Hornklauseln und Prolog

Regula Krapf und Maximilian Doré

Kolleg Formale Mathematik

24. März 2016

Inhalt

Tableaux

Wir führen eine Liste von Literalen `lits` und von skolemisierten Formeln `fms` und gehen wie folgt vor:

1. Falls `lits` zwei komplementäre Literale enthält, so sind wir fertig.
2. Sonst wähle eine Formel R in `fms`.

Tableaux

Wir führen eine Liste von Literalen `lits` und von skolemisierten Formeln `fms` und gehen wie folgt vor:

1. Falls `lits` zwei komplementäre Literale enthält, so sind wir fertig.
2. Sonst wähle eine Formel R in `fms`.
   - Falls R ein Literal ist, so fahre fort mit $\{R\} \cup$ `lits`, `fm`$\backslash\{R\}$.

Tableaux

Wir führen eine Liste von Literalen `lits` und von skolemisierten Formeln `fms` und gehen wie folgt vor:

1. Falls `lits` zwei komplementäre Literale enthält, so sind wir fertig.
2. Sonst wähle eine Formel R in `fms`.
   - Falls R ein Literal ist, so fahre fort mit $\{R\} \cup$ `lits`, `fm`$\backslash\{R\}$.
   - Falls R von der Form $P \wedge Q$ ist, so fahre fort mit `lits`, $\{P, Q\} \cup$ `fm`.

Tableaux

Wir führen eine Liste von Literalen `lits` und von skolemisierten Formeln `fms` und gehen wie folgt vor:

1. Falls `lits` zwei komplementäre Literale enthält, so sind wir fertig.
2. Sonst wähle eine Formel R in `fms`.
   - Falls R ein Literal ist, so fahre fort mit $\{R\} \cup$ `lits`, `fm`$\backslash\{R\}$.
   - Falls R von der Form $P \wedge Q$ ist, so fahre fort mit `lits`, $\{P, Q\} \cup$ `fm`.
   - Falls R von der Form $P \vee Q$ ist, so wiederhole das Verfahren für `lits`, $\{P\} \cup$ `fm` und `lits`, $\{Q\} \cup$ `fm`.

Tableaux

Wir führen eine Liste von Literalen `lits` und von skolemisierten Formeln `fms` und gehen wie folgt vor:

1. Falls `lits` zwei komplementäre Literale enthält, so sind wir fertig.
2. Sonst wähle eine Formel R in `fms`.
   - Falls R ein Literal ist, so fahre fort mit $\{R\} \cup$ `lits`, `fm`$\backslash\{R\}$.
   - Falls R von der Form $P \wedge Q$ ist, so fahre fort mit `lits`, $\{P, Q\} \cup$ `fm`.
   - Falls R von der Form $P \vee Q$ ist, so wiederhole das Verfahren für `lits`, $\{P\} \cup$ `fm` und `lits`, $\{Q\} \cup$ `fm`.
   - Falls R von der Form $\forall x.P(x)$ ist, so fahre fort mit `lits`, $\{P(y)\} \cup$ `fm`.

Tableaux

- `n`: Obergrenze für Instanziierungen von ∀-Quantoren
- `cont`: Continuation function (hier: Identität)
- `env`: aktuelle Instanziierung
- `k`: Counter für neue Variablen

```
let rec tableau (fms,lits,n) cont (env,k) =
if n < 0 then failwith "no proof at this levelëlse
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);;
```

Tableaux

Iteratives Vertiefen:

```
let rec deepen f n =
try print_string SSearching with depth limit ";
  print_int n; print_newline(); f n
with Failure _ -> deepen f (n + 1);;
```

Tableaux

Iteratives Vertiefen:

```
let rec deepen f n =
try print_string SSearching with depth limit ";
  print_int n; print_newline(); f n
with Failure _ -> deepen f (n + 1);;
```

Widerlegungsprozedur:

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

Tableaux

Iteratives Vertiefen:

```
let rec deepen f n =
try print_string SSearching with depth limit ";
  print_int n; print_newline(); f n
with Failure _ -> deepen f (n + 1);;
```

Widerlegungsprozedur:

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

Hauptfunktion:

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

Hornklauseln

Definition

Disjunktive Klauseln der Form

- $\neg P_1 \vee \cdots \vee \neg P_n \vee Q \quad (Q \text{ für } n = 0)$
- $\neg P_1 \vee \cdots \vee \neg P_n$

werden *Hornklauseln* genannt. Hornklauseln mit genau einem positiven Literal heißen *definit*.

