Démonstration automatique de théorèmes
- Annexe -

Nicolas Daire

base.mli

```
type term = Var of int
          | Fn of string * term list

type atom = P of string * term list

type fol = False
         | True
         | Atom of atom
         | Not of fol
         | And of fol * fol
         | Or of fol * fol
         | Imp of fol * fol
         | Iff of fol * fol
         | Forall of int * fol
         | Exists of int * fol

type literal = L of atom | NL of atom

type clause = literal list

type cnf = clause list

type 'a tree = Tree of 'a * 'a tree list

type 'a tree_l = Tree_l of 'a * 'a tree_l list * float

type dsu = { mutable p : int; mutable r : int }

type node = Nil | V of int | NV of string * int list | T of term

type graph = { mutable n : node; mutable p : int; mutable r : int }

type global = { mutable graph : graph array; mutable max : int; mutable vars : int list }
```

parse.mli

```
open Base

val parse_polish : string -> fol
val parse : string -> fol
```

parse.ml

```
open Base
open Fol_manip
open Str

let parse_bracket p r =
  let f, r = p r in
  match r with | ")"::r -> f, r
               | _ -> failwith "closing bracket expected"

```

```
let rec parse_tuple acc p = function
  | ")"::r -> acc, r
  | r -> let f, r = p r in
    parse_tuple (f::acc) p r

let rec parse_variables h vars p b = function
  | "."::r -> p h vars r
  | n::r -> if not (Hashtbl.mem h n) then Hashtbl.add h n (Hashtbl.length h);
    apply (fun f -> let n = Hashtbl.find h n in if b then Forall (n, f) else Exists (n, f))
      (parse_variables h (n::vars) p b r)
  | _ -> failwith "bound variables"

let rec parse_term h vars = function
  | "("::r -> parse_bracket (parse_term h vars) r
  | n::"("::r -> apply (fun l -> Fn(n, List.rev l))
                  (parse_tuple [] (parse_term h vars) r)
  | n::r -> (if List.mem n vars then Var (Hashtbl.find h n) else Fn(n, [])), r
  | _ -> failwith "parse term";;

let rec parse_fol h vars = function
  | "("::r -> parse_bracket (parse_fol h vars) r
  | "false"::r -> False, r
  | "true"::r -> True, r
  | "and"::"("::r -> apply (fun l ->
      List.fold_left (fun a b -> And (b, a)) (List.hd l) (List.tl l))
      (parse_tuple [] (parse_fol h vars) r)
  | "or"::"("::r -> apply (fun l ->
      List.fold_left (fun a b -> Or (b, a)) (List.hd l) (List.tl l))
      (parse_tuple [] (parse_fol h vars) r)
  | "not"::r -> apply (fun f -> Not f) (parse_fol h vars r)
  | "imp"::"("::r -> apply (fun [a; b] -> Imp (b, a)) (parse_tuple [] (parse_fol h vars) r)
  | "iff"::"("::r -> apply (fun [a; b] -> Iff (b, a)) (parse_tuple [] (parse_fol h vars) r)
  | "forall"::r -> parse_variables h vars parse_fol true r
  | "exists"::r -> parse_variables h vars parse_fol false r
  | n::"("::r -> apply (fun l -> Atom (P(n, List.rev l))) (parse_tuple [] (parse_term h vars) r)
  | n::r -> Atom (P(n, [])), r
  | _ -> failwith "parse : empty";;

let to_string = function
  | Str.Delim a | Str.Text a -> a;;

let parse_string s =
  let p = List.map to_string (Str.full_split (Str.regexp "[ \n\t,()~^.]") s) in
  let p = List.filter (fun s -> not (List.mem s [" "; "\n"; "\t"; ","])) p in
  List.map (fun s -> match String.lowercase_ascii s with
      | "false" -> "false"
      | "true" -> "true"
      | "not" | "~" -> "not"
      | "and" | "/\\" -> "and"
      | "or" | "\\/" -> "or"
      | "imp" | "==>" -> "imp"
      | "iff" | "<=>" -> "iff"
      | "forall" -> "forall"
      | "exists" -> "exists"
      | _ -> s) p;;

let parse_polish f = fst (parse_fol (Hashtbl.create 0) [] (parse_string f));;

let priority = function
  | "imp" | "iff" -> 20
  | "or" -> 30
  | "and" -> 40
```

```
  | "not" -> 50
  | _ -> 0;;

let rec parse_atom h vars = function
  | n::"("::r -> apply (fun l -> Atom (P(n, l))) (parse_tuple [] (parse_term h vars) r)
  | n::r -> Atom (P(n, [])), r
  | _ -> failwith "parse_atom : empty";;

let rec update_stack ops forms p = match ops with
  | [] -> [], forms
  | x::r when priority x <= p -> ops, forms
  | "not"::r -> let f::g = forms in update_stack r (Not(f)::g) p
  | "and"::r -> let f::g::h = forms in update_stack r (And(g,f)::h) p
  | "or"::r -> let f::g::h = forms in update_stack r (Or(g,f)::h) p
  | "imp"::r -> let f::g::h = forms in update_stack r (Imp(g,f)::h) p
  | "iff"::r -> let f::g::h = forms in update_stack r (Iff(g,f)::h) p
  | _ -> failwith "update_stack";;

let rec parse_stack ops forms h vars = function
  | [] | ")"::_ as r -> List.hd (snd (update_stack ops forms (-1))), r
  | "("::r -> let f, r = parse_bracket (parse_stack [] [] h vars) r in
    parse_stack ops (f::forms) h vars r
  | "forall"::r -> let f, r = parse_variables h vars (parse_stack [] []) true r in
    parse_stack ops (f::forms) h vars r
  | "exists"::r -> let f, r = parse_variables h vars (parse_stack [] []) false r in
    parse_stack ops (f::forms) h vars r
  | "false"::r -> parse_stack ops (False::forms) h vars r
  | "true"::r -> parse_stack ops (True::forms) h vars r
  | x::r when List.mem x ["not"; "and"; "or"; "imp"; "iff"] ->
    let ops, forms = update_stack ops forms (priority x) in
    parse_stack (x::ops) forms h vars r
  | n::"("::r -> let l, r = parse_tuple [] (parse_term h vars) r in
    parse_stack ops (Atom(P(n, List.rev l))::forms) h vars r
  | n::r -> parse_stack ops (Atom(P(n, []))::forms) h vars r;;

let parse f = (fst (parse_stack [] [] (Hashtbl.create 0) [] (parse_string f)));;
```

