FIRST-ORDER LOGIC: THE METHOD OF ANALYTIC TABLEAUX

Course “Computational Logic”

Wolfgang Schreiner
Research Institute for Symbolic Computation (RISC)
Wolfgang.Schreiner@risc.jku.at

JOHANNES KEPLER UNIVERSITY LINZ

The Method of Analytic ("Semantic") Tableaux

A proof calculus for fist-order logic (Evert W. Beth, 1955) based on two elements:

- Tableau: a way to systematically organize the consequences of a formula.
  - A "proof tree" for the unsatisfiability of a formula (plural: "tableaux").
- Unification: a way to deduce (probably) useful instances of quantified formulas.
  - A generalized form of "matching" a desired goal to the available knowledge.

This method deals with the problem of quantifier instantiation in a much more subtle way than generating all possible ground instances.

Propositional Tableau

A tree that refutes a propositional formula by exhibiting all its consequences.

- Initialization: the root is labeled with the formula to be refuted.
- Expansion: the tree is expanded according to the following rules:

$$\frac{\neg\neg A}{A} \qquad \frac{\neg\top}{\bot} \qquad \frac{\neg\bot}{\top} \qquad \frac{A \wedge B}{\begin{array}{l}A\\B\end{array}} \qquad \frac{A \vee B}{A \mid B} \qquad \frac{A \Rightarrow B}{\neg A \mid B} \qquad \frac{A \Leftrightarrow B}{\begin{array}{l}A \Rightarrow B\\B \Rightarrow A\end{array}}$$

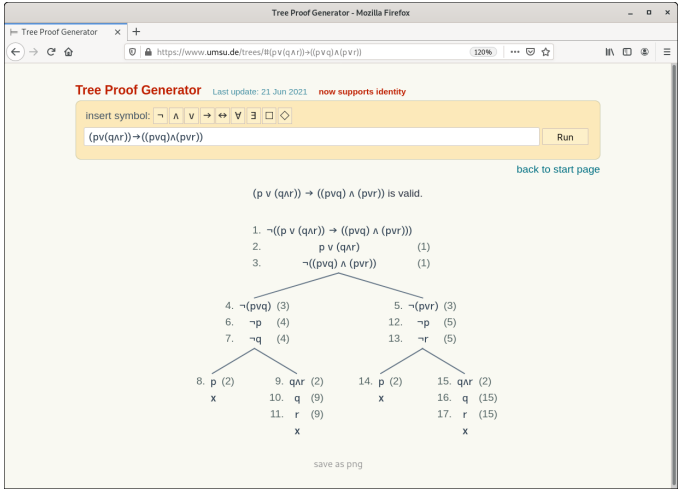
$$\frac{\neg(A \wedge B)}{\neg A \mid \neg B} \qquad \frac{\neg(A \vee B)}{\begin{array}{l}\neg A\\\neg B\end{array}} \qquad \frac{\neg(A \Rightarrow B)}{\begin{array}{l}A\\\neg B\end{array}} \qquad \frac{\neg(A \Leftrightarrow B)}{\neg(A \Rightarrow B) \mid \neg(B \Rightarrow A)}$$

  - A rule is applicable if its top formula occurs along some branch in the tree; the application of the rule extends this branch according to the shape of the bottom of the rule:
    - A: extends the branch by the single formula A.
    - $\begin{array}{l}A\\B\end{array}$: extends the branch by both formulas A and B.
    - $A \mid B$: splits the branch into two by adding one node with A and one node with B.
- Closure: a branch is closed if both formulas A and $\neg A$ occur on it.

If all branches of the tableau are closed, its root formula is unsatisfiable.

Tree Proof Generator

https://www.umsu.de/trees



Correctness of the Method

The method of “propositional tableaux” is sound and complete.

- Soundness: if a closed tableau can be derived, its root formula is unsatisfiable.
  - Proof sketch: the tableau represents a formula that is equivalent to the root formula:
    - Every branch denotes the *conjunction* of all formulas that label its nodes.
    - The tableau represents the *disjunction* of all its branches.

