

Démonstration automatique de théorèmes

- Annexe -

Nicolas Daire

base.mli

```
1 type term = Var of int
2           | Fn of string * term list
3
4 type atom = P of string * term list
5
6 type fol = False
7         | True
8         | Atom of atom
9         | Not of fol
10        | And of fol * fol
11        | Or of fol * fol
12        | Imp of fol * fol
13        | Iff of fol * fol
14        | Forall of int * fol
15        | Exists of int * fol
16
17 type literal = L of atom | NL of atom
18
19 type clause = literal list
20
21 type cnf = clause list
22
23 type 'a tree = Tree of 'a * 'a tree list
24
25 type 'a tree_l = Tree_l of 'a * 'a tree_l list * float
26
27 type dsu = { mutable p : int; mutable r : int }
28
29 type node = Nil | V of int | NV of string * int list | T of term
30
31 type graph = { mutable n : node; mutable p : int; mutable r : int }
32
33 type global = { mutable graph : graph array; mutable max : int; mutable vars : int list }
```

parse.mli

```
1 open Base
2
3 val parse_polish : string -> fol
4 val parse : string -> fol
```

parse.ml

```
1 open Base
2 open Fol_manip
3 open Str
4
5 let parse_bracket p r =
6   let f, r = p r in
7   match r with | ")"::r -> f, r
8               | _ -> failwith "closing bracket expected"
9
```

```

10 let rec parse_tuple acc p = function
11   | ")"::r -> acc, r
12   | r -> let f, r = p r in
13     parse_tuple (f::acc) p r
14
15 let rec parse_variables h vars p b = function
16   | "."::r -> p h vars r
17   | n::r -> if not (Hashtbl.mem h n) then Hashtbl.add h n (Hashtbl.length h);
18     apply (fun f -> let n = Hashtbl.find h n in if b then Forall (n, f) else Exists (n, f)
19       ))
20     (parse_variables h (n::vars) p b r)
21   | _ -> failwith "bound variables"
22
23 let rec parse_term h vars = function
24   | "("::r -> parse_bracket (parse_term h vars) r
25   | n::"("::r -> apply (fun l -> Fn(n, List.rev l))
26     (parse_tuple [] (parse_term h vars) r)
27   | n::r -> (if List.mem n vars then Var (Hashtbl.find h n) else Fn(n, [])), r
28   | _ -> failwith "parse term";;
29
30 let rec parse_fol h vars = function
31   | "("::r -> parse_bracket (parse_fol h vars) r
32   | "false"::r -> False, r
33   | "true"::r -> True, r
34   | "and"::"("::r -> apply (fun l ->
35     List.fold_left (fun a b -> And (b, a)) (List.hd l) (List.tl l))
36     (parse_tuple [] (parse_fol h vars) r)
37   | "or"::"("::r -> apply (fun l ->
38     List.fold_left (fun a b -> Or (b, a)) (List.hd l) (List.tl l))
39     (parse_tuple [] (parse_fol h vars) r)
40   | "not"::r -> apply (fun f -> Not f) (parse_fol h vars r)
41   | "imp"::"("::r -> apply (fun [a; b] -> Imp (b, a)) (parse_tuple [] (parse_fol h vars)
42     r)
43   | "iff"::"("::r -> apply (fun [a; b] -> Iff (b, a)) (parse_tuple [] (parse_fol h vars)
44     r)
45   | "forall"::r -> parse_variables h vars parse_fol true r
46   | "exists"::r -> parse_variables h vars parse_fol false r
47   | n::"("::r -> apply (fun l -> Atom (P(n, List.rev l))) (parse_tuple [] (parse_term h
48     vars) r)
49   | n::r -> Atom (P(n, [])), r
50   | _ -> failwith "parse : empty";;
51
52 let to_string = function
53   | Str.Delim a | Str.Text a -> a;;
54
55 let parse_string s =
56   let p = List.map to_string (Str.full_split (Str.regexp "[ \\n\\t,()~^.]") s) in
57   let p = List.filter (fun s -> not (List.mem s [" "; "\\n"; "\\t"; ",",])) p in
58   List.map (fun s -> match String.lowercase_ascii s with
59     | "false" -> "false"
60     | "true" -> "true"
61     | "not" | "~" -> "not"
62     | "and" | "\\\\" -> "and"
63     | "or" | "\\/" -> "or"
64     | "imp" | "==" -> "imp"
65     | "iff" | "<=" -> "iff"
66     | "forall" -> "forall"
67     | "exists" -> "exists"
68     | _ -> s) p;;
69
70 let parse_polish f = fst (parse_fol (Hashtbl.create 0) [] (parse_string f));;
71
72 let priority = function
73   | "imp" | "iff" -> 20
74   | "or" -> 30
75   | "and" -> 40

```

```

72 | "not" -> 50
73 | _ -> 0;;
74
75 let rec parse_atom h vars = function
76 | n::"("::r -> apply (fun l -> Atom (P(n, l))) (parse_tuple [] (parse_term h vars) r)
77 | n::r -> Atom (P(n, [])), r
78 | _ -> failwith "parse_atom : empty";;
79
80 let rec update_stack ops forms p = match ops with
81 | [] -> [], forms
82 | x::r when priority x <= p -> ops, forms
83 | "not"::r -> let f::g = forms in update_stack r (Not(f)::g) p
84 | "and"::r -> let f::g::h = forms in update_stack r (And(g,f)::h) p
85 | "or"::r -> let f::g::h = forms in update_stack r (Or(g,f)::h) p
86 | "imp"::r -> let f::g::h = forms in update_stack r (Imp(g,f)::h) p
87 | "iff"::r -> let f::g::h = forms in update_stack r (Iff(g,f)::h) p
88 | _ -> failwith "update_stack";;
89
90 let rec parse_stack ops forms h vars = function
91 | [] | ")"::_ as r -> List.hd (snd (update_stack ops forms (-1))), r
92 | "("::r -> let f, r = parse_bracket (parse_stack [] [] h vars) r in
93   parse_stack ops (f::forms) h vars r
94 | "forall"::r -> let f, r = parse_variables h vars (parse_stack [] []) true r in
95   parse_stack ops (f::forms) h vars r
96 | "exists"::r -> let f, r = parse_variables h vars (parse_stack [] []) false r in
97   parse_stack ops (f::forms) h vars r
98 | "false"::r -> parse_stack ops (False::forms) h vars r
99 | "true"::r -> parse_stack ops (True::forms) h vars r
100 | x::r when List.mem x ["not"; "and"; "or"; "imp"; "iff"] ->
101   let ops, forms = update_stack ops forms (priority x) in
102   parse_stack (x::ops) forms h vars r
103 | n::"("::r -> let l, r = parse_tuple [] (parse_term h vars) r in
104   parse_stack ops (Atom(P(n, List.rev l))::forms) h vars r
105 | n::r -> parse_stack ops (Atom(P(n, []))::forms) h vars r;;
106
107 let parse f = (fst (parse_stack [] [] (Hashtbl.create 0) [] (parse_string f)));;

```

fol_manip.mli

```

1 open Base
2
3 val graph_default : int -> graph
4 val graph_make : int -> graph array
5 val global_make : int -> global
6 val apply : ('a -> 'b) -> ('a * 'c) -> ('b * 'c)
7 val term_of_atom : atom -> term
8 val apply_literal : (atom -> atom) -> literal -> literal
9 val is_literal_positive : literal -> bool
10 val atom_of_literal : literal -> atom
11 val negate_literal : literal -> literal
12 val nf : fol -> fol
13 val nnf : bool -> fol -> fol
14 val substitute_term : (int -> term) -> term -> term
15 val substitute_atom : (int -> term) -> atom -> atom
16 val substitute_fol : (int -> term) -> fol -> fol
17 val substitute_literal : (int -> term) -> literal -> literal
18 val substitute_clause : (int -> term) -> clause -> clause
19 val max_variable_fol : fol -> int
20 val max_variable_clause : clause -> int
21 val non_variable_count_clause : clause -> int
22 val rename : fol -> fol
23 val prenex : fol -> fol
24 val skolemization : fol -> fol
25 val rem_quantifiers : fol -> fol
26 val distribute : fol -> fol
27 val convert_to_cnf : fol -> cnf