fol_manip.mli

```
open Base

val graph_default : int -> graph
val graph_make : int -> graph array
val global_make : int -> global
val apply : ('a -> 'b) -> ('a * 'c) -> ('b * 'c)
val term_of_atom : atom -> term
val apply_literal : (atom -> atom) -> literal -> literal
val is_literal_positive : literal -> bool
val atom_of_literal : literal -> atom
val negate_literal : literal -> literal
val nf : fol -> fol
val nnf : bool -> fol -> fol
val substitute_term : (int -> term) -> term -> term
val substitute_atom : (int -> term) -> atom -> atom
val substitute_fol : (int -> term) -> fol -> fol
val substitute_literal : (int -> term) -> literal -> literal
val substitute_clause : (int -> term) -> clause -> clause
val max_variable_fol : fol -> int
val max_variable_clause : clause -> int
val non_variable_count_clause : clause -> int
val rename : fol -> fol
val prenex : fol -> fol
val skolemization : fol -> fol
val rem_quantifiers : fol -> fol
val distribute : fol -> fol
val convert_to_cnf : fol -> cnf
```

```ocaml
open Base

let graph_default i = { n = Nil; p = i; r = -1}

let graph_make n = Array.init n graph_default

let global_make n = { graph = graph_make n; max = -1; vars = [] }

let apply f (s, r) = (f s, r)

let term_of_atom = function
  | P (n, l) -> Fn (n, l)

let apply_literal f = function
  | L p -> L (f p)
  | NL p -> NL (f p)

let is_literal_positive = function
  | L _ -> true;
  | NL _ -> false

let atom_of_literal = function
  | L p | NL p -> p

let negate_literal = function
  | L p -> NL p
  | NL p -> L p

let rec nf = function
  | Not f -> Not (nf f)
  | And (a, b) -> And (nf a, nf b)
  | Or (a, b) -> Or (nf a, nf b)
  | Imp (a, b) -> Or (Not (nf a), nf b)
  | Iff (a, b) -> let a, b = nf a, nf b in Or (And (a, b), And (Not a, Not b))
  | Forall (v, f) -> Forall (v, nf f)
  | Exists (v, f) -> Exists (v, nf f)
  | f -> f

let rec nnf neg = function
  | False -> if neg then True else False
  | True -> if neg then False else True
  | Atom a -> if neg then Not (Atom a) else Atom a
  | Not f -> nnf (not neg) f
  | And (a, b) -> if neg then Or (nnf true a, nnf true b)
    else And (nnf false a, nnf false b)
  | Or (a, b) -> if neg then And (nnf true a, nnf true b)
    else Or (nnf false a, nnf false b)
  | Imp (a, b) -> if neg then And (nnf false a, nnf true b)
    else Or (nnf true a, nnf false b)
  | Iff (a, b) -> if neg then And (Or (nnf true a, nnf true b), Or (nnf false a, nnf false b))
    else Or (And (nnf false a, nnf false b), And (nnf true a, nnf true b))
  | Forall (v, f) -> if neg then Exists (v, nnf true f)
    else Forall (v, nnf false f)
  | Exists (v, f) -> if neg then Forall (v, nnf true f)
    else Exists (v, nnf false f)

let rec eliminate_triv = function
  | False -> False
  | True -> True
  | Not f -> (match eliminate_triv f with
      | True -> False
      | False -> True
      | f -> Not f)
  | And (f, g) -> (match eliminate_triv f, eliminate_triv g with
```

```
      | False, _ | _, False -> False
      | True, f | f, True -> f
      | f, g -> And (f, g))
  | Or (f, g) -> (match eliminate_triv f, eliminate_triv g with
      | True, _ | _, True -> True
      | False, f | f, False -> f
      | f, g -> Or (f, g))
  | Imp (f, g) -> (match eliminate_triv f, eliminate_triv g with
      | True, f -> f
      | f, False -> Not f
      | False, _ | _, True -> True
      | f, g -> Imp (f, g))
  | Iff (f, g) -> (match eliminate_triv f, eliminate_triv g with
      | True, f | f, True -> f
      | False, _ | _, False -> False
      | f, g -> Iff (f, g))
  | Forall (v, f) -> (match eliminate_triv f with
      | False | True as f -> f
      | f -> Forall (v, f))
  | Exists (v, f) -> (match eliminate_triv f with
      | False | True as f -> f
      | f -> Exists (v, f))
  | f -> f

let rec substitute_term s = function
  | Var x -> s x
  | Fn (n, l) -> Fn (n, List.map (substitute_term s) l)

let rec substitute_atom s = function
  | P (n, l) -> P (n, List.map (substitute_term s) l)

let rec substitute_fol s = function
  | False -> False
  | True -> True
  | Atom a -> Atom (substitute_atom s a)
  | Not f -> Not (substitute_fol s f)
  | And (f, g) -> And (substitute_fol s f, substitute_fol s g)
  | Or (f, g) -> Or (substitute_fol s f, substitute_fol s g)
  | Imp (f, g) -> Imp (substitute_fol s f, substitute_fol s g)
  | Iff (f, g) -> Iff (substitute_fol s f, substitute_fol s g)
  | Forall (v, f) -> Forall (v, substitute_fol s f)
  | Exists (v, f) -> Exists (v, substitute_fol s f)

let substitute_literal s = apply_literal (substitute_atom s)

let substitute_clause s =
  List.map (substitute_literal s)

let substitute_of_hashtbl s =
  fun x -> if Hashtbl.mem s x then Hashtbl.find s x else Var x

let rec max_variable_fol = function
  | Forall (v, f) | Exists (v, f) -> max v (max_variable_fol f)
  | Not f -> max_variable_fol f
  | And (f, g) | Or (f, g) | Imp (f, g) | Iff (f, g) ->
    max (max_variable_fol f) (max_variable_fol g)
  | _ -> -1

let rec max_variable_term = function
  | Var x -> x
  | Fn (_, l) -> List.fold_left (fun a b -> max a (max_variable_term b)) (-1) l

let rec max_variable_atom p = max_variable_term (term_of_atom p)

let rec max_variable_clause = function
  | [] -> -1
```

```
  | p::l -> max (max_variable_atom (atom_of_literal p)) (max_variable_clause l)

let rec non_variable_count_term = function
  | Var _ -> 0
  | Fn (_, l) -> List.fold_left (fun a b -> non_variable_count_term b + a) 1 l

let non_variable_count_atom p = non_variable_count_term (term_of_atom p)

let rec non_variable_count_clause = function
  | [] -> 0
  | p::l -> (non_variable_count_atom (atom_of_literal p)) + (non_variable_count_clause l)

let rec rename_term rewrite = function
  | Var v -> Var (Hashtbl.find rewrite v)
  | Fn (f, l) -> Fn (f, List.map (rename_term rewrite) l)

let rec rename_atom rewrite = function
  | P (n, l) -> P (n, List.map (rename_term rewrite) l)

let rename f =
  let c = ref 0 in
  let rewrite = Hashtbl.create 0 in
  let rec aux = function
    | False -> False
    | True -> True
    | Atom a -> Atom (rename_atom rewrite a)
    | Not f -> Not (aux f)
    | And (a, b) -> let a = aux a and b = aux b in And (a, b)
    | Or (a, b) -> let a = aux a and b = aux b in Or (a, b)
    | Imp (a, b) -> let a = aux a and b = aux b in Imp (a, b)
    | Iff (a, b) -> let a = aux a and b = aux b in Iff (a, b)
    | Forall (v, f) -> let d = !c in incr c; Hashtbl.add rewrite v d; Forall (d, aux f)
    | Exists (v, f) -> let d = !c in incr c; Hashtbl.add rewrite v d; Exists (d, aux f)
      in
  aux f;;

let skolem_name n = "S#" ^ (string_of_int n)

let prenex f =
  let f = rename (nnf false f) in
  let n = max_variable_fol f + 1 in
  let r = Array.make n (-1) in
  let rec aux = function
    | Not f -> let p, f = aux f in
      List.map (fun (b, v) -> not b, v) p, Not f
    | And (f, g) -> let pf, f = aux f and pg, g = aux g in
      prefix true pf pg, And (f, g)
    | Or (f, g) -> let pf, f = aux f and pg, g = aux g in
      prefix false pf pg, Or (f, g)
    | Forall (v, f) -> let p, f = aux f in
      if List.exists (fun (_, w) -> v = w) p then p, f
      else (true, v)::p, f
    | Exists (v, f) -> let p, f = aux f in
      if List.exists (fun (_, w) -> v = w) p then p, f
      else (false, v)::p, f
    | f -> [], f
  and prefix b pf pg = match pf, pg with
    | [], p | p, [] -> p
    | (bf, vf)::rf, (bg, vg)::rg when bf = b && bg = b ->
      r.(vg) <- vf ; (b, vf)::prefix b rf rg
    | (bf, vf)::rf, _ when bf <> b -> (bf, vf)::prefix b rf pg
    | _, (bg, vg)::rg when bg <> b ->(bg, vg)::prefix b pf rg
  and convert f = function
    | [] -> f
    | (true, v)::p -> Forall (r.(v), convert f p)
    | (false, v)::p -> Exists (r.(v), convert f p) in
```

```
  let p, f = aux f in
  let c = ref 0 in
  for i = 0 to n - 1 do
    if r.(i) = -1 then (r.(i) <- !c; incr c)
    else r.(i) <- r.(r.(i))
  done;
  let s = (fun i -> Var r.(i)) in
  convert (substitute_fol s f) p

let skolemization f =
  let f = prenex f in
  let n = max_variable_fol f + 1 in
  let c = ref 0 in
  let skolem = Array.init n (fun i -> Var i) in
  let skolem_variable vars v =
    skolem.(v) <- (Fn(skolem_name !c, List.rev vars)); incr c in
  let rec aux vars = function
    | Forall (v, f) -> Forall (v, aux (Var v::vars) f)
    | Exists (v, f) -> skolem_variable vars v; aux vars f
    | f -> substitute_fol (fun i -> skolem.(i)) f in
  aux [] f

let rem_quantifiers f =
  let rec aux = function
    | Forall (_, f) -> aux f
    | f -> f in
  aux (skolemization f)

let distribute f =
  let rec aux = function
    | And (a, b) -> And (aux a, aux b)
    | Or (a, b) -> (match aux a, aux b with
        | And (c, d), e -> And (aux (Or (c, e)), aux (Or (d, e)))
        | c, And (d, e) -> And (aux (Or(c, d)), aux (Or(c, e)))
        | _ -> Or (a, b))
    | f -> f in
  aux (rem_quantifiers f)

let convert_to_cnf f =
  let rec clause c = function
    | Atom a -> (L a)::c
    | Not (Atom a) -> (NL a)::c
    | Or (a, b) -> clause (clause c a) b
    | f -> failwith "error" in
  let rec aux l = function
    | And (a, b) -> aux (aux l a) b
    | f -> (List.rev (clause [] f))::l in
  List.rev (aux [] (eliminate_triv (distribute f)))
```