    It can be easily checked that this property holds initially and is preserved by every expansion rule. If a branch is closed, it represents an unsatisfiable formula. Thus, if the tableau is closed, the original formula is equivalent to a disjunction of unsatisfiable formulas, which is itself unsatisfiable.
- Completeness: from every unsatisfiable formula, a closed tableau can be derived.
  - Proof sketch: the expansion rules transform formulas into “structurally simpler formulas” (the outermost connective of the original formula has less binding power than that of each derived formula); thus eventually no more rule is applicable. We assume that this tableau is not closed and derive a contradiction. Since the tableau is not closed, it has an open branch. From the fact that no more rule is applicable, one can deduce that the conjunction of all literals on that branch represents a satisfying valuation of the branch; this contradicts the assumption that the formula is unsatisfiable.

The DNF of a formula can be deduced from its fully expanded tableau (how?).

First-Order Tableau

A straight-forward extension of a propositional tableau to first-order logic.

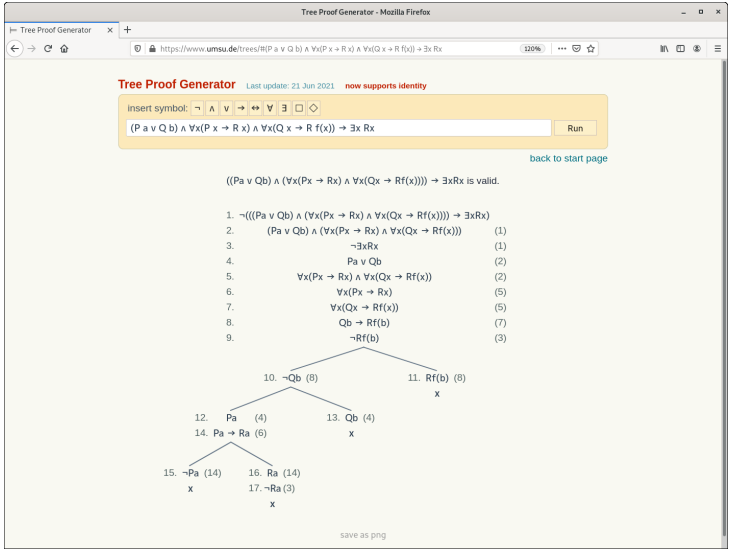
- Initialization: the root of the tree is labeled by a closed first-order formula.
- Expansion/Closure: as for a propositional tableau with the following additional rules:

$$\frac{\forall x.\ F}{F[t/x]} \qquad \frac{\exists x.\ F}{F[c/x]}$$

$$\frac{\neg\forall x.\ F}{\neg F[c/x]} \qquad \frac{\neg\exists x.\ F}{\neg F[t/x]}$$

  - Similar to the rules ($\forall$-L) and ($\exists$-R) in sequent calculus, but with a clear distinction between variables and terms:
    - An arbitrary ground term t.
    - A fresh ("Skolem") constant c.
  - Soundness: as for the propositional tableau, also a first-order tableau represents a formula that is equivalent to the root formula.
  - Completeness: the systematic construction of a tableau is possible similar to the systematic construction of a proof tree in sequent calculus.

Tree Proof Generator



Substitutions

We now turn to the topic of unification, starting with the prerequisites.

- A substitution $\sigma : \mathcal{X} \to \text{Term}$ is a function that maps every variable to a term.
  - The support $\sup(\sigma)$ of σ is the set of all variables $x \in \mathcal{X}$ with $\sigma(x) \neq x$.
  - $\sigma = [x_1 \mapsto t_1, \ldots, x_n \mapsto t_n]$: $\sup(\sigma) \subseteq \{x_1, \ldots, x_n\}, \sigma(x_1) = t_1, \ldots, \sigma(x_n) = t_n$.
- The application $t\sigma$ of a substitution σ to a term t is defined as follows:

$$
\begin{aligned}
x\sigma &:= \sigma(x) \\
c\sigma &:= c \\
f(t_1, \ldots, t_n)\sigma &:= f(t_1\sigma, \ldots, t_n\sigma)
\end{aligned}
$$