Wir können Hornklauseln alternativ auch $P_1 \wedge \cdots \wedge P_n \Rightarrow Q$ bzw. $P_1 \wedge \cdots \wedge P_n \Rightarrow \bot$ aufschreiben.

Hornklauseln

Definition

Disjunktive Klauseln der Form

- $\neg P_1 \vee \cdots \vee \neg P_n \vee Q \quad (Q \text{ für } n = 0)$
- $\neg P_1 \vee \cdots \vee \neg P_n$

werden *Hornklauseln* genannt. Hornklauseln mit genau einem positiven Literal heißen *definit*.

Wir können Hornklauseln alternativ auch $P_1 \wedge \cdots \wedge P_n \Rightarrow Q$ bzw. $P_1 \wedge \cdots \wedge P_n \Rightarrow \bot$ aufschreiben.

Beispiel

$(\neg P(0) \vee \neg Q(0)) \wedge (\neg R(0) \vee P(0)) \wedge R(0) \wedge Q(0)$
$\Leftrightarrow (P(0) \wedge Q(0) \Rightarrow \bot) \wedge (R(0) \Rightarrow P(0)) \wedge R(0) \wedge Q(0)$

Reduktion auf definite Hornklauseln

Wir führen ein neues 0-stelliges Prädikat F (für $\bot$) ein. Dann kann man Klauseln der Form

$$\neg P_1 \vee \cdots \vee \neg P_n$$

als

$$\neg P_1 \vee \cdots \vee \neg P_n \vee F$$

umschreiben. Dies ändert nichts an der Erfüllbarkeit.

Reduktion auf definite Hornklauseln

Wir führen ein neues 0-stelliges Prädikat F (für $\bot$) ein. Dann kann man Klauseln der Form

$$\neg P_1 \vee \cdots \vee \neg P_n$$

als

$$\neg P_1 \vee \cdots \vee \neg P_n \vee F$$

umschreiben. Dies ändert nichts an der Erfüllbarkeit.

Beispiel

$(\neg P(0) \vee \neg Q(0)) \wedge (\neg R(0) \vee P(0)) \wedge R(0) \wedge Q(0)$
$\Leftrightarrow (P(0) \wedge Q(0) \Rightarrow F) \wedge (R(0) \Rightarrow P(0)) \wedge R(0) \wedge Q(0)$

Minimale Herbrand-Modelle

Sei S eine Menge von Klauseln. Wir konstruieren eine Herbrand-Interpretation M durch

$$P_M(t_1, \ldots, t_n) = \texttt{true},$$

falls für jedes Herbrand-Modell H von S, $P_H(t_1, \ldots, t_n) = \texttt{true}$. Falls M ein Modell von S ist, so ist es ein *minimales Herbrand-Modell* von S.

Minimale Herbrand-Modelle

Sei S eine Menge von Klauseln. Wir konstruieren eine Herbrand-Interpretation M durch

$$P_M(t_1, \ldots, t_n) = \texttt{true},$$

falls für jedes Herbrand-Modell H von S, $P_H(t_1, \ldots, t_n) = \texttt{true}$.
Falls M ein Modell von S ist, so ist es ein *minimales Herbrand-Modell* von S.

Beispiel

Das minimale Herbrand-Modell von
$(P(0) \wedge Q(0) \Rightarrow R(0)) \wedge R(0)$
erfüllt $R(0)$.

Eigenschaften von Hornklauseln

Korollar

Jede Menge von definiten Hornklauseln ist erfüllbar.

Eigenschaften von Hornklauseln

Korollar

Jede Menge von definiten Hornklauseln ist erfüllbar.

Korollar

Wenn eine Menge von Hornklauseln erfüllbar ist, existiert ein minimales Herbrand-Modell.