```

```

1  open Base
2
3  let graph_default i = { n = Nil; p = i; r = -1}
4
5  let graph_make n = Array.init n graph_default
6
7  let global_make n = { graph = graph_make n; max = -1; vars = [] }
8
9  let apply f (s, r) = (f s, r)
10
11 let term_of_atom = function
12   | P (n, l) -> Fn (n, l)
13
14 let apply_literal f = function
15   | L p -> L (f p)
16   | NL p -> NL (f p)
17
18 let is_literal_positive = function
19   | L _ -> true;
20   | NL _ -> false
21
22 let atom_of_literal = function
23   | L p | NL p -> p
24
25 let negate_literal = function
26   | L p -> NL p
27   | NL p -> L p
28
29 let rec nf = function
30   | Not f -> Not (nf f)
31   | And (a, b) -> And (nf a, nf b)
32   | Or (a, b) -> Or (nf a, nf b)
33   | Imp (a, b) -> Or (Not (nf a), nf b)
34   | Iff (a, b) -> let a, b = nf a, nf b in Or (And (a, b), And (Not a, Not b))
35   | Forall (v, f) -> Forall (v, nf f)
36   | Exists (v, f) -> Exists (v, nf f)
37   | f -> f
38
39 let rec nnf neg = function
40   | False -> if neg then True else False
41   | True -> if neg then False else True
42   | Atom a -> if neg then Not (Atom a) else Atom a
43   | Not f -> nnf (not neg) f
44   | And (a, b) -> if neg then Or (nnf true a, nnf true b)
45     else And (nnf false a, nnf false b)
46   | Or (a, b) -> if neg then And (nnf true a, nnf true b)
47     else Or (nnf false a, nnf false b)
48   | Imp (a, b) -> if neg then And (nnf false a, nnf true b)
49     else Or (nnf true a, nnf false b)
50   | Iff (a, b) -> if neg then And (Or (nnf true a, nnf true b), Or (nnf false a, nnf
51     false b))
52     else Or (And (nnf false a, nnf false b), And (nnf true a, nnf true b))
53   | Forall (v, f) -> if neg then Exists (v, nnf true f)
54     else Forall (v, nnf false f)
55   | Exists (v, f) -> if neg then Forall (v, nnf true f)
56     else Exists (v, nnf false f)
57
58 let rec eliminate_triv = function
59   | False -> False
60   | True -> True
61   | Not f -> (match eliminate_triv f with
62     | True -> False
63     | False -> True
64     | f -> Not f)
65   | And (f, g) -> (match eliminate_triv f, eliminate_triv g with

```

```

65     | False, _ | _, False -> False
66     | True, f | f, True -> f
67     | f, g -> And (f, g))
68 | Or (f, g) -> (match eliminate_triv f, eliminate_triv g with
69   | True, _ | _, True -> True
70   | False, f | f, False -> f
71   | f, g -> Or (f, g))
72 | Imp (f, g) -> (match eliminate_triv f, eliminate_triv g with
73   | True, f -> f
74   | f, False -> Not f
75   | False, _ | _, True -> True
76   | f, g -> Imp (f, g))
77 | Iff (f, g) -> (match eliminate_triv f, eliminate_triv g with
78   | True, f | f, True -> f
79   | False, _ | _, False -> False
80   | f, g -> Iff (f, g))
81 | Forall (v, f) -> (match eliminate_triv f with
82   | False | True as f -> f
83   | f -> Forall (v, f))
84 | Exists (v, f) -> (match eliminate_triv f with
85   | False | True as f -> f
86   | f -> Exists (v, f))
87 | f -> f
88
89 let rec substitute_term s = function
90   | Var x -> s x
91   | Fn (n, l) -> Fn (n, List.map (substitute_term s) l)
92
93 let rec substitute_atom s = function
94   | P (n, l) -> P (n, List.map (substitute_term s) l)
95
96 let rec substitute_fol s = function
97   | False -> False
98   | True -> True
99   | Atom a -> Atom (substitute_atom s a)
100  | Not f -> Not (substitute_fol s f)
101  | And (f, g) -> And (substitute_fol s f, substitute_fol s g)
102  | Or (f, g) -> Or (substitute_fol s f, substitute_fol s g)
103  | Imp (f, g) -> Imp (substitute_fol s f, substitute_fol s g)
104  | Iff (f, g) -> Iff (substitute_fol s f, substitute_fol s g)
105  | Forall (v, f) -> Forall (v, substitute_fol s f)
106  | Exists (v, f) -> Exists (v, substitute_fol s f)
107
108 let substitute_literal s = apply_literal (substitute_atom s)
109
110 let substitute_clause s =
111   List.map (substitute_literal s)
112
113 let substitute_of_hashtbl s =
114   fun x -> if Hashtbl.mem s x then Hashtbl.find s x else Var x
115
116 let rec max_variable_fol = function
117   | Forall (v, f) | Exists (v, f) -> max v (max_variable_fol f)
118   | Not f -> max_variable_fol f
119   | And (f, g) | Or (f, g) | Imp (f, g) | Iff (f, g) ->
120     max (max_variable_fol f) (max_variable_fol g)
121   | _ -> -1
122
123 let rec max_variable_term = function
124   | Var x -> x
125   | Fn (_, l) -> List.fold_left (fun a b -> max a (max_variable_term b)) (-1) l
126
127 let rec max_variable_atom p = max_variable_term (term_of_atom p)
128
129 let rec max_variable_clause = function
130   | [] -> -1

```

```

131 | p::l -> max (max_variable_atom (atom_of_literal p)) (max_variable_clause l)
132
133 let rec non_variable_count_term = function
134 | Var _ -> 0
135 | Fn (_, l) -> List.fold_left (fun a b -> non_variable_count_term b + a) 1 l
136
137 let non_variable_count_atom p = non_variable_count_term (term_of_atom p)
138
139 let rec non_variable_count_clause = function
140 | [] -> 0
141 | p::l -> (non_variable_count_atom (atom_of_literal p)) + (non_variable_count_clause l)
142
143 let rec rename_term rewrite = function
144 | Var v -> Var (Hashtbl.find rewrite v)
145 | Fn (f, l) -> Fn (f, List.map (rename_term rewrite) l)
146
147 let rec rename_atom rewrite = function
148 | P (n, l) -> P (n, List.map (rename_term rewrite) l)
149
150 let rename f =
151   let c = ref 0 in
152   let rewrite = Hashtbl.create 0 in
153   let rec aux = function
154   | False -> False
155   | True -> True
156   | Atom a -> Atom (rename_atom rewrite a)
157   | Not f -> Not (aux f)
158   | And (a, b) -> let a = aux a and b = aux b in And (a, b)
159   | Or (a, b) -> let a = aux a and b = aux b in Or (a, b)
160   | Imp (a, b) -> let a = aux a and b = aux b in Imp (a, b)
161   | Iff (a, b) -> let a = aux a and b = aux b in Iff (a, b)
162   | Forall (v, f) -> let d = !c in incr c; Hashtbl.add rewrite v d; Forall (d, aux f)
163   | Exists (v, f) -> let d = !c in incr c; Hashtbl.add rewrite v d; Exists (d, aux f)
164   in
165   aux f;;
166
167 let skolem_name n = "S#" ^ (string_of_int n)
168
169 let prenex f =
170   let f = rename (nnf false f) in
171   let n = max_variable_fol f + 1 in
172   let r = Array.make n (-1) in
173   let rec aux = function
174   | Not f -> let p, f = aux f in
175     List.map (fun (b, v) -> not b, v) p, Not f
176   | And (f, g) -> let pf, f = aux f and pg, g = aux g in
177     prefix true pf pg, And (f, g)
178   | Or (f, g) -> let pf, f = aux f and pg, g = aux g in
179     prefix false pf pg, Or (f, g)
180   | Forall (v, f) -> let p, f = aux f in
181     if List.exists (fun (_, w) -> v = w) p then p, f
182     else (true, v)::p, f
183   | Exists (v, f) -> let p, f = aux f in
184     if List.exists (fun (_, w) -> v = w) p then p, f
185     else (false, v)::p, f
186   | f -> [], f
187   and prefix b pf pg = match pf, pg with
188   | [], p | p, [] -> p
189   | (bf, vf)::rf, (bg, vg)::rg when bf = b && bg = b ->
190     r.(vg) <- vf ; (b, vf)::prefix b rf rg
191   | (bf, vf)::rf, _ when bf <> b -> (bf, vf)::prefix b rf pg
192   | _, (bg, vg)::rg when bg <> b -> (bg, vg)::prefix b pf rg
193   and convert f = function
194   | [] -> f
195   | (true, v)::p -> Forall (r.(v), convert f p)
196   | (false, v)::p -> Exists (r.(v), convert f p) in