- Example:

$$
\begin{aligned}
\sigma &= [x \mapsto f(x, y), y \mapsto g(a)] \\
t &= f(\underline{x}, g(f(\underline{x}, f(\underline{y}, z)))) \\
t\sigma &= f(\underline{f(x, y)}, g(f(\underline{f(x, y)}, f(\underline{g(a)}, z))))
\end{aligned}
$$

Substitution Ordering

- The composition $\sigma_1\sigma_2$ of two substitutions σ_1 and σ_2 is the unique substitution that satisfies the following property:

$$t(\sigma_1\sigma_2) = (t\sigma_1)\sigma_2$$

  - Example: $\sigma_1 = [x \mapsto f(y), y \mapsto z]$, $\sigma_2 = [x \mapsto a, y \mapsto b, z \mapsto y]$.

$$\sigma_1\sigma_2 = [x \mapsto f(y)\sigma_2, y \mapsto z\sigma_2, z \mapsto y] = [x \mapsto f(b), y \mapsto y, z \mapsto y] = [x \mapsto f(b), z \mapsto y]$$

- Substitution σ is at least as general as substitution σ', written as $\sigma \preceq \sigma'$, if there is some subsitution ϑ such that $\sigma\vartheta = \sigma'$.
  - Relation $\preceq$ is a quasi-ordering (i.e., reflexive and transitive).
  - Example: $\sigma_1 = [x \mapsto y], \sigma_2 = [y \mapsto x], \sigma_3 = [x \mapsto a, y \mapsto a]$.
    - $\sigma_1 \preceq \sigma_2$ because $\sigma_1[y \mapsto x] = [x \mapsto y][y \mapsto x] = [x \mapsto x, y \mapsto x] = [y \mapsto x] = \sigma_2$.
    - $\sigma_2 \preceq \sigma_1$ because $\sigma_2[x \mapsto y] = [y \mapsto x][x \mapsto y] = [y \mapsto y, x \mapsto y] = [x \mapsto y] = \sigma_2$.
    - $\sigma_1 \preceq \sigma_3$ because $\sigma_1[y \mapsto a] = [x \mapsto y][y \mapsto a] = [x \mapsto a, y \mapsto a] = \sigma_3$.

Unification

- Substitution σ is a unifier of two terms t_1 and t_2 if $t_1\sigma = t_2\sigma$.
  - Example: the following substitutions are unifiers of terms $f(x, z)$ and $f(y, g(a))$:
    - $\sigma_1 = [x \mapsto y, z \mapsto g(a)]$
    - $\sigma_2 = [y \mapsto x, z \mapsto g(a)]$
    - $\sigma_3 = [x \mapsto a, y \mapsto a, z \mapsto g(a)]$
    - $\sigma_4 = [x \mapsto g(z), y \mapsto g(z), z \mapsto g(a)]$
    - $\ldots$
- Substitution σ is a most general unifier of t_1 and t_2 if it is a unifier of t_1 and t_2 and $\sigma \preceq \sigma'$ for every unifier σ' of t_1 and t_2.
  - A most general unifier is unique up to variable renaming.
  - Example: σ_1 and σ_2 above are most general unifiers of $f(x, z)$ and $f(y, g(a))$.
- Substitution σ is a (most general) unifier of a set S of term pairs if it is a (most general) unifier of every term pair in S.

The unification problem is to compute for two terms a most general unifier (respectively to determine that these terms cannot be unified).

Inhibitors of Unification

When can two terms not be unified?

1. Both terms are not variables and differ in their outermost symbol.
   - Constants a and b cannot be unified.
   - Constant a cannot be unified with function application $f(x)$.
   - Function applications $f(x)$ and $g(x)$ cannot be unified.
2. One term is a variable x, the other one a function application in which x occurs.
   - Variable x cannot be unified with function application $f(x)$.

All other differences between two terms can be resolved by variable substitutions.