disp.mli

```
open Base

val print_hashtbl : (int, term) Hashtbl.t -> unit
val print : fol -> unit
val tree_of_fol : bool -> fol -> string tree
val layout_compact : string tree -> string tree_l
val disp_layout : float -> int -> float -> fol -> unit
val print_cnf : cnf -> unit
```

disp.ml

```
open Base
open Fol_manip
open Graphics

```

```
let parenthesize print x =
  print_string "(";
  print x;
  print_string ")"

let rec print_term = function
  | Var v -> print_string "V"; print_int v
  | Fn (f, l) -> print_string f;
    if l <> [] then parenthesize print_term_list l
and print_term_list = function
  | [] -> ()
  | [p] -> print_term p
  | h::t -> print_term h; print_string " "; print_term_list t

let print_atom p = print_term (term_of_atom p)

let rec print = function
  | False -> print_string "FALSE"
  | True -> print_string "TRUE"
  | Atom a -> print_atom a
  | Not (Atom a) -> print_string "NOT "; print_atom a
  | Not f -> print_string "NOT"; parenthesize print f
  | And (f, g) -> parenthesize print f;
    print_string " AND ";
    parenthesize print g
  | Or (f, g) -> parenthesize print f;
    print_string " OR ";
    parenthesize print g
  | Imp (f, g) -> parenthesize print f;
    print_string " IMP ";
    parenthesize print g
  | Iff (f, g) -> parenthesize print f;
    print_string " IFF ";
    parenthesize print g
  | Forall (v, f) -> print_string ("FORALL V" ^ (string_of_int v) ^ ".");
    print f
  | Exists (v, f) -> print_string ("EXISTS V" ^ (string_of_int v) ^ ".");
    print f

let rec tree_of_term b = function
  | Var v -> Tree ((if b then Printf.sprintf "$v_%d$" v
                   else Printf.sprintf "V%d" v), [])
  | Fn (f, l) -> Tree ((if b then Printf.sprintf "$%s$"
                           (Str.global_replace (Str.regexp "#") "_"
                           (Str.global_replace (Str.regexp "_") "\_" f))
                       else f), List.map (tree_of_term b) l)

let tree_of_atom b p = tree_of_term b (term_of_atom p)

let rec tree_of_fol b = function
  | False -> Tree ((if b then "$\\bot$" else "false"), [])
  | True -> Tree ((if b then "$\\top$" else "true"), [])
  | Atom a -> tree_of_atom b a
  | Not f -> Tree ((if b then "$\\neg$" else "not"), [tree_of_fol b f])
  | And (f, g) -> Tree ((if b then "$\\wedge$" else "and"), [tree_of_fol b f; tree_of_fol
     b g])
  | Or (f, g) -> Tree ((if b then "$\\vee$" else "or"), [tree_of_fol b f; tree_of_fol b g
    ])
  | Imp (f, g) -> Tree ((if b then "$\\Rightarrow$" else "imp"),
                       [tree_of_fol b f; tree_of_fol b g])
  | Iff (f, g) -> Tree ((if b then "$\\Leftrightarrow$" else "iff"),
                       [tree_of_fol b f; tree_of_fol b g])
  | Forall (v, f) -> Tree ((if b then Printf.sprintf "$\\forall v_%d$" v
                           else Printf.sprintf "forall V%d" v), [tree_of_fol b f])
  | Exists (v, f) -> Tree ((if b then Printf.sprintf "$\\exists v_%d$" v
                           else Printf.sprintf "exists V%d" v), [tree_of_fol b f])
```

```
let rec tree_height (Tree (_, l)) = List.fold_left (fun a b -> max a (tree_height b)) (-1) l + 1

let layout_compact t =
  let rec min_l m = function
    | [] -> m
    | (l, _)::t -> min_l (min l m) t
  and merge a b d = match a, b with
    | [], [] -> []
    | e, [] -> e
    | [], (lb, rb)::tb -> (lb+.d, rb+.d)::(merge [] tb d)
    | (la, _)::ta, (_, rb)::tb -> (la, rb+.d)::(merge ta tb d)
  and dist a b = match a,b with
    | e, [] | [], e -> 0.
    | (_, lr)::ta, (rl, _)::tb -> max (1.+.lr-.rl) (dist ta tb)
  and move d = function Tree_l (v, t, x) -> Tree_l (v, t, x+.d)
  and center = function
    | [] -> 0.
    | (l, r)::_ -> (l+.r)*.0.5
  and prop t e = function
    | [] -> let c = center e in
      List.rev (List.map (move (-.c)) t),
      List.map (function l, r -> l-.c, r-.c) e
    | h::u -> let h, eh = aux h in
      let dt = dist e eh in
      prop ((move dt h)::t) (merge e eh dt) u
  and aux = function
    | Tree (v, []) -> Tree_l (v, [], 0.), [(0., 0.)]
    | Tree (v, t) ->
      let t, e = prop [] [] t in
      Tree_l (v, t, 0.), (0., 0.)::e in
  let layout, e = aux t in
  move (-.(min_l 0. e)) layout

let disp_layout s h o f =
  open_graph " 1600x720";
  let t = tree_of_fol false f in
  let n = tree_height t in
  let rec aux x0 m = function
    | Tree_l (v,t,x) ->
      moveto (int_of_float (s*.(x0+.x))) (3*h*m);
      draw_string v;
      moveto (int_of_float (s*.(x0+.x))) (h*(3*m+1));
      if m < n then
        lineto (int_of_float (s*.x0)) (h*(3*(m+1)));
      List.iter (aux (x0+.x) (m-1)) t in
  aux o n (layout_compact t)

let save file string =
  let channel = open_out file in
  output_string channel string;
  close_out channel;;

let latex_of_fol f name =
  let rec aux b (Tree_l (n, l, x)) =
    let s = Printf.sprintf "\\begin{scope}[shift={({%f*\\treel},{-\\treehh})}]\n\
                            \\node [above] at (0,0) {%s};\n" x n ^
        (List.fold_right (fun a b -> aux true a ^ b) l "\\end{scope}\n") in
    if b then Printf.sprintf "\\draw (0,0) -- ({%f*\\treel},{-\\treeh});\n" x ^ s else s in
  let s = "\\begin{tikzpicture}\n\\begin{scope}[scale={\\trees}]\n" ^
          (aux false (layout_compact (tree_of_fol true f))) ^
          "\\end{scope}\n\\end{tikzpicture}\n" in
  save (Printf.sprintf "tree_%s.tex" name) s;;
```

```
let print_hashtbl = Hashtbl.iter (fun x y -> print_string ("V" ^ (string_of_int x) ^ " ->
    ");
                                   print_term y;
                                   print_newline ())

let print_literal = function
  | L p -> print_string "L "; print_atom p
  | NL p -> print_string "NL "; print_atom p

let rec print_clause = function
  | [] -> ()
  | [p] -> print_literal p
  | p::t -> parenthesize (fun _ -> print_literal p; print_string " OR "; print_clause t)
     ()

let rec print_cnf = function
  | [] -> ()
  | [c] -> print_clause c
  | c::t -> parenthesize (fun _ -> print_clause c; print_string " AND "; print_cnf t) ()
```