```

```

196 let p, f = aux f in
197 let c = ref 0 in
198 for i = 0 to n - 1 do
199   if r.(i) = -1 then (r.(i) <- !c; incr c)
200   else r.(i) <- r.(r.(i))
201 done;
202 let s = (fun i -> Var r.(i)) in
203 convert (substitute_fol s f) p
204
205 let skolemization f =
206   let f = prenex f in
207   let n = max_variable_fol f + 1 in
208   let c = ref 0 in
209   let skolem = Array.init n (fun i -> Var i) in
210   let skolem_variable vars v =
211     skolem.(v) <- (Fn(skolem_name !c, List.rev vars)); incr c in
212   let rec aux vars = function
213     | Forall (v, f) -> Forall (v, aux (Var v::vars) f)
214     | Exists (v, f) -> skolem_variable vars v; aux vars f
215     | f -> substitute_fol (fun i -> skolem.(i)) f in
216   aux [] f
217
218 let rem_quantifiers f =
219   let rec aux = function
220     | Forall (_, f) -> aux f
221     | f -> f in
222   aux (skolemization f)
223
224 let distribute f =
225   let rec aux = function
226     | And (a, b) -> And (aux a, aux b)
227     | Or (a, b) -> (match aux a, aux b with
228       | And (c, d), e -> And (aux (Or (c, e)), aux (Or (d, e)))
229       | c, And (d, e) -> And (aux (Or(c, d)), aux (Or(c, e)))
230       | _ -> Or (a, b))
231     | f -> f in
232   aux (rem_quantifiers f)
233
234 let convert_to_cnf f =
235   let rec clause c = function
236     | Atom a -> (L a)::c
237     | Not (Atom a) -> (NL a)::c
238     | Or (a, b) -> clause (clause c a) b
239     | f -> failwith "error" in
240   let rec aux l = function
241     | And (a, b) -> aux (aux l a) b
242     | f -> (List.rev (clause [] f))::l in
243   List.rev (aux [] (eliminate_triv (distribute f)))

```

disp.mli

```

1 open Base
2
3 val print_hashtbl : (int, term) Hashtbl.t -> unit
4 val print : fol -> unit
5 val tree_of_fol : bool -> fol -> string tree
6 val layout_compact : string tree -> string tree_l
7 val disp_layout : float -> int -> float -> fol -> unit
8 val print_cnf : cnf -> unit

```

disp.ml

```

1 open Base
2 open Fol_manip
3 open Graphics
4

```

```

5  let parenthesize print x =
6    print_string "(";
7    print x;
8    print_string ")"
9
10 let rec print_term = function
11   | Var v -> print_string "V"; print_int v
12   | Fn (f, l) -> print_string f;
13     if l <> [] then parenthesize print_term_list l
14 and print_term_list = function
15   | [] -> ()
16   | [p] -> print_term p
17   | h::t -> print_term h; print_string " "; print_term_list t
18
19 let print_atom p = print_term (term_of_atom p)
20
21 let rec print = function
22   | False -> print_string "FALSE"
23   | True -> print_string "TRUE"
24   | Atom a -> print_atom a
25   | Not (Atom a) -> print_string "NOT "; print_atom a
26   | Not f -> print_string "NOT"; parenthesize print f
27   | And (f, g) -> parenthesize print f;
28     print_string " AND ";
29     parenthesize print g
30   | Or (f, g) -> parenthesize print f;
31     print_string " OR ";
32     parenthesize print g
33   | Imp (f, g) -> parenthesize print f;
34     print_string " IMP ";
35     parenthesize print g
36   | Iff (f, g) -> parenthesize print f;
37     print_string " IFF ";
38     parenthesize print g
39   | Forall (v, f) -> print_string ("FORALL V" ^ (string_of_int v) ^ ".");
40     print f
41   | Exists (v, f) -> print_string ("EXISTS V" ^ (string_of_int v) ^ ".");
42     print f
43
44 let rec tree_of_term b = function
45   | Var v -> Tree ((if b then Printf.sprintf "$v_%d$" v
46                     else Printf.sprintf "V%d" v), [])
47   | Fn (f, l) -> Tree ((if b then Printf.sprintf "$%s$"
48                         (Str.global_replace (Str.regexp "#") "_"
49                         (Str.global_replace (Str.regexp "_") "\\_" f))
50                     else f), List.map (tree_of_term b) l)
51
52 let tree_of_atom b p = tree_of_term b (term_of_atom p)
53
54 let rec tree_of_fol b = function
55   | False -> Tree ((if b then "$\\bot$" else "false"), [])
56   | True -> Tree ((if b then "$\\top$" else "true"), [])
57   | Atom a -> tree_of_atom b a
58   | Not f -> Tree ((if b then "$\\neg$" else "not"), [tree_of_fol b f])
59   | And (f, g) -> Tree ((if b then "$\\wedge$" else "and"), [tree_of_fol b f; tree_of_fol b g])
60   | Or (f, g) -> Tree ((if b then "$\\vee$" else "or"), [tree_of_fol b f; tree_of_fol b g])
61   | Imp (f, g) -> Tree ((if b then "$\\rightarrow$" else "imp"),
62                         [tree_of_fol b f; tree_of_fol b g])
63   | Iff (f, g) -> Tree ((if b then "$\\leftrightarrow$" else "iff"),
64                         [tree_of_fol b f; tree_of_fol b g])
65   | Forall (v, f) -> Tree ((if b then Printf.sprintf "$\\forall v_%d$" v
66                             else Printf.sprintf "forall V%d" v), [tree_of_fol b f])
67   | Exists (v, f) -> Tree ((if b then Printf.sprintf "$\\exists v_%d$" v
68                             else Printf.sprintf "exists V%d" v), [tree_of_fol b f])