A Unification Algorithm

John Alan Robinson, 1965.

procedure UNIFY(t, u, σ) ▷ called with $\sigma = [\ ]$, returns a most general unifier of t and u
  if t and u are the same constant c **then** ▷ but fails with an error message if t and u cannot be unified
    return σ
  else if t and u are $f(t_1, \ldots, t_n)$ and $f(u_1, \ldots, u_n)$ **then** ▷ the same n-ary function symbol f
    for i **from** 1 **to** n **do**
      $\sigma \leftarrow$ UNIFY(t_i, u_i, σ)
    end for
    return σ
  else if t is a variable x **then** ▷ eliminate x
    if $x \in \sup(\sigma)$ **return** UNIFY($\sigma(x), u, \sigma$)
    if x does not occur in $u\sigma$ **return** $\sigma[x \mapsto u\sigma]$
  else if u is a variable x **then** ▷ eliminate x
    if $x \in \sup(\sigma)$ **return** UNIFY($t, \sigma(x), \sigma$)
    if x does not occur in $t\sigma$ **return** $\sigma[x \mapsto t\sigma]$
  end if
  fail “unification is impossible”
end procedure

Algorithm needs worst-case exponential time and space.

Examples

- Unify $f(x, g(a), g(z))$ and $f(g(y), g(y), g(g(x)))$:
  1. Unify $f(\underline{x}, g(a), g(z))$ and $f(\underline{g(y)}, g(y), g(g(x)))$ with $x \notin \sup(\sigma) = \emptyset$:
     - $\sigma = [x \mapsto g(y)]$
  2. Unify $f(x, g(\underline{a}), g(z))$ and $f(g(y), g(\underline{y}), g(g(x)))$ with $y \notin \sup(\sigma) = \{x\}$:
     - $\sigma = [x \mapsto g(y)][y \mapsto a] = [x \mapsto g(a), y \mapsto a]$
  3. Unify $f(x, g(a), g(\underline{z}))$ and $f(g(y), g(y), g(\underline{g(x)}))$ with $z \notin \sup(\sigma) = \{x, y\}$:
     - $\sigma = [x \mapsto g(a), y \mapsto a][z \mapsto g(g(a))] = [x \mapsto g(a), y \mapsto a, z \mapsto g(g(a))]$
     - Unification succeeds:

       $f(x, g(a), g(z))\sigma = f(g(a), g(a), g(g(g(a)))) = f(g(y), g(y), g(g(x)))\sigma.$
- Unify $f(x, g(y))$ and $f(y, x)$:
  1. Unify $f(\underline{x}, g(y))$ and $f(\underline{y}, x)$ with $x \notin \sup(\sigma) = \emptyset$:
     - $\sigma = [x \mapsto y]$
  2. Unify $f(x, \underline{g(y)})$ and $f(y, \underline{x})$ with $x \in \sup(\sigma) = \{x\}$:
     - $\sigma(x) = y \rightsquigarrow$ unify $f(x, \underline{g(y)})$ and $f(y, \underline{y})$.
     - Variable y occurs in $g(y) \rightsquigarrow$ unification fails.

Correctness of the Algorithm

- Theorem: a function call UNIFY($t, u, [\,]$) returns a most general unifier for terms t and u, if such a unifier exists, and fails with an error message, otherwise.
  - Proof sketch: to prove that UNIFY terminates, it suffices to determine a well-founded ordering on its argument tuple (t, u, σ) that is decreased in every recursive call. For this, we define a variable relation $x \to y$ that holds if $x \in \sup(\sigma)$ and $y \in \mathrm{fv}(\sigma(x))$. One can show that every recursive call with argument tuple (t', u', σ') ensures that the transitive closure $\to^+$ is acyclic (because we add a mapping $x \mapsto t\sigma$ respectively $x \mapsto u\sigma$ to σ only if x does not occur in $t\sigma$ respectively $u\sigma$). Then we can deduce the required well-founded ordering from the fact that t' (u') is either a syntactic subterm of t (u) or a term $\sigma(t)$ ($\sigma(u)$) in which every variable y satisfies $t \to^+ y$ ($u \to^+ y$).
    If UNIFY fails with an error message, all the previously tested branch conditions do not hold; from our previous considerations, it is not difficult to establish the correctness of the algorithm in this case. It remains to prove the correctness of UNIFY, if it indeed returns a substitution. For this we denote by $S(\sigma) := \{\langle x, \sigma(x)\rangle \mid x \in \sup(\sigma)\}$ the set of term pairs represented by σ. Now the correctness of each branch that returns a substitution depends on two central properties: first, in every call of UNIFY, its argument σ is already the most general unifier of $S(\sigma)$; second, in every branch, the set of unifiers of $S(\sigma) \cup \{\langle t, u\rangle\}$ equals the set of unifiers of $\bigcup\{S(\sigma') \cup \{\langle t', u'\rangle\} \mid \langle t', u', \sigma'\rangle \in R\}$ where R is the set of all argument tuples of the calls of UNIFY in that branch. We skip the (many) details.