resolution.mli

```
open Base

val unify_naive : atom -> atom -> ((int, term) Hashtbl.t * bool)

val unify_terms : global -> term -> term -> bool

val pack_clause : global -> clause -> clause

val simplify_clause : clause -> clause

val subsumes : graph array -> clause -> clause -> bool

val tautology : clause -> bool

val resolve : global -> clause list -> literal -> clause -> (literal * bool) list ->
    clause list

val resolve_binary : global -> clause list -> clause -> clause -> clause -> clause ->
    clause list

val resolution_step : global -> clause list -> clause -> clause list -> (clause list *
    clause list)

val resolution : global -> clause list -> (clause list * clause list)

val resolution_process : fol -> (bool * (clause list * clause list))
```

resolution.ml

```
open Base
open Fol_manip

let rec occurs_check s x = function
  | Var y -> y = x || (Hashtbl.mem s y && occurs_check s x (Hashtbl.find s y))
  | Fn (_, l) -> List.exists (occurs_check s x) l

let unify_find_naive s = function
  | Var u when Hashtbl.mem s u -> Hashtbl.find s u
  | x -> x

let rec unify_var_naive s p q = match unify_find_naive s p, unify_find_naive s q with
  | Var u, Var v when u = v -> true
  | Var u, v | v, Var u ->
    if Hashtbl.mem s u then unify_var_naive s (Hashtbl.find s u) v
```

```
    else if not (occurs_check s u v) then
        (Hashtbl.add s u v; true)
    else false
  | Fn (fp, lp), Fn (fq, lq) -> fp = fq && unify_list_naive s lp lq
and unify_list_naive s lp lq =
  List.length lp = List.length lq && List.for_all2 (unify_var_naive s) lp lq

let reconstruct_naive s =
  let t = Hashtbl.create (Hashtbl.length s) in
  let rec aux = function
    | Var x -> if not (Hashtbl.mem t x) then
        Hashtbl.add t x (aux (match Hashtbl.find_opt s x with Some y -> y | None -> Var x
            ));
      Hashtbl.find t x
    | Fn (f, l) -> Fn (f, List.map aux l) in
  Hashtbl.iter (fun a _ -> ignore (aux (Var a))) s;
  t

let unify_naive p q =
  let s = Hashtbl.create 0 in
  let s, b = (match p, q with
      | P (np, lp), P (nq, lq) -> s, np = nq && unify_list_naive s lp lq) in
  if b then
    reconstruct_naive s, b
  else
    s, b

let rec dsu_find g x =
  if g.(x).p <> x then g.(x).p <- dsu_find g g.(x).p;
  g.(x).p

let dsu_union g a b =
  let a, b = dsu_find g a, dsu_find g b in
  if a <> b then
    begin
      if g.(a).r > g.(b).r then
        g.(b).p <- a
      else if g.(a).r < g.(b).r then
        g.(a).p <- b
      else
        begin
          g.(b).p <- a;
          g.(a).r <- g.(a).r + 1
        end
    end

let rec vars_from_term g = function
  | Var x -> if g.graph.(x).r = -1 then
      begin
        g.max <- max g.max (x + 1);
        g.graph.(x).r <- 0;
        g.vars <- x::g.vars
      end
  | Fn (_ , l) -> List.iter (vars_from_term g) l

let rec graph_from_term g = function
  | Var x -> x
  | Fn (f, l) -> g.max <- g.max + 1; let d = g.max in g.graph.(d).r <- 0;
    g.graph.(d).n <- NV (f, List.map (graph_from_term g) l); d

let unify_find g x =
  let x = dsu_find g x in
  match g.(x) with
  | {n=V y} -> dsu_find g y
  | _ -> x
```

```
let rec unify g p q =
  let p, q = unify_find g p, unify_find g q in
  p = q || match g.(p).n, g.(q).n with
  | NV (u, r), NV (v, s) -> dsu_union g p q; u = v && unify_list g r s
  | _, NV _ -> g.(p).n <- V q; true
  | NV _, _ -> g.(q).n <- V p; true
  | _, _ -> dsu_union g p q; true
and unify_list g r s =
  List.length r = List.length s && List.for_all2 (unify g) r s

let acyclic g =
  let next x =
    let x = dsu_find g x in
    match g.(x) with
    | {n=V y} -> y
    | _ -> x in
  let rec dfs x =
    if g.(x).r >= 0 then
      begin
        g.(x).r <- -1;
        let b = (match g.(x).n with
            | NV (_, l) -> List.for_all dfs l
            | _ -> let y = next x in x = y || dfs y) in
        g.(x).r <- -2; b
      end
    else g.(x).r = -2 in
  List.for_all dfs

let rec reconstruct g =
  let next x =
    let x = dsu_find g.graph x in
    match g.graph.(x) with
    | { n = V y} -> y
    | _ -> x in
  let rec aux x = match g.graph.(x).n with
    | T t -> t
    | NV (f, l) -> let t = Fn (f, List.map aux l) in
      g.graph.(x).n <- T t; t
    | _ -> let y = next x in
      if y = x then
        Var x
      else
        (let t = aux y in
         g.graph.(x).n <- T t; t) in
  List.iter (fun x -> ignore (aux x)) g.vars

let rec unify_fast p q = match p, q with
  | Var _, _ | _, Var _ -> true
  | Fn (f, r), Fn (g, s) -> f = g && unify_list_fast r s
and unify_list_fast r s =
  List.length r = List.length s && List.for_all2 unify_fast r s

let unify_terms g p q =
  g.max <- -1; g.vars <- [];
  vars_from_term g p; vars_from_term g q;
  let r = graph_from_term g p and s = graph_from_term g q in
  let b = unify g.graph r s && acyclic g.graph g.vars in
  if b then
    reconstruct g
  else
    (List.iter (fun x -> g.graph.(x) <- graph_default x) g.vars; g.vars <- []);
  for i = r to g.max do
    g.graph.(i) <- graph_default i
  done;
  g.max <- -1;
  b
```