Eigenschaften von Hornklauseln

Korollar

Jede Menge von definiten Hornklauseln ist erfüllbar.

Korollar

Wenn eine Menge von Hornklauseln erfüllbar ist, existiert ein minimales Herbrand-Modell.

Korollar

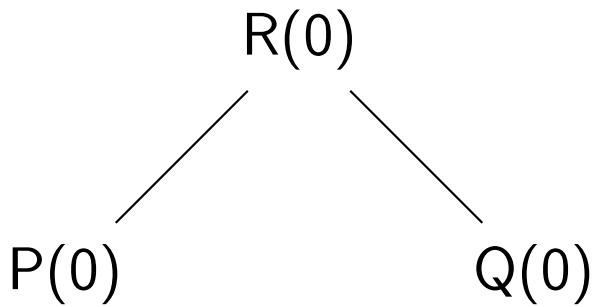
Sei S eine Menge von Hornklauseln und $P[x_1, \ldots, x_n]$ quantorenfrei, so gilt

$$S \models \exists x_1, \ldots, x_n P[x_1, \ldots, x_n] \quad \textit{gdw.} \quad S \models P[t_1, \ldots, t_n]$$

Minimale Herbrand-Modelle von definiten Hornklauseln

Beispiel

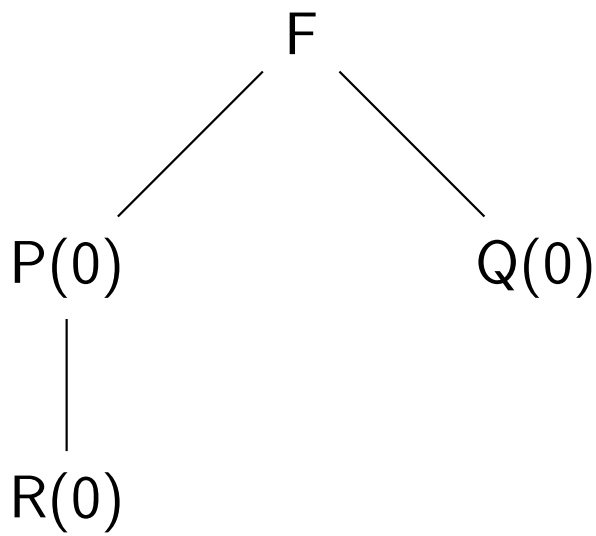
$(P(0) \wedge Q(0) \Rightarrow R(0)) \wedge R(0)$



Beweisbäume

Beispiel

$(P(0) \wedge Q(0) \Rightarrow F) \wedge (R(0) \Rightarrow P(0)) \wedge R(0) \wedge Q(0)$



Hilfsfunktionen

```
let renamerule k (asm,c) =
  let fvs = fv(list_conj(c::asm)) in
  let n = length fvs in
  let vvs = map (fun i -> "_"^ string_of_int i)
    (k -- (k+n-1)) in
  let inst = subst(fpf fvs (map (fun x -> Var x) vvs))
  in (map inst asm,inst c),k+n;;

let hornify cls =
  let pos,neg = partition positive cls in
  if length pos > 1 then failwith "non-Horn clause"
  else (map negate neg,if pos = [] then False else hd
                                                  pos);;
```

Backchaining

```
let rec backchain rules n k env goals =
match goals with
  [] -> env
| g::gs ->
  if n = 0 then failwith "Too deep" else
  tryfind (fun rule ->
    let (a,c),k' = renamerule k rule in
    backchain rules (n - 1) k' (unify_literals env (c,g))
                                          (a @ gs)) rules;;
```

Beweisen mit Hornklauseln

```
let hprove fm =
  let rules = map hornify (simpcnf(fm)) in
  deepen (fun n -> backchain rules n 0 undefined [False],
                                                    n) 0;;
```

Beweisen mit Hornklauseln

```
let hprove fm =
  let rules = map hornify (simpcnf(fm)) in
  deepen (fun n -> backchain rules n 0 undefined [False],
                                                    n) 0;;
```

Beispiel

$(P(0) \wedge Q(0) \Rightarrow F) \wedge (R(0) \Rightarrow P(0)) \wedge R(0) \wedge Q(0)$