Unification in OCaml

```
let rec istriv env x t = (* decides if env* t = x for reflexive transitive closure env* *)
  match t with           (* fails if adding substitution x|->t to env would lead to a cycle *)
    Var y -> y = x or defined env y & istriv env x (apply env y)
  | Fn(f,args) -> exists (istriv env x) args & failwith "cyclic";;

let rec unify env eqs =  (* result s=[x|->t] is not fully solved: it remains to apply s to t *)
  match eqs with
    [] -> env
  | (Fn(f,fargs),Fn(g,gargs))::oth ->
        if f = g & length fargs = length gargs
        then unify env (zip fargs gargs @ oth)
        else failwith "impossible unification"
  | (Var x,t)::oth | (t,Var x)::oth ->
        if defined env x then unify env ((apply env x,t)::oth)
        else unify (if istriv env x t then env else (x|->t) env) oth;;

let rec solve env = (* computes the fully solved form of substitution env *)
  let env' = mapf (tsubst env) env in
  if env' = env then env else solve env';;
let fullunify eqs = solve (unify undefined eqs);;
```

Unification in OCaml

```
(* WS: convenience function not in Harrison's book *)
let unification_result fm1 fm2 =
  let fv = setify (fvt fm1)@(fvt fm2) in
  let i = fullunify [fm1,fm2] in
  let f x = if defined i x then [x,apply i x] else [] in
  setify (List.concat (map f fv))
;;

# unification_result <<|f(x,y)|>> <<|f(y,x)|>> ;;
- : (string * term) list = [("x", <<|y|>>)]

# unification_result <<|f(x,g(y))|>> <<|f(f(z),w)|>> ;;
- : (string * term) list = [("x", <<|f(z)|>>); ("w", <<|g(y)|>>)]

# unification_result <<|f(x,g(a()),g(z))|>> <<|f(g(y),g(y),g(g(x)))|>> ;;
 - : (string * term) list = [("x", <<|g(a)|>>); ("y", <<|a|>>); ("z", <<|g(g(a))|>>)]

# unification_result <<|f(x,g(y))|>> <<|f(y,x)|>> ;;
Exception: Failure "cyclic".
```

Free-Variable Tableau

Instantiate variables “on demand” by unification (Cohen, Trillin, Wegner, 1974).

$$\frac{\forall x.\ F}{F[x']} \qquad \frac{\exists x.\ F}{F[f(x_1,\ldots,x_n)]} \qquad \frac{\neg\forall x.\ F}{\neg F[f(x_1,\ldots,x_n)]} \qquad \frac{\neg\exists x.\ F}{\neg F[x']}$$

- Similar to the rules for a first-order tableau without unification, except:
  - A fresh variable x'.
  - A fresh n-ary (“Skolem”) function f where $\{x_1,\ldots,x_n\} = \mathrm{fv}(F)\backslash\{x\}$.
    - $n = 0$: Skolem constant c.
- Furthermore, we have the following new closure rule:
  - If literals $p(t_1,\ldots,t_n)$ and $\neg p(t'_1,\ldots,t'_n)$ occur in the same branch of tableaux T where $t_1\sigma = t'_1\sigma,\ldots,t_n\sigma = t'_n\sigma$ for some most general unifier σ, then we may replace T by tableaux $T\sigma$ (in which the corresponding branch is closed).
    - $T\sigma$: identical to T except that every term t in T is replaced by term σt in $T\sigma$.