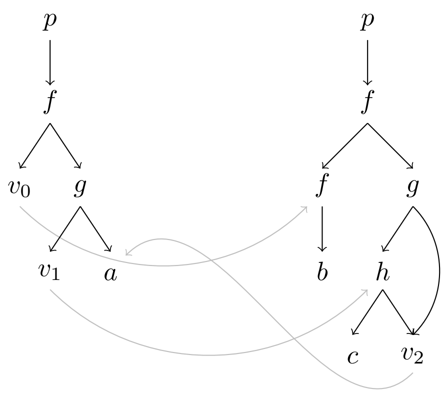
```
let unify_bool g p q =
  g.max <- -1; g.vars <- [];
  vars_from_term g p; vars_from_term g q;
  let r = graph_from_term g p and s = graph_from_term g q in
  let b = unify g.graph r s && acyclic g.graph g.vars in
  List.iter (fun x -> g.graph.(x) <- graph_default x) g.vars;
  g.vars <- [];
  for i = r to g.max do
    g.graph.(i) <- graph_default i
  done;
  g.max <- -1;
  b

let unify_atom_bool g p q = match p, q with
  | P (m, r), P (n, s) -> unify_bool g (Fn (m, r)) (Fn (n, s))

let unify_atoms g p q =
  let p, q = term_of_atom p, term_of_atom q in
  unify_terms g p q

let unify_routine g p q b =
  let p, q = term_of_atom p, term_of_atom q in
  unify_fast p q &&
  begin
    if b then
      begin
        let p = substitute_term (fun x -> Var (2 * x)) p in
        let q = substitute_term (fun x -> Var (2 * x + 1)) q in
        unify_terms g p q
      end
    else
      unify_terms g p q
  end

let rec pack_term g = function
  | Var x -> (match g.graph.(x).n with
      | T t -> t
      | _ -> g.max <- g.max + 1; g.vars <- x::g.vars;
        g.graph.(x).n <- T (Var g.max); Var g.max)
  | Fn (f, l) -> Fn (f, List.map (pack_term g) l)

let pack_atom g = function
  | P (n, l) -> P (n, List.map (pack_term g) l)

let pack_literal g = apply_literal (pack_atom g)

let pack_clause g c =
  g.max <- -1; g.vars <- [];
  let c = List.map (pack_literal g) c in
  List.iter (fun x -> g.graph.(x) <- graph_default x) g.vars;
  g.vars <- []; g.max <- -1;
  c

let rec simplify_clause = function
  | [] -> []
  | h::t -> if List.mem h t then
      simplify_clause t
    else
      h::simplify_clause t

let rec subsumes_unify g vars p q = match p, q with
  | Var x, t -> (match g.(x).n with
      | Nil -> g.(x).n <- T t; vars := x::!vars; true
      | T u -> t = u
      | _ -> false)
```

```
  | Fn (f, r), Fn (h, s) when f = h && List.length r = List.length s ->
    List.for_all2 (subsumes_unify g vars) r s
  | _ -> false

let subsumes_atom g vars p q =
  subsumes_unify g vars (term_of_atom p) (term_of_atom q)

let subsumes_literal g vars p q = match p, q with
  | L p, L q | NL p, NL q -> subsumes_atom g vars p q
  | _ -> false

let rec subsumes g cp cq = match cp with
  | [] -> true
  | p::tp -> List.exists (fun q ->
      let vars = ref [] in
      let b = subsumes_literal g vars p q && subsumes g tp cq in
      List.iter (fun x -> g.(x).n <- Nil) !vars; b) cq

let rec tautology = function
  | [] -> false
  | h::t -> List.mem (negate_literal h) t || tautology t

let rec replace g v = function
  | [] -> [v]
  | h::t when subsumes g v h -> v::List.filter (fun p -> not (subsumes g v p)) t
  | h::t -> h::replace g v t

let insert g u t w =
  if tautology w ||
     List.exists (fun p -> subsumes g p w) t ||
     List.exists (fun p -> subsumes g p w) u then t
  else replace g w t

let rec resolve g acc u h = function
  | [] -> let h = simplify_clause h in
    let h = pack_clause g h in
    let m = max_variable_clause h in
    let n = non_variable_count_clause h in
    if 2*(m+n+1) > Array.length g.graph then
      g.graph <- graph_make (4*(m+n+1));
    h::acc
  | (p, b)::t -> if b = (is_literal_positive p = is_literal_positive u) &&
                   unify_routine g (atom_of_literal p) (atom_of_literal u) false then
      begin
        let s = (fun x -> match g.graph.(x).n with T t -> t | _ -> Var x) in
        let h2 = substitute_clause s h in
        let t2 = List.map (apply (substitute_literal s)) t in
        let u2 = substitute_literal s u in
        List.iter (fun x -> g.graph.(x) <- graph_default x) g.vars;
        g.vars <- [];
        resolve g (resolve g acc u2 h2 t2) u (p::h) t
      end
    else
      resolve g acc u (p::h) t

let rec resolve_binary g acc hp tp hq tq = match tp, tq with
  | [], _ -> acc
  | p::tp, [] -> resolve_binary g acc (p::hp) tp [] hq
  | p::tp, q::tq -> if is_literal_positive p <> is_literal_positive q &&
                      unify_routine g (atom_of_literal p) (atom_of_literal q) true then
      begin
        let s = (fun x -> match g.graph.(2*x).n with T t -> t | _ -> Var (2*x)) in
        let t = (fun x -> match g.graph.(2*x+1).n with T t -> t | _ -> Var (2*x+1)) in
        let u = substitute_literal s p in
        let h = substitute_clause s hp@substitute_clause t hq in
        let t = List.map (fun p -> p, true) (substitute_clause s tp)@
```

```
                List.map (fun p -> p, false) (substitute_clause t tq) in
        List.iter (fun x -> g.graph.(x) <- graph_default x) g.vars;
        g.vars <- [];
        resolve_binary g (resolve g acc u h t) hp (p::tp) (q::hq) tq
      end
    else
      resolve_binary g acc hp (p::tp) (q::hq) tq

let resolution_step g u v t =
  let p = List.for_all is_literal_positive v in
  let w = List.fold_left (fun a b ->
      if p || List.for_all is_literal_positive b
      then resolve_binary g a [] v [] b
      else a) [] u in
  v::u, List.fold_left (insert g.graph (v::u)) t w

let resolution g =
  let rec aux u = function
    | h::t when not (List.mem [] u) ->
      print_int (List.length u); print_string ";\t";
      print_int (List.length t + 1); print_string ";\t";
      print_int (Array.length g.graph); print_newline ();
      let a, b = resolution_step g u h t in
      aux a b
    | v -> u, v in
  aux []

let preprocess g f =
  List.fold_left (insert g []) [] (List.map simplify_clause f)

let resolution_process f =
  let f = convert_to_cnf f in
  let n = List.fold_left (fun a b ->
      max ((max_variable_clause b) + (non_variable_count_clause b)) a) 0 f in
  let g = global_make (4 * (n + 1)) in
  let u = preprocess g.graph f in
  let a, b = resolution g u in
  List.mem [] a, (a, b)
```

Unification du système $\{(f(v_0, g(v_1, a)), f(f(b), g(h(c, v_2), v_2))\}$, dont un unificateur principal est $[v_0 \to f(b), v_1 \to h(c, a), v_2 \to a]$.

```
# let t1 = Fn ("f", [Var 0; Fn ("g", [Var 1; Fn ("a", [])])]);;

val t1 : Base.term = Fn ("f", [Var 0; Fn ("g", [Var 1; Fn ("a", [])])])

# let t2 = Fn ("f", [Fn ("f", [Fn ("b", [])]); Fn ("g", [Fn ("h", [Fn ("c", []); Var 2]);
    Var 2])]);;

val t2 : Base.term =
  Fn ("f",
   [Fn ("f", [Fn ("b", [])]);
    Fn ("g", [Fn ("h", [Fn ("c", []); Var 2]); Var 2])])

# let g = global_make 16;;

val g : Base.global =
  {graph =
    [|{n = Nil; p = 0; r = -1}; {n = Nil; p = 1; r = -1};
      {n = Nil; p = 2; r = -1}; {n = Nil; p = 3; r = -1};
      {n = Nil; p = 4; r = -1}; {n = Nil; p = 5; r = -1};
      {n = Nil; p = 6; r = -1}; {n = Nil; p = 7; r = -1};
      {n = Nil; p = 8; r = -1}; {n = Nil; p = 9; r = -1};
      {n = Nil; p = 10; r = -1}; {n = Nil; p = 11; r = -1};
      {n = Nil; p = 12; r = -1}; {n = Nil; p = 13; r = -1};
      {n = Nil; p = 14; r = -1}; {n = Nil; p = 15; r = -1}|];
   max = -1; vars = []}

# unify_terms g t1 t2;;

- : bool = true

# g;;

- : Base.global =
{graph =
  [|{n = T (Fn ("f", [Fn ("b", [])])); p = 0; r = -2};
    {n = T (Fn ("h", [Fn ("c", []); Fn ("a", [])])); p = 1; r = -2};
    {n = T (Fn ("a", [])); p = 2; r = -2}; {n = Nil; p = 3; r = -1};
    {n = Nil; p = 4; r = -1}; {n = Nil; p = 5; r = -1};
    {n = Nil; p = 6; r = -1}; {n = Nil; p = 7; r = -1};
    {n = Nil; p = 8; r = -1}; {n = Nil; p = 9; r = -1};
    {n = Nil; p = 10; r = -1}; {n = Nil; p = 11; r = -1};
    {n = Nil; p = 12; r = -1}; {n = Nil; p = 13; r = -1};
    {n = Nil; p = 14; r = -1}; {n = Nil; p = 15; r = -1}|];
 max = -1; vars = [2; 1; 0]}

# let s = (fun x -> match g.graph.(x).n with T t -> t | _ -> Var x);;

val s : int -> Base.term = <fun>

# substitute_term s t1;;

- : Base.term =
Fn ("f",
 [Fn ("f", [Fn ("b", [])]);
  Fn ("g", [Fn ("h", [Fn ("c", []); Fn ("a", [])]); Fn ("a", [])])])

# substitute_term s t2;;

- : Base.term =
Fn ("f",
 [Fn ("f", [Fn ("b", [])]);
  Fn ("g", [Fn ("h", [Fn ("c", []); Fn ("a", [])]); Fn ("a", [])])])
```


