TD : points fixes des fonctions monotones, sémantique dénotationnelle

Le plus petit point (pre)fixe

Exercice 1. Soit $(G, \rightarrow)$ un graphe dirigé, possiblement infini. On définit un ensemble de règles R sur G de cette façon : X/y ssi $X = S(y) = \{\, x \,|\, y \rightarrow x \,\}$.

– Montrer que l'opérateur f_R associé à R est :

$$f_R(Z) = \Box Z \;=\; \{\, y \,|\, \forall x\, y \rightarrow x \text{ implique } x \in Z \}\,.$$

– Donner un exemple d'un graphe $(G, \rightarrow)$ où la propriété

$$f_R(\bigcup_{n\geq 0} f_R^n(\emptyset)) = \bigcup_{n\geq 0} f_R^n(\emptyset)$$

n'est pas vraie.

– Dans un tel graphe, existe-t'il un point fixe de l'opérateur f_R ?

Exercice 2. Soient $f : P(A) \longrightarrow P(B)$ et $g : P(B) \longrightarrow P(A)$ deux fonctions monotones. On assume qu'un plus petit point fixe de $g \circ f$, noté $fix(g \circ f)$, et un plus petit point fixe de $f \circ f$, noté $fix(f \circ g)$, existent. Montrer que :

$$f(\, fix(g \circ f)\,) = fix(f \circ g)\,.$$

On pourra se servir des axiomes suivants :

$$\begin{aligned} & h(fix(h)) \leq fix(h) \\ h(y) \leq y \;\; & \rightarrow fix(h) \leq y\,. \end{aligned}$$

Sémantique dénotationnelle du langage IML

Exercice 3. Soit $\Sigma : Loc \longrightarrow N$ l'ensemble de états. Rappelons la définition de l'opérateur

$$\Gamma : P(\Sigma \times \Sigma) \longrightarrow P(\Sigma \times \Sigma)\,.$$

On a

$$\begin{aligned} \Gamma(X) \;=& \{\, (\sigma, \sigma') \,|\, \beta(\sigma) = true \text{ et } (\sigma, \sigma') \in X \circ \gamma \,\} \\ & \cup \{\, (\sigma, \sigma) \,|\, \beta(\sigma) = false \,\}\,, \end{aligned}$$

où $\beta = \mathcal{B}\|b\|$ pour quelque $b \in Bexpr$ et $\gamma \subseteq \Sigma \times \Sigma$ est arbitraire.

1. Montrer que $\Gamma = f_R$ pour un ensemble R de règles sur $\Sigma \times \Sigma$.
2. En déduire que Γ est monotone et que $fix(\Gamma)$, le plus petit point préfixe de Γ existe.
3. Prouver, par induction sur les règles, que si γ est une fonction partielle alors $fix(\Gamma)$ est une fonction partielle.

Le démonstrateur automatique

Exercice 4. Voici une première implémentation du démonstrateur automatique.

1. Le code est subdivisé en modules. Analyser chaque module et en comprendre la fonction dans le contexte.
2. Dessiner le graphe de dépendances des modules.
3. Comment améliorer la modularité du code ?
4. Quel fonctionnalités (= modules) sont manquantes ?
5. Pour quelle raison, dans le module `Terme` on défini une fonction `print` ? Peut-on laisser OCAML imprimer lui-même un terme suite à son évaluation ?
6. Dans quels modules on utilise un style fonctionnel ? Dans quels modules on utilise un style impératif ?

outils.mli

```
 1 : val mapfilter : ('a -> 'b) -> 'a list -> 'b list
```

outils.ml

```
 1 : let rec mapfilter fonction liste =
 2 :   let rec mapfilter accu = function
 3 :       [] -> List.rev accu
 4 :     | t :: q ->
 5 :         let tl=
 6 :           try
 7 :             [ fonction t ]
 8 :           with
 9 :               Failure(_)  -> []
10 :         in
11 :           mapfilter (tl @ accu) q
12 :   in
13 :     mapfilter [] liste;;
14 :
```