Formula instances are generated in an “incremental” fashion.

Example

$$(\exists x.\ p(x)) \wedge (\forall x.\ p(x) \Rightarrow \exists y.\ q(x,y)) \Rightarrow \exists x.\ \exists y.\ q(x,y)$$

1.	$(\exists x.\ p(x)) \wedge (\forall x.\ p(x) \Rightarrow \exists y.\ q(x,y)) \wedge \neg\exists x.\ \exists y.\ q(x,y)$	
2.	$(\exists x.\ p(x)) \wedge (\forall x.\ p(x) \Rightarrow \exists y.\ q(x,y))$	(1)
3.	$\neg\exists x.\ \exists y.\ q(x,y)$	(1)
4.	$\neg\exists y.\ q(x_1,y)$	(3)
5.	$\neg q(x_1,y_1)$	(4)
6.	$\exists x.\ p(x)$	(2)
7.	$\forall x.\ p(x) \Rightarrow \exists y.\ q(x,y)$	(2)
8.	$p(c)$	(6)
9.	$p(x_2) \Rightarrow \exists y.\ q(x_2,y)$	(7)

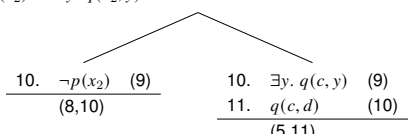


Tableau closed with substitution $[x_2 \mapsto c, x_1 \mapsto c, y_1 \mapsto d]$.

Example

$$(p(a) \vee q(b)) \wedge (\forall x.\ p(x) \Rightarrow r(x)) \wedge (\forall x.\ q(x) \Rightarrow r(f(x))) \Rightarrow \exists x.\ r(x)$$

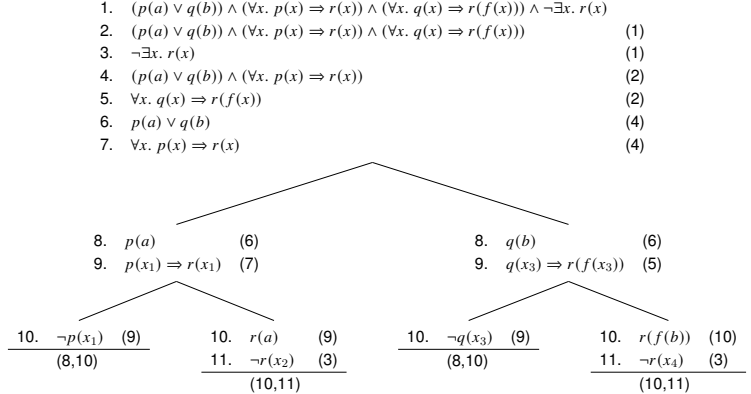


Tableau closed with substitution $[x_1 \mapsto a, x_2 \mapsto a, x_3 \mapsto b, x_4 \mapsto f(b)]$.

Implementation of the Free-Variable Tableaux Method

How to implement a complete search for a closed tableau?

- We require arbitrarily many instantiations of all quantified subformulas.
  - Complex implementation by a repeated traversal of the tableau with a new instantiation of every quantified formula in every branch.
- Assume that we knew in advance a bound n on the maximum number of instantiations required in every branch for a successful refutation.
  - We could simply expand every branch of tableau in a “depth-first” fashion by instantating every quantifier until bound n is reached.
- Iterative Deepening: we repeatedly run the search with bound $n = 0, 1, \ldots$
  - If the root formula is unsatisfiable, we eventually reach the appropriate bound.

Iterative deepening provides a simple implementation strategy.

Free-Variable Tableau in OCaml

The core function essentially computes a DNF in an incremental way.