p
f
v0
g
v1
a
p
f
f
g
b
h
c
v2

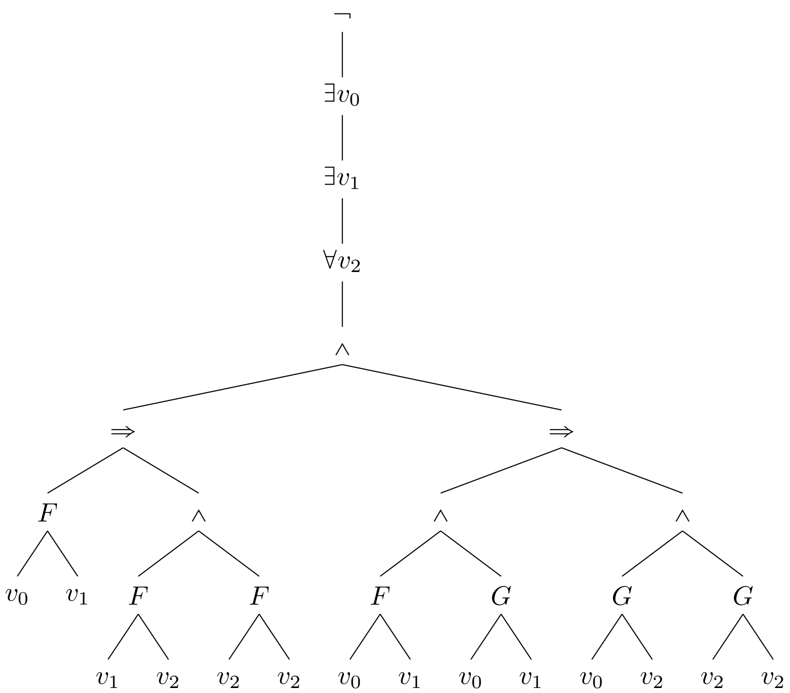
On teste la méthode de résolution sur la formule suivante :

$$\exists x \exists y \forall z (F(x,y) \Rightarrow (F(y,z) \wedge F(z,z))) \wedge ((F(x,y) \Rightarrow G(x,y)) \Rightarrow (G(x,z) \wedge G(z,z)))$$

interpréteur

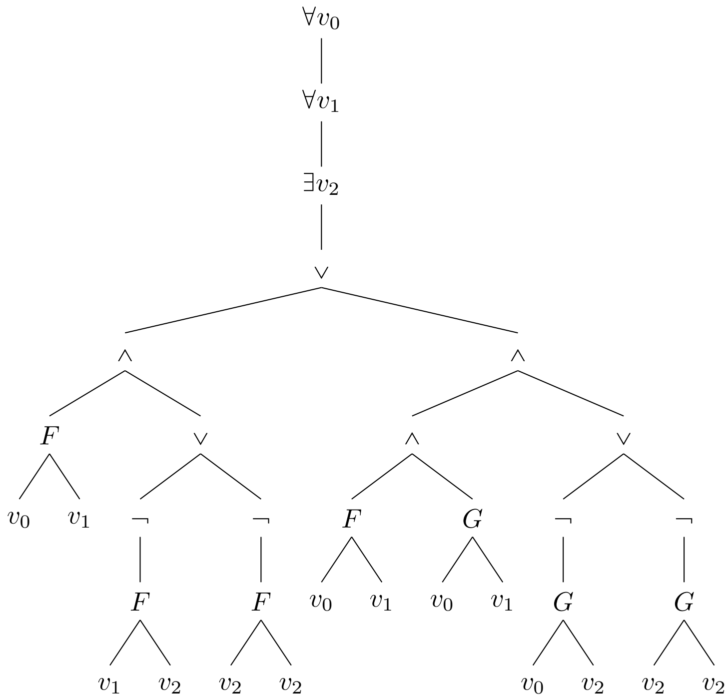
```
# let f = parse "~(exists x. exists y. forall z.
        (F(x,y) ==> (F(y,z) /\\ F(z,z))) /\\
        ((F(x,y) /\\ G(x,y)) ==> (G(x,z) /\\ G(z,z))))";;

    val f : Base.fol =
  Not
   (Exists (0,
     Exists (1,
      Forall (2,
       And
        (Imp (Atom (P ("F", [Var 0; Var 1])),
          And (Atom (P ("F", [Var 1; Var 2])),
           Atom (P ("F", [Var 2; Var 2])))),
        Imp
         (And (Atom (P ("F", [Var 0; Var 1])),
           Atom (P ("G", [Var 0; Var 1]))),
         And (Atom (P ("G", [Var 0; Var 2])), Atom (P ("G", [Var 2; Var 2])))))))))

# let t = tree_of_fol true f;;

val t : string Base.tree =
  Tree ("$\\neg$",
   [Tree ("$\\exists v_0$",
     [Tree ("$\\exists v_1$",
       [Tree ("$\\forall v_2$",
         [Tree ("$\\wedge$",
           [Tree ("$\\Rightarrow$",
             [Tree ("F", [Tree ("$v_0$", []); Tree ("$v_1$", [])]);
              Tree ("$\\wedge$",
               [Tree ("F", [Tree ("$v_1$", []); Tree ("$v_2$", [])]);
                Tree ("F", [Tree ("$v_2$", []); Tree ("$v_2$", [])])])]);
            Tree ("$\\Rightarrow$",
             [Tree ("$\\wedge$",
               [Tree ("F", [Tree ("$v_0$", []); Tree ("$v_1$", [])]);
                Tree ("G", [Tree ("$v_0$", []); Tree ("$v_1$", [])])]);
              Tree ("$\\wedge$",
               [Tree ("G", [Tree ("$v_0$", []); Tree ("$v_2$", [])]);
                Tree ("G", [Tree ("$v_2$", []); Tree ("$v_2$", [])])])])])])])])])

# let tl = layout_compact t;;

val tl : string Base.tree_l =
  Tree_l ("$\\neg$",
   [Tree_l ("$\\exists v_0$",
     [Tree_l ("$\\exists v_1$",
       [Tree_l ("$\\forall v_2$",
         [Tree_l ("$\\wedge$",
           [Tree_l ("$\\Rightarrow$",
             [Tree_l ("F",
               [Tree_l ("$v_0$", [], -0.5); Tree_l ("$v_1$", [], 0.5)],
               -1.25);
              Tree_l ("$\\wedge$",
               [Tree_l ("F",
                 [Tree_l ("$v_1$", [], -0.5); Tree_l ("$v_2$", [], 0.5)],
                 -1.);
                Tree_l ("F",
                 [Tree_l ("$v_2$", [], -0.5); Tree_l ("$v_2$", [], 0.5)], 1.)],
               1.25)],
             -3.625);
            Tree_l ("$\\Rightarrow$",
```

```
             [Tree_l ("$\\wedge$",
               [Tree_l ("F",
                 [Tree_l ("$v_0$", [], -0.5); Tree_l ("$v_1$", [], 0.5)],
                 -1.);
                Tree_l ("G",
                 [Tree_l ("$v_0$", [], -0.5); Tree_l ("$v_1$", [], 0.5)], 1.)],
               -2.);
              Tree_l ("$\\wedge$",
               [Tree_l ("G",
                 [Tree_l ("$v_0$", [], -0.5); Tree_l ("$v_2$", [], 0.5)],
                 -1.);
                Tree_l ("G",
                 [Tree_l ("$v_2$", [], -0.5); Tree_l ("$v_2$", [], 0.5)], 1.)],
               2.)],
             3.625)],
           0.)],
         0.)],
       0.)],
     0.)],
   5.375)
```