Beweisen mit Hornklauseln

```
let hprove fm =
  let rules = map hornify (simpcnf(fm)) in
  deepen (fun n -> backchain rules n 0 undefined [False],
                                                  n) 0;;
```

Beispiel

$(P(0) \wedge Q(0) \Rightarrow F) \wedge (R(0) \Rightarrow P(0)) \wedge R(0) \wedge Q(0)$

Hornifiziert

```
[([<<P(0)>>; <<Q(0)>>], <<false>>);
([<<R(0)>>], <<P(0)>>);
([], <<Q(0)>>);
([], <<R(0)>>)]
```

Prolog

- logische Programmiersprache
- funktioniert durch Backchaining
- Regeln sind der Form

$$\texttt{Q :- P}_1, \ldots, \texttt{P}_n.$$

was für

$$P_1 \wedge \cdots \wedge P_n \rightarrow Q$$

steht.

Prolog

- logische Programmiersprache
- funktioniert durch Backchaining
- Regeln sind der Form

$$\texttt{Q :- P}_1, \ldots, \texttt{P}_n.$$

was für

$$P_1 \wedge \cdots \wedge P_n \rightarrow Q$$

steht.

Beispiel

```
loe(0, X).
loe(s(X), s(Y)) :- loe(X, Y).
```

Parser

Der Parser wandelt Prolog-Regeln in Hornklauseln um:

```
let parserule s =
let c,rest = parse_formula
  (parse_infix_atom,parse_atom) [] (lex(explode s))
in let asm,rest1 = if rest <> [] & hd rest = ":-"then
  parse_list ","(parse_formula
  (parse_infix_atom,parse_atom) []) (tl rest)
  else [],rest in
if rest1 = [] then (asm,c) else failwith " Extra
material after rule";;
```

Simple-Prolog

Simulation von Prolog in Ocaml:

```
let simpleprolog rules gl =
  backchain (map parserule rules) (-1) 0 undefined
                                              [parse gl];;
```

Simple-Prolog

Simulation von Prolog in Ocaml:

```
let simpleprolog rules gl =
  backchain (map parserule rules) (-1) 0 undefined
                                              [parse gl];;
```

Beispiel

```
let lerules = ["0 <= X"; „S(X) <= S(Y) :- X <= Y"];;
```

Simple-Prolog

Simulation von Prolog in Ocaml:

```
let simpleprolog rules gl =
  backchain (map parserule rules) (-1) 0 undefined
                                          [parse gl];;
```

Beispiel

```
let lerules = ["0 <= X"; „S(X) <= S(Y) :- X <= Y"];;
```

Geparst

```
[([], <<0 <= X>>); ([<<X <= Y>>], <<S(X) <= S(Y)>>)]
```

Simple-Prolog

Simulation von Prolog in Ocaml:

```
let simpleprolog rules gl =
  backchain (map parserule rules) (-1) 0 undefined
                                          [parse gl];;
```

Beispiel

```
let lerules = ["0 <= X"; „S(X) <= S(Y) :- X <= Y"];;
```

Geparst

```
[([], <<0 <= X>>); ([<<X <= Y>>], <<S(X) <= S(Y)>>)]
```

Anfragen

```
# simpleprolog lerules „S(S(0)) <= S(S(S(0)))";;
# simpleprolog lerules „S(S(0)) <= S(0)";;
```

Prolog

Auslesen von unifizierten Variablen:

```
let prolog rules gl =
  let i = solve(simpleprolog rules gl) in
  mapfilter (fun x -> Atom(R(-",[Var x; apply i x])))
                                              (fv(parse gl));;
```

Prolog

Auslesen von unifizierten Variablen:

```
let prolog rules gl =
  let i = solve(simpleprolog rules gl) in
  mapfilter (fun x -> Atom(R(-",[Var x; apply i x])))
                                          (fv(parse gl));;
```

Beispiel

```
let lerules = ["0 <= X"; „S(X) <= S(Y) :- X <= Y"];;
# prolog lerules " S(S(0)) <= X";;
```