terme.mli

```
 1 : type ('a,'b) t
 2 : val faire_op : 'a -> ('a, 'b) t list -> ('a, 'b) t
 3 : val faire_var : 'a -> ('b, 'a) t
 4 : val print : (string,string) t -> unit
 5 : val vars : ('a, 'b) t -> 'b list
 6 : type ('a, 'b) subst
 7 : val identite : ('a, 'b) subst
 8 : val faire_subst : 'a -> ('b, 'a) t -> ('b, 'a) subst
 9 : val apply : ('a ,'b) subst -> ('a, 'b) t -> ('a, 'b) t
10 : val compose : ('a ,'b) subst -> ('a, 'b) subst -> ('a, 'b) subst
11 : val filtrer : (('a, 'b) t * ('a, 'b) t) list -> ('a, 'b) subst
12 : val unify : (('a, 'b) t * ('a, 'b) t) list -> ('a, 'b) subst
13 : val merge_l :
14 :   ('a,'b) subst list -> ('a,'b) subst list -> ('a,'b) subst list
```

terme.ml (1-69)

```
 1 : (* Termes *)
 2 : type ('a,'b) t = Var of 'b
 3 :                | Operation of 'a * ('a,'b) t list ;;
 4 :
 5 : let faire_op a fils = Operation(a,fils);;
 6 : let faire_var b  = Var(b);;
 7 :
 8 : let rec to_string = function
 9 :     Var(x) -> x
10 :   | Operation(f,fils) ->
11 :       let rec fils_to_string = function
12 :           [] -> ""
13 :         | [t] -> to_string t
14 :         | t::q -> (to_string t)^","^(fils_to_string q)
15 :       in
16 :         f^"("^(fils_to_string fils)^")";;
17 :
18 : let print t = print_string(to_string t);;
19 :
20 : let rec vars = function
21 :     Var x -> [x]
22 :   | Operation(f,fils) -> List.concat (List.map vars fils);;
23 :
24 :
25 : (* Substitutions *)
26 : type ('a,'b) subst = ('b * ('a,'b) t) list;;
27 :
28 : let identite = [];;
29 : let faire_subst x  t  = [(x,t)] ;;
30 : let rec apply subst = function
31 :     Var x ->
32 :       (
33 :         try
34 :           List.assoc x subst
35 :         with
36 :             Not_found -> Var x
37 :       )
38 :   | Operation(f,fils) ->
39 :       Operation(f,List.map (apply subst) fils);;
40 : let compose sub1 sub2 =
41 :   let
42 :     predicat = fun (x,t) -> not (List.mem x (List.map fst sub2) )
43 :   in
44 :     (List.map (fun (x,t) -> (x, apply sub2 t)) sub1)
45 :     @ List.filter predicat sub1;;
46 :
47 : let add couple subst =
48 :   match couple with
49 :       (x,t) ->
50 :         try
51 :           if (t = (List.assoc x subst) ) then subst else failwith "add"
52 :         with
53 :             Not_found -> couple::subst
54 :           | Failure "add" -> failwith "add";;
55 :
56 : let rec merge (sub1: ('a,'b) subst) sub2 =
57 :   try
58 :     match sub1 with
59 :         [] -> sub2
60 :       | t::q -> merge q (add t sub2)
61 :   with
62 :       Failure "add" -> failwith "merge";;
63 :
64 : let merge_el sub liste =
65 :   Outils.mapfilter (fun t -> merge sub t ) liste;;
66 :
67 : let merge_l liste1 liste2 =
68 :   List.concat (List.map (fun sub -> merge_el sub liste2) liste1);;
69 :
```