```



```

69
70 let rec tree_height (Tree (_, l)) = List.fold_left (fun a b -> max a (tree_height b))
    (-1) l + 1
71
72 let layout_compact t =
73   let rec min_l m = function
74     | [] -> m
75     | (l, _)::t -> min_l (min l m) t
76   and merge a b d = match a, b with
77     | [], [] -> []
78     | e, [] -> e
79     | [], (lb, rb)::tb -> (lb+.d, rb+.d)::(merge [] tb d)
80     | (la, _)::ta, (_, rb)::tb -> (la, rb+.d)::(merge ta tb d)
81   and dist a b = match a,b with
82     | e, [] | [], e -> 0.
83     | (_, lr)::ta, (rl, _)::tb -> max (l+.lr-.rl) (dist ta tb)
84   and move d = function Tree_l (v, t, x) -> Tree_l (v, t, x+.d)
85   and center = function
86     | [] -> 0.
87     | (l, r)::_ -> (l+.r)*.0.5
88   and prop t e = function
89     | [] -> let c = center e in
90       List.rev (List.map (move (-.c)) t),
91       List.map (function l, r -> l-.c, r-.c) e
92     | h::u -> let h, eh = aux h in
93       let dt = dist e eh in
94       prop ((move dt h)::t) (merge e eh dt) u
95   and aux = function
96     | Tree (v, []) -> Tree_l (v, [], 0.), [(0., 0.)]
97     | Tree (v, t) ->
98       let t, e = prop [] [] t in
99       Tree_l (v, t, 0.), (0., 0.)::e in
100   let layout, e = aux t in
101   move (-.(min_l 0. e)) layout
102
103 let disp_layout s h o f =
104   open_graph " 1600x720";
105   let t = tree_of_fol false f in
106   let n = tree_height t in
107   let rec aux x0 m = function
108     | Tree_l (v,t,x) ->
109       moveto (int_of_float (s*.(x0+.x))) (3*h*m);
110       draw_string v;
111       moveto (int_of_float (s*.(x0+.x))) (h*(3*m+1));
112       if m < n then
113         lineto (int_of_float (s*.x0)) (h*(3*(m+1)));
114         List.iter (aux (x0+.x) (m-1)) t in
115   aux o n (layout_compact t)
116
117 let save_file string =
118   let channel = open_out file in
119   output_string channel string;
120   close_out channel;;
121
122 let latex_of_fol f name =
123   let rec aux b (Tree_l (n, l, x)) =
124     let s = Printf.sprintf "\\begin{scope}[shift={{(f*\\treel},{-\\treehh)}}]\\n\\
125       \\node [above] at (0,0) {s};\\n" x n ^
126     (List.fold_right (fun a b -> aux true a ^ b) l "\\end{scope}\\n") in
127     if b then Printf.sprintf "\\draw (0,0) -- ({f*\\treel},{-\\treeh});\\n" x ^ s else s
128     in
129   let s = "\\begin{tikzpicture}\\n\\begin{scope}[scale={\\trees}]\\n" ^
130     (aux false (layout_compact (tree_of_fol true f)) ^
131     "\\end{scope}\\n\\end{tikzpicture}\\n" in
132   save (Printf.sprintf "tree_%s.tex" name) s;;
133

```

```

133 let print_hashtbl = Hashtbl.iter (fun x y -> print_string ("V" ^ (string_of_int x) ^ " ->
    ");
134                                     print_term y;
135                                     print_newline ())
136
137 let print_literal = function
138   | L p -> print_string "L "; print_atom p
139   | NL p -> print_string "NL "; print_atom p
140
141 let rec print_clause = function
142   | [] -> ()
143   | [p] -> print_literal p
144   | p::t -> parenthesize (fun _ -> print_literal p; print_string " OR "; print_clause t)
145   ()
146
147 let rec print_cnf = function
148   | [] -> ()
149   | [c] -> print_clause c
150   | c::t -> parenthesize (fun _ -> print_clause c; print_string " AND "; print_cnf t) ()

```

resolution.mli

```

1 open Base
2
3 val unify_naive : atom -> atom -> ((int, term) Hashtbl.t * bool)
4
5 val unify_terms : global -> term -> term -> bool
6
7 val pack_clause : global -> clause -> clause
8
9 val simplify_clause : clause -> clause
10
11 val subsumes : graph array -> clause -> clause -> bool
12
13 val tautology : clause -> bool
14
15 val resolve : global -> clause list -> literal -> clause -> (literal * bool) list ->
    clause list
16
17 val resolve_binary : global -> clause list -> clause -> clause -> clause -> clause ->
    clause list
18
19 val resolution_step : global -> clause list -> clause -> clause list -> (clause list *
    clause list)
20
21 val resolution : global -> clause list -> (clause list * clause list)
22
23 val resolution_process : fol -> (bool * (clause list * clause list))

```

resolution.ml

```

1 open Base
2 open Fol_manip
3
4 let rec occurs_check s x = function
5   | Var y -> y = x || (Hashtbl.mem s y && occurs_check s x (Hashtbl.find s y))
6   | Fn (_, l) -> List.exists (occurs_check s x) l
7
8 let unify_find_naive s = function
9   | Var u when Hashtbl.mem s u -> Hashtbl.find s u
10  | x -> x
11
12 let rec unify_var_naive s p q = match unify_find_naive s p, unify_find_naive s q with
13   | Var u, Var v when u = v -> true
14   | Var u, v | v, Var u ->
15     if Hashtbl.mem s u then unify_var_naive s (Hashtbl.find s u) v

```

```

16     else if not (occurs_check s u v) then
17         (Hashtbl.add s u v; true)
18     else false
19 | Fn (fp, lp), Fn (fq, lq) -> fp = fq && unify_list_naive s lp lq
20 and unify_list_naive s lp lq =
21     List.length lp = List.length lq && List.for_all2 (unify_var_naive s) lp lq
22
23 let reconstruct_naive s =
24     let t = Hashtbl.create (Hashtbl.length s) in
25     let rec aux = function
26         | Var x -> if not (Hashtbl.mem t x) then
27             Hashtbl.add t x (aux (match Hashtbl.find_opt s x with Some y -> y | None -> Var x
28                 ));
29         | Fn (f, l) -> Fn (f, List.map aux l) in
30     Hashtbl.iter (fun a _ -> ignore (aux (Var a))) s;
31     t
32
33 let unify_naive p q =
34     let s = Hashtbl.create 0 in
35     let s, b = (match p, q with
36         | P (np, lp), P (nq, lq) -> s, np = nq && unify_list_naive s lp lq) in
37     if b then
38         reconstruct_naive s, b
39     else
40         s, b
41
42 let rec dsu_find g x =
43     if g.(x).p <> x then g.(x).p <- dsu_find g g.(x).p;
44     g.(x).p
45
46 let dsu_union g a b =
47     let a, b = dsu_find g a, dsu_find g b in
48     if a <> b then
49         begin
50             if g.(a).r > g.(b).r then
51                 g.(b).p <- a
52             else if g.(a).r < g.(b).r then
53                 g.(a).p <- b
54             else
55                 begin
56                     g.(b).p <- a;
57                     g.(a).r <- g.(a).r + 1
58                 end
59         end
60
61 let rec vars_from_term g = function
62     | Var x -> if g.graph.(x).r = -1 then
63         begin
64             g.max <- max g.max (x + 1);
65             g.graph.(x).r <- 0;
66             g.vars <- x::g.vars
67         end
68     | Fn (_, l) -> List.iter (vars_from_term g) l
69
70 let rec graph_from_term g = function
71     | Var x -> x
72     | Fn (f, l) -> g.max <- g.max + 1; let d = g.max in g.graph.(d).r <- 0;
73         g.graph.(d).n <- NV (f, List.map (graph_from_term g) l); d
74
75 let unify_find g x =
76     let x = dsu_find g x in
77     match g.(x) with
78     | {n=V y} -> dsu_find g y
79     | _ -> x
80

