been used to check whether the tableau is closable. It turns out that it can also be useful for goal selection, i.e. deciding on which goal the next expansion step should take place.

[3] We shall adopt a more familiar and compact notation for tableaux here and in the sequel, by writing only the newly introduced formulae of each goal.

The buffers B contain representations of the closing instantiation for subtableaux. If, for instance, one subtree of a node has no closing instantiation yet, while the other does, expansions should take place in the first subtree, until at least one closing instantiation has also been found there. It is also possible to use the size of the buffers or of the constraints they contain for heuristics that tend to expand branches that seem harder to close.

4.4 Pruning

Pruning (see e.g. [BH98,BFN96]) is an important technique known from backtracking procedures that can reduce the search space dramatically: The prover keeps track of the *ancestry* of formulae, i.e. the set of formulae in the tableau which were used to derive it.[4] If a branch is closed by unifying a particular complementary pair $\phi, \neg\psi$, the prover examines the β-expansions that occurred earlier on the branch. If for a particular expansion, neither the ancestry of ϕ, nor that of $\neg\psi$ contains the sub-formula $\beta_{1/2}$ introduced by the expansion, then the closure would have been possible without that expansion. Consequently, the expansion can be removed *a posteriori*, saving the work of closing the other branch introduced by it. Of course, the decision for that particular complementary pair might be revised in a backtracking step, and then the expansion has to be reintroduced.

We will now see how the pruning technique can be adapted to the incremental closure approach. We record for each formula an ancestry of β-expansions on which it depends. We can use references to the Merger objects for this, as there is exactly one of these for each β-expansion. In the abstract view of the procedure, we now compute with sets of pairs (σ, h) of instantiations with ancestries, instead of just sets of instantiations. We have to redefine the operations unif, $\cap$, $\cup$ and $\setminus$ to work with sets of such pairs. In particular, the $\cap$ operation in the Merger has to *combine* the histories of instantiations.

The 'pruning' takes place, when a Merger m receives an instantiation that does not have m in its ancestry: it can pass such an instantiation to the output sink independently of the contents of the buffer of the other branch:

```
MergerSink::put(S) **is**
  P := $\{(\sigma, h) \in$ S | this Merger $\notin h\}$
  out.put(P)
  S := S $\setminus$ P;
  J := S $\cap$ other.B
  B := B $\cup$ S
  out.put(J)
**end**
```

As the complementary pair that led to this closing instantiation might not be the one that is ultimately needed to close the proof, the other branch may not, in general, be deleted. But the gain of passing a closing instantiation further up the Sink structure turns out to be very important in practice. Of course, in the case

[4] Actually, it suffices to record only the β-subformulae introduced by β-expansions.

of a propositional closure, it is even possible to delete the whole sub-tableau, giving an even greater advantage.

4.5 Constraints

A wide range of refinements and variations of the incremental closure method becomes possible if the prover is modified to work with *constrained formulae*. A constrained formula is a pair $\phi \ll C$ of a formula ϕ and a constraint C. The meaning of this is that ϕ may be used to close a branch only if the instantiation of the free variables of the tableau satisfies the constraint. Constrained formulae may be used to port tableau rules that normally require an instantiation to the incremental closure method, and also for restrictions of the search procedure that limit the permissible instantiations in some way.

Rules introducing constraints. In [Gie00a], a simplification rule using constraints was presented. Consider for instance a tableau branch containing the formulae

$$(1) : \forall y.(q(y) \land p(X))$$
$$(2) : p(a)$$

In a backtracking framework, (1) could be simplified with (2), by instantiating X with a, then replacing the occurrence of $p(X)$ by *true*, and finally rewriting the formula to $\forall y.q(y)$. The original, unsimplified formula (1) could be discarded. It would however be necessary to backtrack over the instantiation of X. Using constraints, one would not perform the instantiation explicitly; instead one would derive the constrained formula

$$(3) : \forall y.q(y) \ll X \equiv a$$

and replace the original formula by

$$(4) : \forall y.(q(y) \land p(X)) \ll X \not\equiv a \quad ,$$

using the constraint to keep track of the fact, that for instantiations with $X \equiv a$, formula (3) should be used instead of (4).

This approach blends perfectly with the incremental closure method. Only one change is needed: When a new complementary pair $\phi \ll C, \neg\psi \ll D$ is found, the constraints of the formulae have to be added to the unification constraint. One defines:

$$\mathrm{unif}(\phi \ll C, \psi \ll D) := \mathrm{unif}(\phi, \psi) \,\&\, C \,\&\, D \quad .$$

The same approach can be used to build an incremental closure version of hyper tableaux ([Bau98]) with (rigid) free variables: For a hyper tableau rule

$$p(a) \to q(y) \lor r(y)$$

and a literal $p(X)$, one can produce an expanded tableau

$$\begin{tikzcd}
& \vdots & \\
& p(X) \arrow[dl, no head] \arrow[dr, no head] & \\
q(Y) \ll X \equiv a & & r(Y) \ll X \equiv a
\end{tikzcd}$$

where Y is a new free variable.