terme.ml (70-117)

```
70 : (* Filtrage *)
71 : let rec filtrer = function
72 :   [] -> identite
73 :   | tete::queue ->
74 :       let
75 :         phi = match tete  with
76 :             (Var x, t )  ->
77 :               if List.mem x (vars t) then failwith "filtrer"
78 :               else faire_subst x t
79 :           | (Operation(f,ff),Operation(g,fg)) ->
80 :               if( f = g && (List.length ff) = (List.length fg))
81 :               then filtrer (List.combine ff fg)
82 :               else failwith "filtrer"
83 :           | (_,Var x)  -> failwith "filtrer"
84 :       and
85 :         psi = filtrer queue
86 :       in
87 :         try
88 :           merge psi phi
89 :         with
90 :             Failure "merge" -> failwith "filtrer"
91 : ;;
92 :
93 : (* Unification *)
94 :
95 : let rec unify = function
96 :     [] -> identite
97 :   | tete::queue ->
98 :       let
99 :         phi =
100 :         match tete  with
101 :             (Var x, t ) | (t, Var x ) ->
102 :               if List.mem x (vars t) then failwith "unify"
103 :               else faire_subst x t
104 :           | (Operation(f,ff),Operation(g,fg)) ->
105 :               if( f = g && (List.length ff) = (List.length fg))
106 :               then unify (List.combine ff fg)
107 :               else failwith "unify"
108 :       in
109 :       let
110 :         psi =
111 :         unify
112 :           (List.map
113 :              (fun (t1,t2) -> (apply phi t1, apply phi t2)) queue )
114 :       in
115 :         compose psi phi
116 : ;;
117 :
```

clause.mli

```
 1 : type ('a,'b,'c) atomic_formula
 2 : type ('a,'b,'c) clause
 3 : val faire_atf : 'a -> ('b, 'c) Terme.t list
 4 :   -> ('a, 'b, 'c) atomic_formula
 5 : val faire_clause :
 6 :   ('a, 'b, 'c) atomic_formula list ->
 7 :   ('a, 'b, 'c) atomic_formula list -> ('a, 'b, 'c) clause
 8 : val vide : ('a,'b,'c) clause
 9 : val length : ('a,'b,'c) clause -> int*int
10 : val apply : ('b,'c) Terme.subst ->
11 :   ('a,'b,'c) atomic_formula ->
12 :   ('a,'b,'c) atomic_formula
13 : val apply_to_clause : ('b,'c) Terme.subst ->
14 :   ('a,'b,'c) clause ->
15 :   ('a,'b,'c) clause
16 : val unify : ('a,'b,'c) atomic_formula *('a,'b,'c) atomic_formula ->
17 :   ('b,'c) Terme.subst
18 : val factoriser : ('a,'b,'c) clause*int*int
19 :   -> ('a,'b,'c) clause
20 : val resolution : ('a,'b,'c) clause*int
21 :   -> ('a,'b,'c) clause*int -> ('a,'b,'c) clause
22 : val taut : ('a, 'b, 'c) clause -> bool
23 : val sous : ('a, 'b, 'c) clause ->
24 :   ('a, 'b, 'c) clause -> bool
```

clause.ml (1-35)

```
 1 : type ('a,'b,'c) atomic_formula = Formula of 'a * ('b,'c) Terme.t list;;
 2 : type ('a,'b,'c) clause = { g : ('a,'b,'c) atomic_formula list ;
 3 :                            d : ('a,'b,'c) atomic_formula list
 4 :                          };;
 5 :
 6 : let faire_atf p termes = Formula(p,termes);;
 7 : let faire_clause g d = { g=g; d=d};;
 8 :
 9 : let vide = { g = []; d =[]};;
10 : let length cl = (List.length cl.g,List.length cl.d);;
11 : let apply subst = function
12 :     Formula(p,terms) ->
13 :       Formula(p, List.map (Terme.apply subst) terms);;
14 : let apply_to_clause subst = function
15 :     { g=g; d=d } ->
16 :       { g=List.map (apply subst) g; d=List.map (apply subst) d };;
17 :
18 : let unify = function
19 :      (Formula(p1,terms1), Formula(p2,terms2)) ->
20 :        if
21 :          ( p1 = p2 && (List.length terms1) = (List.length terms2) )
22 :        then
23 :          Terme.unify (List.combine terms1 terms2 )
24 :        else
25 :          failwith "unify";;
26 :
27 : let  remove n l =
28 :   let rec remove m acc = function
29 :       [] -> failwith "remove"
30 :     | t::q -> if m = 0 then (List.rev acc) @ q
31 :       else remove (m -1) (t::acc) q
32 :   in
33 :     remove n [] l
34 : ;;
35 :
```