```

```

81 let rec unify g p q =
82   let p, q = unify_find g p, unify_find g q in
83   p = q || match g.(p).n, g.(q).n with
84   | NV (u, r), NV (v, s) -> dsu_union g p q; u = v && unify_list g r s
85   | _, NV _ -> g.(p).n <- V q; true
86   | NV _, _ -> g.(q).n <- V p; true
87   | _, _ -> dsu_union g p q; true
88 and unify_list g r s =
89   List.length r = List.length s && List.for_all2 (unify g) r s
90
91 let acyclic g =
92   let next x =
93     let x = dsu_find g x in
94     match g.(x) with
95     | {n=V y} -> y
96     | _ -> x in
97   let rec dfs x =
98     if g.(x).r >= 0 then
99       begin
100         g.(x).r <- -1;
101         let b = (match g.(x).n with
102          | NV (_, l) -> List.for_all dfs l
103          | _ -> let y = next x in x = y || dfs y) in
104         g.(x).r <- -2; b
105       end
106     else g.(x).r = -2 in
107   List.for_all dfs
108
109 let rec reconstruct g =
110   let next x =
111     let x = dsu_find g.graph x in
112     match g.graph.(x) with
113     | { n = V y } -> y
114     | _ -> x in
115   let rec aux x = match g.graph.(x).n with
116   | T t -> t
117   | NV (f, l) -> let t = Fn (f, List.map aux l) in
118     g.graph.(x).n <- T t; t
119   | _ -> let y = next x in
120     if y = x then
121       Var x
122     else
123       (let t = aux y in
124        g.graph.(x).n <- T t; t) in
125   List.iter (fun x -> ignore (aux x)) g.vars
126
127 let rec unify_fast p q = match p, q with
128 | Var _, _ | _, Var _ -> true
129 | Fn (f, r), Fn (g, s) -> f = g && unify_list_fast r s
130 and unify_list_fast r s =
131   List.length r = List.length s && List.for_all2 unify_fast r s
132
133 let unify_terms g p q =
134   g.max <- -1; g.vars <- [];
135   vars_from_term g p; vars_from_term g q;
136   let r = graph_from_term g p and s = graph_from_term g q in
137   let b = unify g.graph r s && acyclic g.graph g.vars in
138   if b then
139     reconstruct g
140   else
141     (List.iter (fun x -> g.graph.(x) <- graph_default x) g.vars; g.vars <- []);
142   for i = r to g.max do
143     g.graph.(i) <- graph_default i
144   done;
145   g.max <- -1;
146   b

```

```

147
148 let unify_bool g p q =
149   g.max <- -1; g.vars <- [];
150   vars_from_term g p; vars_from_term g q;
151   let r = graph_from_term g p and s = graph_from_term g q in
152   let b = unify g.graph r s && acyclic g.graph g.vars in
153   List.iter (fun x -> g.graph.(x) <- graph_default x) g.vars;
154   g.vars <- [];
155   for i = r to g.max do
156     g.graph.(i) <- graph_default i
157   done;
158   g.max <- -1;
159   b
160
161 let unify_atom_bool g p q = match p, q with
162   | P (m, r), P (n, s) -> unify_bool g (Fn (m, r)) (Fn (n, s))
163
164 let unify_atoms g p q =
165   let p, q = term_of_atom p, term_of_atom q in
166   unify_terms g p q
167
168 let unify_routine g p q b =
169   let p, q = term_of_atom p, term_of_atom q in
170   unify_fast p q &&
171   begin
172     if b then
173       begin
174         let p = substitute_term (fun x -> Var (2 * x)) p in
175         let q = substitute_term (fun x -> Var (2 * x + 1)) q in
176         unify_terms g p q
177       end
178     else
179       unify_terms g p q
180   end
181
182 let rec pack_term g = function
183   | Var x -> (match g.graph.(x).n with
184     | T t -> t
185     | _ -> g.max <- g.max + 1; g.vars <- x::g.vars;
186       g.graph.(x).n <- T (Var g.max); Var g.max)
187   | Fn (f, l) -> Fn (f, List.map (pack_term g) l)
188
189 let pack_atom g = function
190   | P (n, l) -> P (n, List.map (pack_term g) l)
191
192 let pack_literal g = apply_literal (pack_atom g)
193
194 let pack_clause g c =
195   g.max <- -1; g.vars <- [];
196   let c = List.map (pack_literal g) c in
197   List.iter (fun x -> g.graph.(x) <- graph_default x) g.vars;
198   g.vars <- []; g.max <- -1;
199   c
200
201 let rec simplify_clause = function
202   | [] -> []
203   | h::t -> if List.mem h t then
204     simplify_clause t
205   else
206     h::simplify_clause t
207
208 let rec subsumes_unify g vars p q = match p, q with
209   | Var x, t -> (match g.(x).n with
210     | Nil -> g.(x).n <- T t; vars := x::!vars; true
211     | T u -> t = u
212     | _ -> false)

```

```

213 | Fn (f, r), Fn (h, s) when f = h && List.length r = List.length s ->
214   List.for_all2 (subsumes_unify g vars) r s
215 | _ -> false
216
217 let subsumes_atom g vars p q =
218   subsumes_unify g vars (term_of_atom p) (term_of_atom q)
219
220 let subsumes_literal g vars p q = match p, q with
221 | L p, L q | NL p, NL q -> subsumes_atom g vars p q
222 | _ -> false
223
224 let rec subsumes g cp cq = match cp with
225 | [] -> true
226 | p::tp -> List.exists (fun q ->
227   let vars = ref [] in
228   let b = subsumes_literal g vars p q && subsumes g tp cq in
229   List.iter (fun x -> g.(x).n <- Nil) !vars; b) cq
230
231 let rec tautology = function
232 | [] -> false
233 | h::t -> List.mem (negate_literal h) t || tautology t
234
235 let rec replace g v = function
236 | [] -> [v]
237 | h::t when subsumes g v h -> v::List.filter (fun p -> not (subsumes g v p)) t
238 | h::t -> h::replace g v t
239
240 let insert g u t w =
241   if tautology w ||
242     List.exists (fun p -> subsumes g p w) t ||
243     List.exists (fun p -> subsumes g p w) u then t
244   else replace g w t
245
246 let rec resolve g acc u h = function
247 | [] -> let h = simplify_clause h in
248   let h = pack_clause g h in
249   let m = max_variable_clause h in
250   let n = non_variable_count_clause h in
251   if 2*(m+n+1) > Array.length g.graph then
252     g.graph <- graph_make (4*(m+n+1));
253   h::acc
254 | (p, b)::t -> if b = (is_literal_positive p = is_literal_positive u) &&
255   unify_routine g (atom_of_literal p) (atom_of_literal u) false then
256   begin
257     let s = (fun x -> match g.graph.(x).n with T t -> t | _ -> Var x) in
258     let h2 = substitute_clause s h in
259     let t2 = List.map (apply (substitute_literal s)) t in
260     let u2 = substitute_literal s u in
261     List.iter (fun x -> g.graph.(x) <- graph_default x) g.vars;
262     g.vars <- [];
263     resolve g (resolve g acc u2 h2 t2) u (p::h) t
264   end
265 else
266   resolve g acc u (p::h) t
267
268 let rec resolve_binary g acc hp tp hq tq = match tp, tq with
269 | [], _ -> acc
270 | p::tp, [] -> resolve_binary g acc (p::hp) tp [] hq
271 | p::tp, q::tq -> if is_literal_positive p <> is_literal_positive q &&
272   unify_routine g (atom_of_literal p) (atom_of_literal q) true then
273   begin
274     let s = (fun x -> match g.graph.(2*x).n with T t -> t | _ -> Var (2*x)) in
275     let t = (fun x -> match g.graph.(2*x+1).n with T t -> t | _ -> Var (2*x+1)) in
276     let u = substitute_literal s p in
277     let h = substitute_clause s hp@substitute_clause t hq in
278     let t = List.map (fun p -> p, true) (substitute_clause s tp)@