Another use for constrained formulae is equality handling: In [NR01], Sect. 5, Nieuwenhuis and Rubio point out the importance of using ordering constraints to reduce the search space in automated equality reasoning, and [Bec94] presents a constraint based method for equality handling in tableaux that can be neatly integrated with the incremental closure approach.

To use ordering constraints, one simply has to extend the constraint language to contain an ordering predicate $\prec$ in addition to the syntactic equality $\equiv$. The semantics of constraints is given by fixing the interpretation of this predicate to some suitable reduction ordering.

Restrictions introducing constraints. Constraints can also be introduced to adapt certain proof search restrictions from backtracking procedures, in a similar way to what is described in [LS01], Sect. 8.

One example is *regularity*. In its simplest form, the regularity condition requires that no rule is applied that introduces a formula on a branch that already occurs on it. While this is easy to enforce for the (purely academic) case of ground tableaux, it requires a certain effort when free variables are used, because two formulae might become equal through an instantiation.

In the incremental closure framework, constraints can be used to ensure regularity. A goal containing $p(a)$ and $p(X) \vee q(X)$ might be expanded thus:

$$\begin{tikzcd}
& p(a) \arrow[d, no head] & \\
& p(X) \vee q(X) \arrow[dl, no head] \arrow[dr, no head] & \\
p(X) \ll X \not\equiv a & & q(X) \ll X \not\equiv a
\end{tikzcd}$$

Then, if the instantiation $X \equiv a$ ultimately *does* lead to a proof of, say, the left branch, the ancestry of that instantiation cannot contain this β-split, so the pruning mechanism will take care that the redundant splitting expansion does no harm.

5 Experimental Results

In this section we shall quote some results obtained with an experimental implementation of the iterative closure procedure.

To cleanly separate the effects of incremental closure from those of various refinements, the technique was tested with a very simple implementation: No refinements like pruning or simplification were employed. Only the goal selection strategy from Sect. 4.3 was used. Formulae in goals are kept in a list and the first formula in this list is used for expansion. On the other hand, an equally simple backtracking prover was implemented using the same data-structures. Iterative deepening was applied on the number of γ-expansions per branch. The proof search of this backtracking prover is practically identical to that of lean$T\!A\!P$[BP94].

Comparing the two provers on some simple probelms (harder problems require refinements in both cases) shows that the incremental closure prover is nearly always faster. The difference is particularly noticeable in cases that require heavy backtracking, i.e. where many complementary pairs are found that do *not* lead to a proof. This is the case, e.g. in SYN054+1 from the TPTP library, where the backtracking procedure requires 14317 rule applications and 37817 unifications versus 151 rule applications and 680 unifications with incremental closure. The problem family $p(c) \wedge \neg p(f^n(c)) \wedge \forall x.(p(x) \rightarrow p(f(x)))$ also shows this phenomenon very clearly, because there are many possible closures, and only few of them are correct for a low depth limit. For $n = 15$, the backtracking prover performs over 300,000 rule applications and over 1,000,000 unifications, while the incremental closure procedure requires 32 rule applications and 133 unifications.

The current implementation incorporates most of the refinements given in Sect. 4. It proves 161 of the 237 full-first-order theorems without equality in TPTP v.2.3.0, one of which, SYN067+1, is rated 0.67.

One would expect the described procedure to give rise to memory problems. But with resource limits of 300 s CPU and 160 MB heap, only about 30% of the failures were due to lack of memory. We hope to further reduce this amount by implementing more powerful rules and a better goal selection strategy, leading to shorter proofs. The idea is that without backtracking, one can afford to spend more time on individual rule applications, as these do not need to be repeated.

6 Conclusion, Related Work, and Future Research

We have presented an approach to eliminate backtracking from a proof procedure for free variable tableaux. It is built around the idea of incrementally computing instantiations that close sub-tableaux until one global instantiation is found that closes the whole tableau. We have demonstrated that this technique allows various refinements to be incorporated in the calculus and the procedure. Finally we have given some experimental results obtained by comparing the approach with a backtracking solution.

A similar approach has independently been described by B. Konev and T. Jebelean in [KJ00]. However, they hardly consider possibilies for refinements.