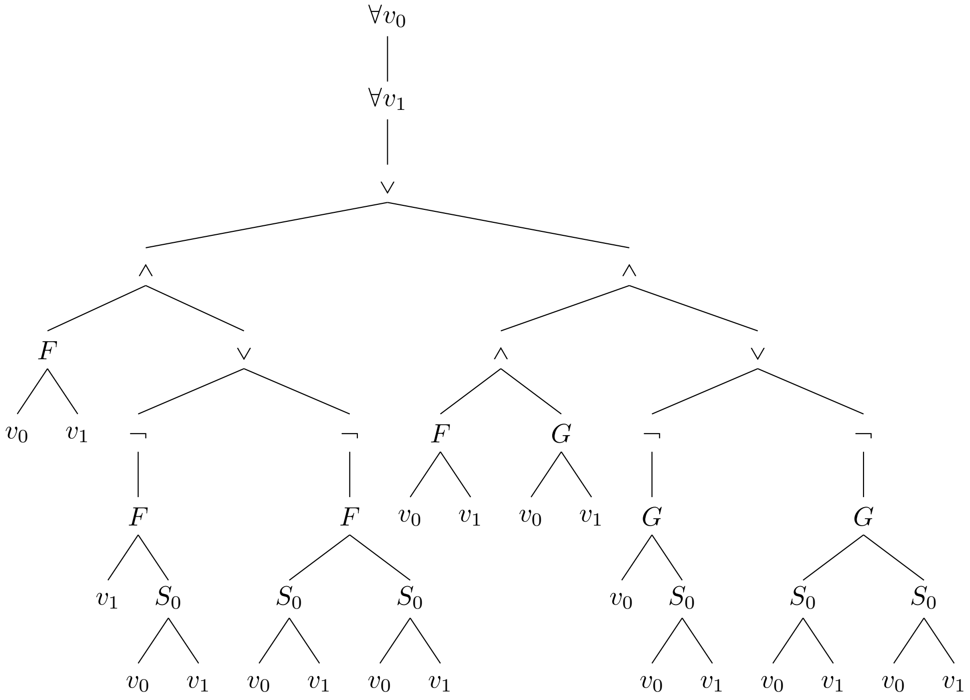



```
# nnf false f;;

- : Base.fol =
Forall (0,
 Forall (1,
  Exists (2,
   Or
    (And (Atom (P ("F", [Var 0; Var 1])),
      Or (Not (Atom (P ("F", [Var 1; Var 2]))),
       Not (Atom (P ("F", [Var 2; Var 2]))))),
    And
     (And (Atom (P ("F", [Var 0; Var 1])), Atom (P ("G", [Var 0; Var 1]))),
     Or (Not (Atom (P ("G", [Var 0; Var 2]))),
      Not (Atom (P ("G", [Var 2; Var 2])))))))))
\end{listlisting}

\input{tree_nnf.tex}

\begin{lstlisting}[name=interp]
# prenex f;;