```

```

279         List.map (fun p -> p, false) (substitute_clause t tq) in
280     List.iter (fun x -> g.graph.(x) <- graph_default x) g.vars;
281     g.vars <- [];
282     resolve_binary g (resolve g acc u h t) hp (p::tp) (q::hq) tq
283 end
284 else
285     resolve_binary g acc hp (p::tp) (q::hq) tq
286
287 let resolution_step g u v t =
288     let p = List.for_all is_literal_positive v in
289     let w = List.fold_left (fun a b ->
290         if p || List.for_all is_literal_positive b
291         then resolve_binary g a [] v [] b
292         else a) [] u in
293     v::u, List.fold_left (insert g.graph (v::u)) t w
294
295 let resolution g =
296     let rec aux u = function
297         | h::t when not (List.mem [] u) ->
298             print_int (List.length u); print_string ";\t";
299             print_int (List.length t + 1); print_string ";\t";
300             print_int (Array.length g.graph); print_newline ();
301             let a, b = resolution_step g u h t in
302             aux a b
303         | v -> u, v in
304     aux []
305
306 let preprocess g f =
307     List.fold_left (insert g []) [] (List.map simplify_clause f)
308
309 let resolution_process f =
310     let f = convert_to_cnf f in
311     let n = List.fold_left (fun a b ->
312         max ((max_variable_clause b) + (non_variable_count_clause b)) a) 0 f in
313     let g = global_make (4 * (n + 1)) in
314     let u = preprocess g.graph f in
315     let a, b = resolution g u in
316     List.mem [] a, (a, b)

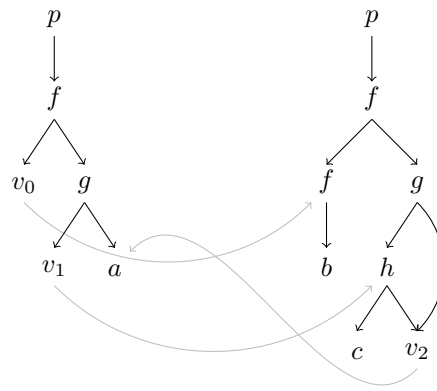
```

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 $[v_0 \rightarrow f(b), v_1 \rightarrow h(c, a), v_2 \rightarrow a]$.

```

1  # let t1 = Fn ("f", [Var 0; Fn ("g", [Var 1; Fn ("a", [])])]);;
2
3  val t1 : Base.term = Fn ("f", [Var 0; Fn ("g", [Var 1; Fn ("a", [])])])
4
5  # let t2 = Fn ("f", [Fn ("f", [Fn ("b", [])]); Fn ("g", [Fn ("h", [Fn ("c", []); Var 2]); Var 2])]);;
6
7  val t2 : Base.term =
8      Fn ("f",
9          [Fn ("f", [Fn ("b", [])]);
10             Fn ("g", [Fn ("h", [Fn ("c", []); Var 2]); Var 2])])
11
12  # let g = global_make 16;;
13
14  val g : Base.global =
15      {graph =
16          [|{n = Nil; p = 0; r = -1}; {n = Nil; p = 1; r = -1};
17             {n = Nil; p = 2; r = -1}; {n = Nil; p = 3; r = -1};
18             {n = Nil; p = 4; r = -1}; {n = Nil; p = 5; r = -1};
19             {n = Nil; p = 6; r = -1}; {n = Nil; p = 7; r = -1};
20             {n = Nil; p = 8; r = -1}; {n = Nil; p = 9; r = -1};
21             {n = Nil; p = 10; r = -1}; {n = Nil; p = 11; r = -1};
22             {n = Nil; p = 12; r = -1}; {n = Nil; p = 13; r = -1};
23             {n = Nil; p = 14; r = -1}; {n = Nil; p = 15; r = -1}|];
24      max = -1; vars = []}
25
26  # unify_terms g t1 t2;;
27
28  - : bool = true
29
30  # g;;
31
32  - : Base.global =
33      {graph =
34          [|{n = T (Fn ("f", [Fn ("b", [])])); p = 0; r = -2};
35             {n = T (Fn ("h", [Fn ("c", []); Fn ("a", [])])); p = 1; r = -2};
36             {n = T (Fn ("a", [])); p = 2; r = -2}; {n = Nil; p = 3; r = -1};
37             {n = Nil; p = 4; r = -1}; {n = Nil; p = 5; r = -1};
38             {n = Nil; p = 6; r = -1}; {n = Nil; p = 7; r = -1};
39             {n = Nil; p = 8; r = -1}; {n = Nil; p = 9; r = -1};
40             {n = Nil; p = 10; r = -1}; {n = Nil; p = 11; r = -1};
41             {n = Nil; p = 12; r = -1}; {n = Nil; p = 13; r = -1};
42             {n = Nil; p = 14; r = -1}; {n = Nil; p = 15; r = -1}|];
43      max = -1; vars = [2; 1; 0]}
44
45  # let s = (fun x -> match g.graph.(x).n with T t -> t | _ -> Var x);;
46
47  val s : int -> Base.term = <fun>
48
49  # substitute_term s t1;;
50
51  - : Base.term =
52      Fn ("f",
53          [Fn ("f", [Fn ("b", [])]);
54             Fn ("g", [Fn ("h", [Fn ("c", []); Fn ("a", [])]); Fn ("a", [])])])
55
56  # substitute_term s t2;;
57
58  - : Base.term =
59      Fn ("f",
60          [Fn ("f", [Fn ("b", [])]);
61             Fn ("g", [Fn ("h", [Fn ("c", []); Fn ("a", [])]); Fn ("a", [])])])

```



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("$v_2$", [])])])]]))]
)]
)]
)]

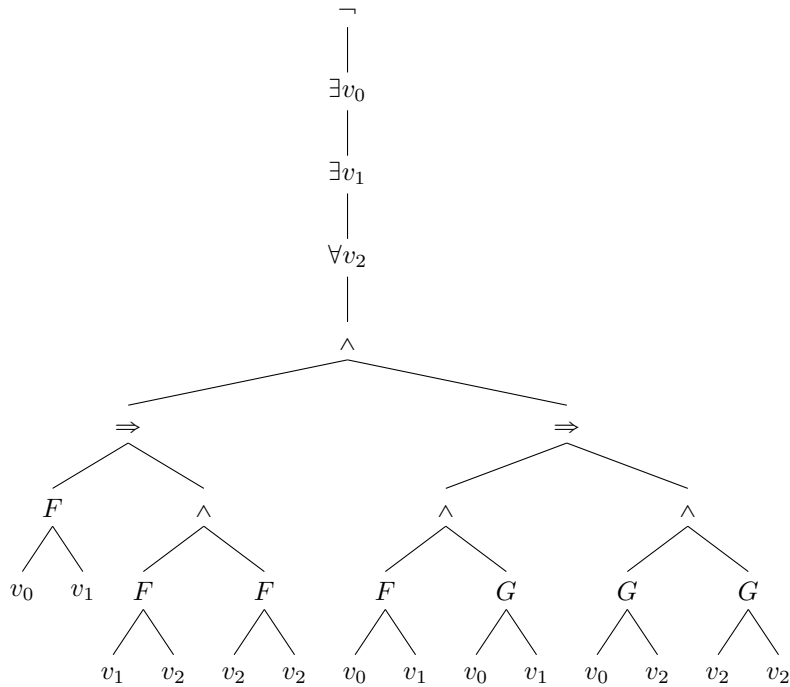
# let tl = layout_compact t;;

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$", [], 0.5)], 1.)],
1.25)],
-3.625);
Tree_l ("\rightarrow$",
```

```

61      [Tree_1 ("$\wedge$",
62      [Tree_1 ("F",
63      [Tree_1 ("v_0", [], -0.5); Tree_1 ("v_1", [], 0.5)],
64      -1.);
65      Tree_1 ("G",
66      [Tree_1 ("v_0", [], -0.5); Tree_1 ("v_1", [], 0.5)], 1.)],
67      -2.);
68      Tree_1 ("$\wedge$",
69      [Tree_1 ("G",
70      [Tree_1 ("v_0", [], -0.5); Tree_1 ("v_2", [], 0.5)],
71      -1.);
72      Tree_1 ("G",
73      [Tree_1 ("v_2", [], -0.5); Tree_1 ("v_2", [], 0.5)], 1.)],
74      2.)],
75      3.625)],
76      0.)],
77      0.)],
78      0.)],
79      0.)],
80      5.375)