Backchaining

Beispiel

```
let lerules = ["0 <= X"; „S(X) <= S(Y) :- X <= Y"];;
# prolog lerules " S(S(0)) <= X";;
```

```
let rec backchain rules n k env goals =
match goals with
  [] -> env
| g::gs ->
  if n = 0 then failwith "Too deep" else
  tryfind (fun rule ->
    let (a,c),k' = renamerule k rule in
    backchain rules (n - 1) k' (unify_literals env (c,g))
                                          (a @ gs)) rules;;
```

Backchaining

Beispiel

```
let appendrules = [„append(nil,L,L)";
                   „append(H::T,L,H::A) :- append(T,L,A)"];;
```

Funktioniert Prolog auch für Nicht-Hornklauseln?

Man kann bspw.

$$\{P \vee Q, \neg P, \neg Q\}$$

in eine erfüllbarkeitsäquivalente Menge von Hornklauseln transformieren:

$$\{\neg P' \vee \neg Q', P', Q'\}.$$

Funktioniert Prolog auch für Nicht-Hornklauseln?

Man kann bspw.

$$\{P \vee Q, \neg P, \neg Q\}$$

in eine erfüllbarkeitsäquivalente Menge von Hornklauseln transformieren:

$$\{\neg P' \vee \neg Q', P', Q'\}.$$

Funktioniert aber nicht für

$$\{P \vee Q, P \vee \neg Q, \neg P \vee Q, \neg P \vee \neg Q\}...$$

Kontrapositive

Wir können Klauseln mit n Literalen in n Prolog-Regeln umschreiben. Die Formel

$$P \vee Q \vee \neg R$$

wird dann zu

$$\neg Q \wedge R \to P$$
$$\neg P \wedge R \to Q$$
$$\neg P \wedge \neg Q \to \neg R.$$

Falle Literale negativ sind, führen wir eine zusätzliche Regeln ein: $\neg P \vee \neg Q \vee \neg R$ wird zu

$$P \wedge Q \wedge R \to \bot.$$

Connection Tableaux

Wir geben eine Algorithmus an um Unerfüllbarkeit einer endlichen Menge von propositionalen Klauseln zu testen.

Eingabe: Ein Baum, mit Listen `lits` und `fms` als Knoten und folgender Bedingung:

Connection Tableaux

Wir geben eine Algorithmus an um Unerfüllbarkeit einer endlichen Menge von propositionalen Klauseln zu testen.

Eingabe: Ein Baum, mit Listen `lits` und `fms` als Knoten und folgender Bedingung:

Es gibt eine minimale unerfüllbare Teilmenge von `lits` $\cup$ `fms`, die das vorderste Element von `lits` enthält.

Connection Tableaux

Wir geben eine Algorithmus an um Unerfüllbarkeit einer endlichen Menge von propositionalen Klauseln zu testen.

Eingabe: Ein Baum, mit Listen `lits` und `fms` als Knoten und folgender Bedingung:

Es gibt eine minimale unerfüllbare Teilmenge von `lits` $\cup$ `fms`, die das vorderste Element von `lits` enthält.

1. Falls `lits` $= \emptyset$, wähle eine Klausel C aus `fms` der Form $\overline{P}_1 \vee \cdots \vee \overline{P}_n$ and generiere einen neuen Ast mit Literalen $\{\overline{P}_i\}$ and Formeln `fms`$\backslash\{C\}$.

Connection Tableaux

Wir geben eine Algorithmus an um Unerfüllbarkeit einer endlichen Menge von propositionalen Klauseln zu testen.

Eingabe: Ein Baum, mit Listen `lits` und `fms` als Knoten und folgender Bedingung:

Es gibt eine minimale unerfüllbare Teilmenge von `lits` $\cup$ `fms`, die das vorderste Element von `lits` enthält.

1. Falls `lits` $= \emptyset$, wähle eine Klausel C aus `fms` der Form $\overline{P}_1 \vee \cdots \vee \overline{P}_n$ and generiere einen neuen Ast mit Literalen $\{\overline{P}_i\}$ and Formeln `fms`$\backslash\{C\}$.
2. Falls P das vorderste Element von `lits` ist, suche ein komplementäres Literal $\overline{P}$ und beende den Ast.