Further work includes experiments with integrated equality handling, search for specialized data structures, e.g. for the buffers B, better goal selection strategies and adaptations of further refinements from backtracking procedures. It might also be interesting to experiment with a parallelized implementation.

Acknowledgments. I thank Wolfgang Ahrendt, Elmar Habermalz, Reiner Hähnle and the anonymous referees for their numerous and helpful comments on drafts of this paper.

References

[Bau98] Peter Baumgartner. Hyper Tableaux — The Next Generation. In Harrie de Swart, editor, *Proc. International Conference on Automated Reasoning with Analytic Tableaux and Related Methods, Oosterwijk, The Netherlands*, number 1397 in LNCS, pages 60–76. Springer-Verlag, 1998.

[Bec94] Bernhard Beckert. A completion-based method for mixed universal and rigid E-unification. In Alan Bundy, editor, *Proc. 12th Conf. on Automated Deduction CADE, Nancy/France*, LNAI 814, pages 678–692. Springer, 1994.

[Bec00] Bernhard Beckert. Depth-first proof search without backtracking for free variable clausal tableaux. In P. Baumgartner and H. Zhang, editors, *3rd Int. Workshop on First-Order Theorem Proving (FTP), St. Andrews, Scotland, TR 5/2000 Univ. of Koblenz*, pages 44–55, 2000.

[BEF99] Peter Baumgartner, Norbert Eisinger, and Ulrich Furbach. A confluent connection calculus. In Harald Ganzinger, editor, *Proc. 16th International Conference on Automated Deduction, CADE-16, Trento, Italy*, volume 1632 of *LNCS*, pages 329–343. Springer-Verlag, 1999.

[BFN96] Peter Baumgartner, Ulrich Furbach, and Ilkka Niemelä. Hyper tableaux. In José Júlio Alferes, Luís Moniz Pereira, and Ewa Orłowska, editors, *Proc. European Workshop: Logics in Artificial Intelligence, JELIA*, volume 1126 of *LNCS*, pages 1–17. Springer-Verlag, 1996.

[BH98] 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.

[Bil96] Jean-Paul Billon. The disconnection method: a confluent integration of unification in the analytic framework. In P. Miglioli et al., editor, *Theorem Proving with Tableaux and Related Methods, TABLEAUX'96, Terrasini, Italy*, volume 1071 of *LNCS*, pages 110–126. Springer-Verlag, 1996.

[BP94] Bernhard Beckert and Joachim Posegga. leanT^AP: Lean tableau-based theorem proving. extended abstract. In Alan Bundy, editor, *Proceedings, 12th International Conference on Automated Deduction (CADE), Nancy, France*, volume 814 of *LNCS 814*, pages 793–797. Springer-Verlag, 1994.

[Com91] Hubert Comon. Disunification: a survey. In Jean-Louis Lassez and Gordon Plotkin, editors, *Computational Logic: Essays in Honor of Alan Robinson*, chapter 9, pages 322–359. MIT Press, Cambridge, MA, USA, 1991.

[Gie00a] Martin Giese. A first-order simplification rule with constraints. In Peter Baumgartner and Hantao Zhang, editors, *3rd Int. Workshop on First-Order Theorem Proving (FTP), St. Andrews, Scotland, TR 5/2000 Univ. of Koblenz*, pages 113–121, 2000.

[Gie00b] Martin Giese. Proof search without backtracking using instance streams, position paper. In Peter Baumgartner and Hantao Zhang, editors, *3rd Int. Workshop on First-Order Theorem Proving (FTP), St. Andrews, Scotland, TR 5/2000 Univ. of Koblenz*, pages 227–228, 2000.

[KJ00] Boris Konev and Tudor Jebelean. Using meta-variables for natural deduction in theorema. In M. Kohlhase and M. Kerber, editors, *Proceedings of Calculemus-2000 Conference.* Electronic Notes in Computer Science, 2000.

[LS01] Reinhold Letz and Gernot Stenz. Model elimination and connection tableau procedures. In Alan Robinson and Andrei Voronkov, editors, *Handbook of Automated Reasoning*. Elsevier Science, 2001. to appear.

[NR01] Robert Nieuwenhuis and Albert Rubio. Paramodulation-based theorem proving. In Alan Robinson and Andrei Voronkov, editors, *Handbook of Automated Reasoning*. Elsevier Science, 2001. to appear.

[PW78] M. S. Paterson and M. N. Wegman. Linear unification. *Journal of Computer and System Sciences*, 16(2):158–167, April 1978.

[Smu68] Raymond M. Smullyan. *First-Order Logic*, volume 43 of *Ergebnisse der Mathematik und ihrer Grenzgebiete*. Springer-Verlag, New York, 1968.