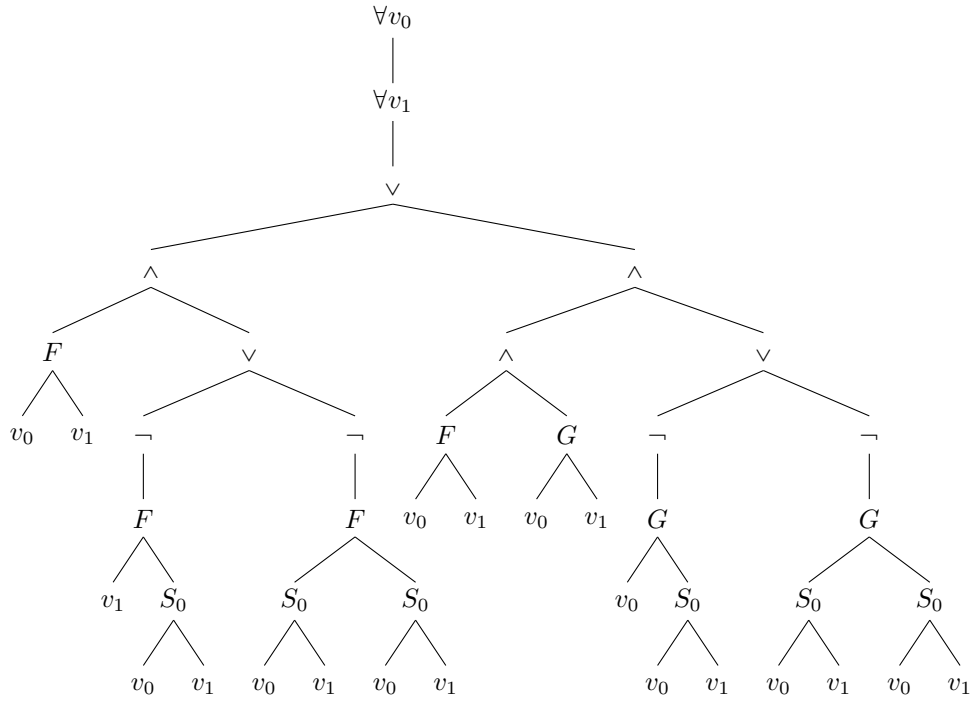
```



```

81 # nnf false f;;
82
83 - : Base.fol =
84 Forall (0,
85   Forall (1,
86     Exists (2,
87       Or
88       (And (Atom (P ("F", [Var 0; Var 1])),
89         Or (Not (Atom (P ("F", [Var 1; Var 2]))),
90         Not (Atom (P ("F", [Var 2; Var 2]))))),
91       And
92       (And (Atom (P ("F", [Var 0; Var 1])), Atom (P ("G", [Var 0; Var 1])),
93         Or (Not (Atom (P ("G", [Var 0; Var 2])),
94         Not (Atom (P ("G", [Var 2; Var 2]))))))))
95 \end{listlisting}
96
97 \input{tree_nnf.tex}
98
99 \begin{lstlisting}[name=interp]
100 # prenex f;;
101

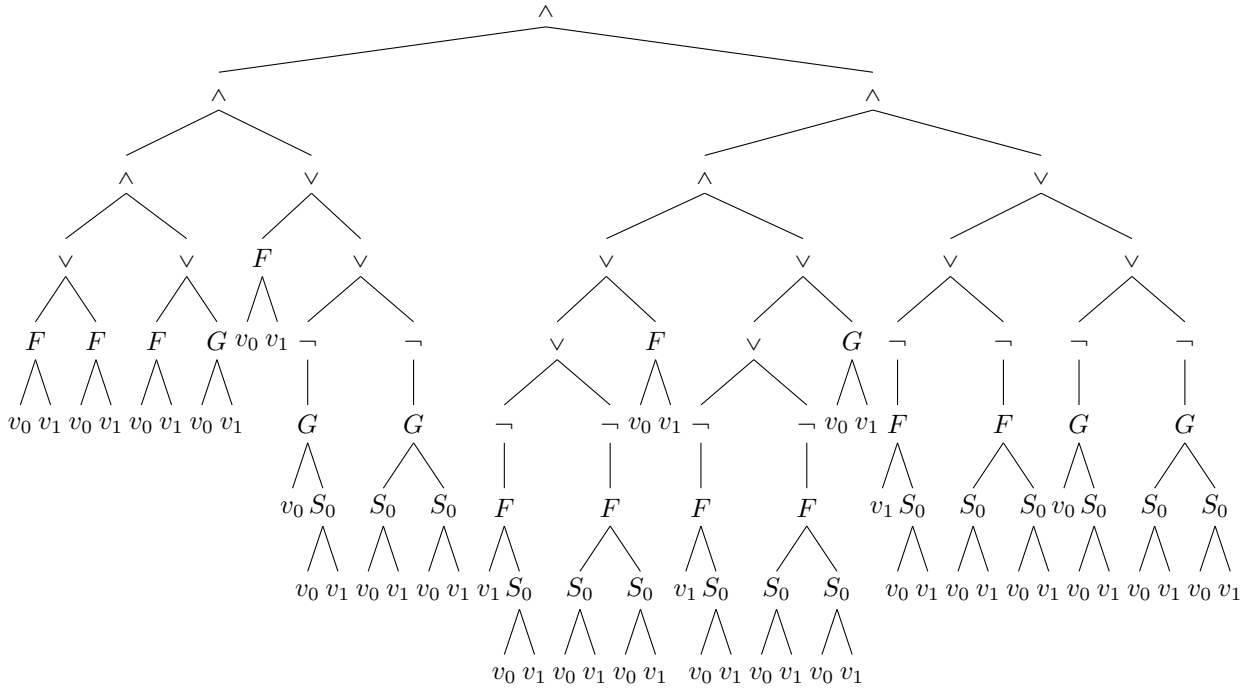
```

```

130 # distribute f;;
131
132 - : Base.fol =
133 And
134 (And
135   (And (Or (Atom (P ("F", [Var 0; Var 1])), Atom (P ("F", [Var 0; Var 1]))),
136     Or (Atom (P ("F", [Var 0; Var 1])), Atom (P ("G", [Var 0; Var 1])))),
137   Or (Atom (P ("F", [Var 0; Var 1])),
138     Or (Not (Atom (P ("G", [Var 0; Fn ("S#0", [Var 0; Var 1]))))),
139     Not
140       (Atom
141         (P ("G", [Fn ("S#0", [Var 0; Var 1]); Fn ("S#0", [Var 0; Var 1])])))),
142 And
143   (And
144     (Or
145       (Or (Not (Atom (P ("F", [Var 1; Fn ("S#0", [Var 0; Var 1])])),
146         Not
147           (Atom
148             (P ("F", [Fn ("S#0", [Var 0; Var 1]); Fn ("S#0", [Var 0; Var 1])])),
149             Atom (P ("F", [Var 0; Var 1]))),
150       Or
151         (Or (Not (Atom (P ("F", [Var 1; Fn ("S#0", [Var 0; Var 1])])),
152           Not
153             (Atom
154               (P ("F", [Fn ("S#0", [Var 0; Var 1]); Fn ("S#0", [Var 0; Var 1])])),
155               Atom (P ("G", [Var 0; Var 1]))),
156       Or
157         (Or (Not (Atom (P ("F", [Var 1; Fn ("S#0", [Var 0; Var 1])])),
158           Not
159             (Atom
160               (P ("F", [Fn ("S#0", [Var 0; Var 1]); Fn ("S#0", [Var 0; Var 1])])),
161               Or (Not (Atom (P ("G", [Var 0; Fn ("S#0", [Var 0; Var 1])])),
162                 Not
163                   (Atom
164                     (P ("G", [Fn ("S#0", [Var 0; Var 1]); Fn ("S#0", [Var 0; Var 1])])))))))

```