Connection Tableaux

Wir geben eine Algorithmus an um Unerfüllbarkeit einer endlichen Menge von propositionalen Klauseln zu testen.

Eingabe: Ein Baum, mit Listen `lits` und `fms` als Knoten und folgender Bedingung:

Es gibt eine minimale unerfüllbare Teilmenge von `lits` $\cup$ `fms`, die das vorderste Element von `lits` enthält.

1. Falls `lits` $= \emptyset$, wähle eine Klausel C aus `fms` der Form $\overline{P}_1 \vee \cdots \vee \overline{P}_n$ and generiere einen neuen Ast mit Literalen $\{\overline{P}_i\}$ and Formeln `fms`$\backslash\{C\}$.
2. Falls P das vorderste Element von `lits` ist, suche ein komplementäres Literal $\overline{P}$ und beende den Ast.
3. Falls $\overline{P}$ nicht in `lits` vorkommt, wähle eine Klausel der Form $C = \overline{P} \vee P_1 \vee \cdots \vee P_n$ in `fms` und generiere für jedes $i \in \{1, \ldots, n\}$ einen Ast mit Literalen $\{P_i\} \cup$ `lits` und Formeln `fms`$\backslash\{C\}$.

Connection Tableaux

- Umformungen (1) - (3) erhalten die gewünschte Eigenschaft.
- Um das Ganze auf FOL zu übertragen, benutzen wir Unifikation, d.h. wenn P das zuletzt hinzugefügte Literal ist, dann suchen wir eine Klausel, die ein Literal enthält, das mit $\neg P$ unifizierbar ist.
- Implementierung mit prolog-artigem Backtracking mit Anfangsgoal $\bot$ und Kontrapositiven als Regeln, d.h. Formel wird zuerst in CNF umgeformt.

Kontrapositive in Prolog:

```
let contrapositives cls =
let base = map (fun c -> map negate
  (subtract cls [c]),c) cls in
if forall negative cls then
  (map negate cls,False)::base else base;;
```

`ancestors`: vorangehende Goals
`g`: aktuelles Goal
`cont`: continuation function (hier: Identität)
`env`: aktuelle Instanziierung
`n`: Counter für maximale Anzahl neuer Knoten im Baum
`k`: Counter für Einführung neuer Variablen

```
let rec mexpand rules ancestors g cont (env,n,k)
=
if n < 0 then failwith "Too deepëlse
try tryfind (fun a -> cont (unify_literals env
  (g,negate a),n,k)) ancestors
with Failure _ -> tryfind
  (fun rule -> let (asm,c),k' = renamerule k rule in
    itlist (mexpand rules (g::ancestors)) asm cont
    (unify_literals env (g,c),n-length asm,k'))
  rules;;
```

MESON

```
let puremeson fm =
let cls = simpcnf(specialize(pnf fm)) in
let rules = itlist ((@) ** contrapositives) cls [] in
deepen (fun n -> mexpand rules [] False (fun x -> x)
  (undefined,n,0); n) 0;;
```

MESON

```
let puremeson fm =
let cls = simpcnf(specialize(pnf fm)) in
let rules = itlist ((@) ** contrapositives) cls [] in
deepen (fun n -> mexpand rules [] False (fun x -> x)
  (undefined,n,0); n) 0;;
```

Hauptfunktion:

```
let meson fm =
let fm1 = askolemize(Not(generalize fm)) in
map (puremeson ** list_conj) (simpdnf fm1);;
```

MESON

Vorteile von MESON:

- Oft effizienter als Tableaux.
- Wenn nur Hornklauseln vorkommen, sind alle Literale positiv. Insbesondere kann man auf Resolution mit vorangehenden Klauseln verzichten.

Nachteil: Erst Umformung in CNF

Optimierungen von MESON

1. Falls ein aktuelles Goal einen identischen Vorfahren hat, so ist es effizienter nur den Vorfahren zu expandieren.

```
let rec equal env fm1 fm2 =
try unify_literals env (fm1,fm2) == env
with Failure _ -> false;;
```

2. Wenn n die Suchgrenze für zwei Subgoals `g1, g2` ist, so kann man die Obergrenze für jedes Subgoal auf $n/2$ setzen.