```

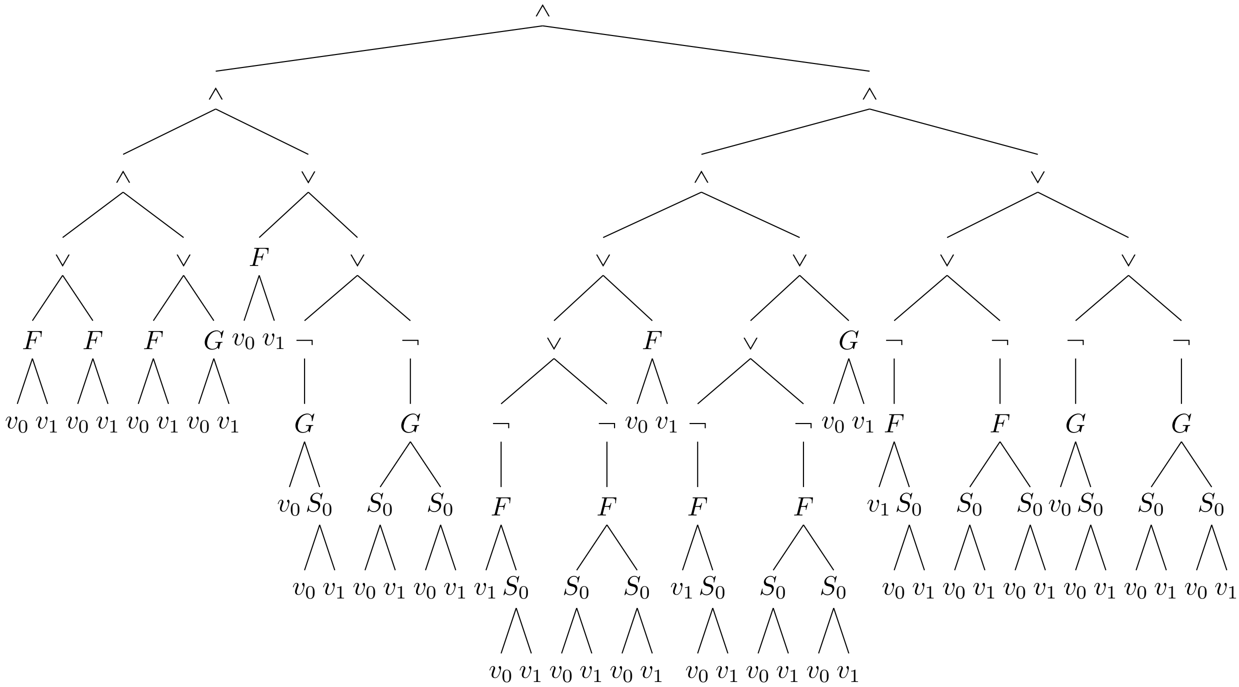
```
- : Base.fol =
Forall (0,
 Forall (1,
  Exists (2,
   Or
    (And (Atom (P ("F", [Var 0; Var 1])),
      Or (Not (Atom (P ("F", [Var 1; Var 2]))),
       Not (Atom (P ("F", [Var 2; Var 2]))))),
    And
     (And (Atom (P ("F", [Var 0; Var 1])), Atom (P ("G", [Var 0; Var 1]))),
     Or (Not (Atom (P ("G", [Var 0; Var 2]))),
      Not (Atom (P ("G", [Var 2; Var 2])))))))))
```




```
# skolemization f;;

- : Base.fol =
Forall (0,
 Forall (1,
  Or
   (And (Atom (P ("F", [Var 0; Var 1])),
     Or (Not (Atom (P ("F", [Var 1; Fn ("S#0", [Var 0; Var 1])]))),
      Not
       (Atom
         (P ("F", [Fn ("S#0", [Var 0; Var 1]); Fn ("S#0", [Var 0; Var 1])]))))),
   And (And (Atom (P ("F", [Var 0; Var 1])), Atom (P ("G", [Var 0; Var 1]))),
    Or (Not (Atom (P ("G", [Var 0; Fn ("S#0", [Var 0; Var 1])]))),
     Not
      (Atom
        (P ("G", [Fn ("S#0", [Var 0; Var 1]); Fn ("S#0", [Var 0; Var 1])]))))))))
```



```
# distribute f;;

- : Base.fol =
And
 (And
   (And (Or (Atom (P ("F", [Var 0; Var 1])), Atom (P ("F", [Var 0; Var 1]))),
     Or (Atom (P ("F", [Var 0; Var 1])), Atom (P ("G", [Var 0; Var 1])))),
   Or (Atom (P ("F", [Var 0; Var 1])),
    Or (Not (Atom (P ("G", [Var 0; Fn ("S#0", [Var 0; Var 1])]))),
     Not
      (Atom
        (P ("G", [Fn ("S#0", [Var 0; Var 1]); Fn ("S#0", [Var 0; Var 1])])))))),
 And
  (And
    (Or
      (Or (Not (Atom (P ("F", [Var 1; Fn ("S#0", [Var 0; Var 1])]))),
        Not
         (Atom
           (P ("F", [Fn ("S#0", [Var 0; Var 1]); Fn ("S#0", [Var 0; Var 1])])))),
      Atom (P ("F", [Var 0; Var 1]))),
    Or
     (Or (Not (Atom (P ("F", [Var 1; Fn ("S#0", [Var 0; Var 1])]))),
       Not
        (Atom
          (P ("F", [Fn ("S#0", [Var 0; Var 1]); Fn ("S#0", [Var 0; Var 1])])))),
     Atom (P ("G", [Var 0; Var 1])))),
  Or
   (Or (Not (Atom (P ("F", [Var 1; Fn ("S#0", [Var 0; Var 1])]))),
     Not
      (Atom
        (P ("F", [Fn ("S#0", [Var 0; Var 1]); Fn ("S#0", [Var 0; Var 1])])))),
   Or (Not (Atom (P ("G", [Var 0; Fn ("S#0", [Var 0; Var 1])]))),
    Not
     (Atom
       (P ("G", [Fn ("S#0", [Var 0; Var 1]); Fn ("S#0", [Var 0; Var 1])])))))))
```



```
# let c = convert_to_cnf f;;

val c : Base.cnf =
  [[L (P ("F", [Var 0; Var 1])); L (P ("F", [Var 0; Var 1]))];
   [L (P ("F", [Var 0; Var 1])); L (P ("G", [Var 0; Var 1]))];
   [L (P ("F", [Var 0; Var 1]));
    NL (P ("G", [Var 0; Fn ("S#0", [Var 0; Var 1])]));
    NL (P ("G", [Fn ("S#0", [Var 0; Var 1]); Fn ("S#0", [Var 0; Var 1])]))];
   [NL (P ("F", [Var 1; Fn ("S#0", [Var 0; Var 1])]));
    NL (P ("F", [Fn ("S#0", [Var 0; Var 1]); Fn ("S#0", [Var 0; Var 1])]));
    L (P ("F", [Var 0; Var 1]))];
   [NL (P ("F", [Var 1; Fn ("S#0", [Var 0; Var 1])]));
    NL (P ("F", [Fn ("S#0", [Var 0; Var 1]); Fn ("S#0", [Var 0; Var 1])]));
    L (P ("G", [Var 0; Var 1]))];
   [NL (P ("F", [Var 1; Fn ("S#0", [Var 0; Var 1])]));
    NL (P ("F", [Fn ("S#0", [Var 0; Var 1]); Fn ("S#0", [Var 0; Var 1])]));
    NL (P ("G", [Var 0; Fn ("S#0", [Var 0; Var 1])]));
    NL (P ("G", [Fn ("S#0", [Var 0; Var 1]); Fn ("S#0", [Var 0; Var 1])]))]]

# let g = global_make 10;;

val g : Base.global =
  {graph =
    [|{n = Nil; p = 0; r = -1}; {n = Nil; p = 1; r = -1};
      {n = Nil; p = 2; r = -1}; {n = Nil; p = 3; r = -1};
      {n = Nil; p = 4; r = -1}; {n = Nil; p = 5; r = -1};
      {n = Nil; p = 6; r = -1}; {n = Nil; p = 7; r = -1};
      {n = Nil; p = 8; r = -1}; {n = Nil; p = 9; r = -1}|];
   max = -1; vars = []}

# let u::v::w::x::y::z = c;;

# resolve_binary g [] [] u [] x;;

- : Base.literal list list =
[[L (P ("F", [Var 0; Fn ("S#0", [Var 1; Var 0])]));
  NL (P ("F", [Fn ("S#0", [Var 1; Var 0]); Fn ("S#0", [Var 1; Var 0])]));
  L (P ("F", [Var 1; Var 0]))];
 [NL (P ("F", [Var 0; Fn ("S#0", [Var 1; Var 0])]));
  L (P ("F", [Fn ("S#0", [Var 1; Var 0]); Fn ("S#0", [Var 1; Var 0])]));
  L (P ("F", [Var 1; Var 0]))];
```

```
 [L (P ("F", [Var 0; Var 1]));
  L (P ("F", [Fn ("S#0", [Var 0; Var 1]); Fn ("S#0", [Var 0; Var 1])]));
  NL (P ("F", [Var 1; Fn ("S#0", [Var 0; Var 1])]))];
 [L (P ("F", [Var 0; Var 1]));
  NL (P ("F", [Var 1; Fn ("S#0", [Var 0; Var 1])]))];
 [L (P ("F", [Var 0; Var 1]));
  NL (P ("F", [Fn ("S#0", [Var 0; Var 1]); Fn ("S#0", [Var 0; Var 1])]));
  L (P ("F", [Var 1; Fn ("S#0", [Var 0; Var 1])]))];
 [L (P ("F", [Var 0; Var 1]));
  NL (P ("F", [Fn ("S#0", [Var 0; Var 1]); Fn ("S#0", [Var 0; Var 1])]))]]

# resolution_step g [u] v (w::x::y::z);;

- : Base.clause list * Base.clause list =
([[L (P ("F", [Var 0; Var 1])); L (P ("G", [Var 0; Var 1]))];
  [L (P ("F", [Var 0; Var 1])); L (P ("F", [Var 0; Var 1]))]],
 [[L (P ("F", [Var 0; Var 1]));
   NL (P ("G", [Var 0; Fn ("S#0", [Var 0; Var 1])]));
   NL (P ("G", [Fn ("S#0", [Var 0; Var 1]); Fn ("S#0", [Var 0; Var 1])]))];
  [NL (P ("F", [Var 1; Fn ("S#0", [Var 0; Var 1])]));
   NL (P ("F", [Fn ("S#0", [Var 0; Var 1]); Fn ("S#0", [Var 0; Var 1])]));
   L (P ("F", [Var 0; Var 1]))];
  [NL (P ("F", [Var 1; Fn ("S#0", [Var 0; Var 1])]));
   NL (P ("F", [Fn ("S#0", [Var 0; Var 1]); Fn ("S#0", [Var 0; Var 1])]));
   L (P ("G", [Var 0; Var 1]))];
  [NL (P ("F", [Var 1; Fn ("S#0", [Var 0; Var 1])]));
   NL (P ("F", [Fn ("S#0", [Var 0; Var 1]); Fn ("S#0", [Var 0; Var 1])]));
   NL (P ("G", [Var 0; Fn ("S#0", [Var 0; Var 1])]));
   NL (P ("G", [Fn ("S#0", [Var 0; Var 1]); Fn ("S#0", [Var 0; Var 1])]))]])

# resolution_process f;;

0;      3;      48
1;      2;      48
2;      3;      48
3;      4;      48
4;      3;      48
5;      4;      48
6;      3;      48
7;      2;      48
8;      1;      48
- : bool * (Base.clause list * Base.clause list) =
(true,
 ([[]; [NL (P ("G", [Var 0; Fn ("S#0", [Var 0; Var 1])]))];
   [NL (P ("G", [Var 0; Fn ("S#0", [Var 0; Var 1])]));
    NL (P ("G", [Fn ("S#0", [Var 0; Var 1]); Fn ("S#0", [Var 0; Var 1])]))];
   [NL (P ("G", [Fn ("S#0", [Var 0; Var 1]); Fn ("S#0", [Var 0; Var 1])]));
    NL (P ("G", [Var 0; Fn ("S#0", [Var 0; Var 1])]));
    NL (P ("F", [Var 1; Fn ("S#0", [Var 0; Var 1])]))];
   [L (P ("G", [Var 0; Var 1]))];
   [L (P ("G", [Var 0; Var 1]));
    NL (P ("F", [Var 1; Fn ("S#0", [Var 0; Var 1])]))];
   [NL (P ("F", [Var 1; Fn ("S#0", [Var 0; Var 1])]));
    NL (P ("F", [Fn ("S#0", [Var 0; Var 1]); Fn ("S#0", [Var 0; Var 1])]));
    NL (P ("G", [Var 0; Fn ("S#0", [Var 0; Var 1])]));
    NL (P ("G", [Fn ("S#0", [Var 0; Var 1]); Fn ("S#0", [Var 0; Var 1])]))];
   [NL (P ("F", [Var 1; Fn ("S#0", [Var 0; Var 1])]));
    NL (P ("F", [Fn ("S#0", [Var 0; Var 1]); Fn ("S#0", [Var 0; Var 1])]));
    L (P ("G", [Var 0; Var 1]))];
   [L (P ("F", [Var 0; Var 1]))]],
  []))
```