```

165 # let c = convert_to_cnf f;;
166
167 val c : Base.cnf =
168   [[L (P ("F", [Var 0; Var 1])); L (P ("F", [Var 0; Var 1]))];
169    [L (P ("F", [Var 0; Var 1])); L (P ("G", [Var 0; Var 1]))];
170    [L (P ("F", [Var 0; Var 1]));
171     NL (P ("G", [Var 0; Fn ("S#0", [Var 0; Var 1])));
172     NL (P ("G", [Fn ("S#0", [Var 0; Var 1]); Fn ("S#0", [Var 0; Var 1]))]);
173     [NL (P ("F", [Var 1; Fn ("S#0", [Var 0; Var 1])));
174      NL (P ("F", [Fn ("S#0", [Var 0; Var 1]); Fn ("S#0", [Var 0; Var 1])));
175      L (P ("F", [Var 0; Var 1]));
176      [NL (P ("F", [Var 1; Fn ("S#0", [Var 0; Var 1])));
177       NL (P ("F", [Fn ("S#0", [Var 0; Var 1]); Fn ("S#0", [Var 0; Var 1])));
178       L (P ("G", [Var 0; Var 1]));
179       [NL (P ("F", [Var 1; Fn ("S#0", [Var 0; Var 1])));
180        NL (P ("F", [Fn ("S#0", [Var 0; Var 1]); Fn ("S#0", [Var 0; Var 1])));
181        NL (P ("G", [Var 0; Fn ("S#0", [Var 0; Var 1])));
182        NL (P ("G", [Fn ("S#0", [Var 0; Var 1]); Fn ("S#0", [Var 0; Var 1]))])]]]]];
183
184 # let g = global_make 10;;
185
186 val g : Base.global =
187   {graph =
188     [|{n = Nil; p = 0; r = -1}; {n = Nil; p = 1; r = -1};
189      {n = Nil; p = 2; r = -1}; {n = Nil; p = 3; r = -1};
190      {n = Nil; p = 4; r = -1}; {n = Nil; p = 5; r = -1};
191      {n = Nil; p = 6; r = -1}; {n = Nil; p = 7; r = -1};
192      {n = Nil; p = 8; r = -1}; {n = Nil; p = 9; r = -1}];
193     max = -1; vars = []}
194
195 # let u::v::w::x::y::z = c;;
196
197 # resolve_binary g [] [] u [] x;;
198
199 - : Base.literal list list =
200   [[L (P ("F", [Var 0; Fn ("S#0", [Var 1; Var 0])));
201     NL (P ("F", [Fn ("S#0", [Var 1; Var 0]); Fn ("S#0", [Var 1; Var 0])));
202     L (P ("F", [Var 1; Var 0]))];
203    [NL (P ("F", [Var 0; Fn ("S#0", [Var 1; Var 0])));
204     L (P ("F", [Fn ("S#0", [Var 1; Var 0]); Fn ("S#0", [Var 1; Var 0])));
205     L (P ("F", [Var 1; Var 0]))];

```

```

206 [L (P ("F", [Var 0; Var 1]));
207 L (P ("F", [Fn ("S#0", [Var 0; Var 1]); Fn ("S#0", [Var 0; Var 1]))));
208 NL (P ("F", [Var 1; Fn ("S#0", [Var 0; Var 1]))));
209 [L (P ("F", [Var 0; Var 1]));
210 NL (P ("F", [Var 1; Fn ("S#0", [Var 0; Var 1]))));
211 [L (P ("F", [Var 0; Var 1]));
212 NL (P ("F", [Fn ("S#0", [Var 0; Var 1]); Fn ("S#0", [Var 0; Var 1]))));
213 L (P ("F", [Var 1; Fn ("S#0", [Var 0; Var 1]))));
214 [L (P ("F", [Var 0; Var 1]));
215 NL (P ("F", [Fn ("S#0", [Var 0; Var 1]); Fn ("S#0", [Var 0; Var 1]))))]
216
217 # resolution_step g [u] v (w::x::y::z);;
218
219 - : Base.clause list * Base.clause list =
220 ([L (P ("F", [Var 0; Var 1])); L (P ("G", [Var 0; Var 1]))];
221 [L (P ("F", [Var 0; Var 1])); L (P ("F", [Var 0; Var 1]))],
222 [L (P ("F", [Var 0; Var 1]));
223 NL (P ("G", [Var 0; Fn ("S#0", [Var 0; Var 1])));
224 NL (P ("G", [Fn ("S#0", [Var 0; Var 1]); Fn ("S#0", [Var 0; Var 1]))));
225 [NL (P ("F", [Var 1; Fn ("S#0", [Var 0; Var 1])));
226 NL (P ("F", [Fn ("S#0", [Var 0; Var 1]); Fn ("S#0", [Var 0; Var 1])));
227 L (P ("F", [Var 0; Var 1]))];
228 [NL (P ("F", [Var 1; Fn ("S#0", [Var 0; Var 1])));
229 NL (P ("F", [Fn ("S#0", [Var 0; Var 1]); Fn ("S#0", [Var 0; Var 1])));
230 L (P ("G", [Var 0; Var 1]))];
231 [NL (P ("F", [Var 1; Fn ("S#0", [Var 0; Var 1])));
232 NL (P ("F", [Fn ("S#0", [Var 0; Var 1]); Fn ("S#0", [Var 0; Var 1])));
233 NL (P ("G", [Var 0; Fn ("S#0", [Var 0; Var 1])));
234 NL (P ("G", [Fn ("S#0", [Var 0; Var 1]); Fn ("S#0", [Var 0; Var 1]))))]
235
236 # resolution_process f;;
237
238 0;      3;      48
239 1;      2;      48
240 2;      3;      48
241 3;      4;      48
242 4;      3;      48
243 5;      4;      48
244 6;      3;      48
245 7;      2;      48
246 8;      1;      48
247 - : bool * (Base.clause list * Base.clause list) =
248 (true,
249 ([[]; [NL (P ("G", [Var 0; Fn ("S#0", [Var 0; Var 1]))]);
250 [NL (P ("G", [Var 0; Fn ("S#0", [Var 0; Var 1])));
251 NL (P ("G", [Fn ("S#0", [Var 0; Var 1]); Fn ("S#0", [Var 0; Var 1]))]);
252 [NL (P ("G", [Fn ("S#0", [Var 0; Var 1]); Fn ("S#0", [Var 0; Var 1])));
253 NL (P ("G", [Var 0; Fn ("S#0", [Var 0; Var 1])));
254 NL (P ("F", [Var 1; Fn ("S#0", [Var 0; Var 1]))]);
255 [L (P ("G", [Var 0; Var 1]))];
256 [L (P ("G", [Var 0; Var 1]))];
257 NL (P ("F", [Var 1; Fn ("S#0", [Var 0; Var 1])));
258 [NL (P ("F", [Var 1; Fn ("S#0", [Var 0; Var 1])));
259 NL (P ("F", [Fn ("S#0", [Var 0; Var 1]); Fn ("S#0", [Var 0; Var 1])));
260 NL (P ("G", [Var 0; Fn ("S#0", [Var 0; Var 1])));
261 NL (P ("G", [Fn ("S#0", [Var 0; Var 1]); Fn ("S#0", [Var 0; Var 1])));
262 [NL (P ("F", [Var 1; Fn ("S#0", [Var 0; Var 1])));
263 NL (P ("F", [Fn ("S#0", [Var 0; Var 1]); Fn ("S#0", [Var 0; Var 1])));
264 L (P ("G", [Var 0; Var 1]))];
265 [L (P ("F", [Var 0; Var 1]))],
266 []))

```