- `lits`: literals on current branch of the tableau; `fms`: other formulas on current branch.
- `n`: the number of variable instantantations allowed in the branch before giving up.
- `cont`: a function ("continuation") for processing the remaining branches.
- `env`: the substitution computed so far; `k`: a counter for generating fresh variables.

```
let rec tableau (fms,lits,n) cont (env,k) = (* assumes formulas are in Skolem normal form *)
  if n < 0 then failwith "no proof at this level" else
  match fms with
    [] -> failwith "tableau: no proof"
  | And(p,q)::unexp -> tableau (p::q::unexp,lits,n) cont (env,k)
  | Or(p,q)::unexp -> tableau (p::unexp,lits,n) (tableau (q::unexp,lits,n) cont) (env,k)
  | Forall(x,p)::unexp ->
      let y = Var("_" ^ string_of_int k) in
      let p' = subst (x |=> y) p in
      tableau (p'::unexp@[Forall(x,p)],lits,n-1) cont (env,k+1)
  | fm::unexp ->
      try tryfind (fun l -> cont(unify_complements env (fm,l),k)) lits
      with Failure _ -> tableau (unexp,fm::lits,n) cont (env,k);;
```

Free-Variable Tableau in OCaml

```
let rec deepen f n = (* iterative deepending of the proof search *)
  try print_string "Searching with depth limit ";
      print_int n; print_newline(); f n
  with Failure _ -> deepen f (n + 1);;

let tabrefute fms =
  deepen (fun n -> tableau (fms,[],n) (fun x -> x) (undefined,0); n) 0;;

let tab fm =
  let sfm = askolemize(Not(generalize fm)) in
  if sfm = False then 0 else tabrefute [sfm];;

# tab << (P(a) \/ Q(b)) /\ (forall x. P(x) ==> R(x)) /\ (forall x. Q(x) ==> R(f(x)))
    ==> (exists x. R(x)) >>;;
Searching with depth limit 0
Searching with depth limit 1
Searching with depth limit 2
Searching with depth limit 3
- : int = 3
```

Free-Variable Tableau in OCaml

```
(* using prawitz, does not terminate after one minute *)
# let p38 = tab
 <<(forall x.
     P(a) /\ (P(x) ==> (exists y. P(y) /\ R(x,y))) ==>
     (exists z w. P(z) /\ R(x,w) /\ R(w,z))) <=>
   (forall x.
     (~P(a) \/ P(x) \/ (exists z w. P(z) /\ R(x,w) /\ R(w,z))) /\
     (~P(a) \/ ~(exists y. P(y) /\ R(x,y)) \/
     (exists z w. P(z) /\ R(x,w) /\ R(w,z))))>>;;
Searching with depth limit 0
Searching with depth limit 1
Searching with depth limit 2
Searching with depth limit 3
Searching with depth limit 4
val p38 : int = 4
```

Free-Variable Tableau in OCaml

```
(* try to split up the initial formula first; often a big improvement. *)
let splittab fm =
  map tabrefute (simpdnf(askolemize(Not(generalize fm))));;

(* using tab, does not terminate after one minute *)
# let ewd1062 = splittab
 <<(forall x. x <= x) /\
   (forall x y z. x <= y /\ y <= z ==> x <= z) /\
   (forall x y. f(x) <= y <=> x <= g(y))
   ==> (forall x y. x <= y ==> f(x) <= f(y)) /\
       (forall x y. x <= y ==> g(x) <= g(y))>>;;
Searching with depth limit 1
...
Searching with depth limit 9
val ewd1062 : int list = [9; 9]
```

Beneficial if the formula is a propositional composition of closed universally quantified formulas (e.g., “axioms” and “conjectures”).

The Method of Free-Variable Tableaux

The tableaux method has “top down” and “global” characteristics:

- Top-down: The generation of the tableau and the creation of formula instances is driven by the structure of the proof “goal”; formula instances represent knowledge that is relevant in the current context for the derivation of a subgoal.
- Global: When creating two branches from a disjunction

  $$F[x_1] \vee G[x_1]$$

  with free (generated) variable x_1, the method effectively performs a “case split” over a universally quantified disjunction:

  $$\forall x.\ F[x] \vee G[x]$$

  Thus variable x_1 must be instantiated in the same way in all tableau branches.

Next, we will consider another unification-based method with “dual” characteristics.