clause.ml (36-112)

```
36 : let rec factoriser ({g=g;d=d},i,j) =
37 :   let
38 :     f_at1 = List.nth d i
39 :   and
40 :     f_at2 = List.nth d j
41 :   in
42 :     try
43 :       let
44 :         sigma = unify (f_at1,f_at2)
45 :       in
46 :         apply_to_clause sigma {g=g;d=(remove i d)}
47 :     with
48 :         Failure "unify" -> failwith "factoriser"
49 : ;;
50 :
51 :
52 : let rec resolution ({g=g1;d=d1},i) ({g=g2;d=d2},j) =
53 :   let
54 :     f_at1 = List.nth d1 i
55 :   and
56 :     f_at2 = List.nth g2 j
57 :   in
58 :     try
59 :       let
60 :         sigma = unify (f_at1,f_at2)
61 :       in
62 :         apply_to_clause sigma {g=g1@(remove j g2);d=(remove i d1)@d2}
63 :     with
64 :         Failure "unify" -> failwith "resolution"
65 : ;;
66 :
67 : (* Predicats pour la simplification *)
68 :
69 : (* Est une clause une tautologie ? *)
70 : let taut {g=g;d=d} =
71 :   let
72 :     predicat af_d = List.exists (fun af_g -> af_g = af_d) g
73 :   in
74 :     List.exists predicat d;;
75 :
76 : let filtrer = function
77 :     (Formula(p1,terms1), Formula(p2,terms2)) ->
78 :       if
79 :         (p1 = p2 && (List.length terms1) = (List.length terms2) )
80 :       then
81 :         Terme.filtrer (List.combine terms1 terms2 )
82 :       else
83 :         failwith "filtrer";;
84 :
85 : (* Est une clause une sous clause d'une autre ? *)
86 : (* Plus fin de la logique propositionelle, *)
87 : (* Il faut se servir du filtrage. *)
88 :
89 : let rec filtrer_l at liste =
90 :   match liste with
91 :       [] -> []
92 :     | t::q ->
93 :         let
94 :           tetel =
95 :           try
96 :             [ filtrer (at,t) ]
97 :           with
98 :               Failure "filtrer" -> []
99 :         in
100 :           tetel @ (filtrer_l at q);;
101 :
102 : let rec sous_l liste1 liste2 = match liste1 with
103 :     [] -> [Terme.identite]
104 :   | t::q ->
105 :       let
106 :         candidatq =
107 :         sous_l liste1 liste2
108 :       and
109 :         candidatt =  filtrer_l t liste2
110 :       in
111 :         Terme.merge_l candidatt candidatq ;;
112 :
113 :
```

produire.mli

```
1 : val faire_reagir :
2 :   ('a, 'b, 'c) Clause.clause ->
3 :   ('a, 'b, 'c) Clause.clause list -> ('a, 'b, 'c) Clause.clause list
```

produire.ml

```
1 : let all_factorisations cl  =
2 :   let (lg,ld) = Clause.length cl
3 :   and
4 :     result = ref []
5 :   in
6 :     for i = 1 to ld - 1 do
7 :       for j = i+1 to ld do
8 :         try
9 :           result := (Clause.factoriser (cl,i,j))::!result
10 :        with
11 :            Failure "factoriser" -> ()
12 :      done
13 :    done;
14 :    !result ;;
15 :
16 : let all_resolutions cl1 cl2  =
17 :   let
18 :     (lg1,ld1) = Clause.length cl1
19 :   and
20 :     (lg2,ld2) = Clause.length cl2
21 :   and
22 :     result = ref []
23 :   in
24 :     for i = 1 to ld1  do
25 :       for j = 1 to lg2 do
26 :         try
27 :           result := (Clause.resolution (cl1,i) (cl2,j))::!result
28 :         with
29 :             Failure "resolution" -> ()
30 :       done
31 :     done;
32 :     for i = 1 to ld2  do
33 :       for j = 1 to lg1 do
34 :         try
35 :           result := (Clause.resolution (cl2,i) (cl1,j))::!result
36 :         with
37 :             Failure "resolution" -> ()
38 :       done
39 :     done;
40 :     !result ;;
41 :
42 : let faire_reagir clause liste =
43 :   (all_factorisations clause)@
44 :   (List.concat (List.map (all_resolutions clause) liste));;
```

simplifier.mli

```
1 : val tautologies :
2 :   ('a, 'b, 'c) Clause.clause list -> ('a, 'b, 'c) Clause.clause list
3 : val susclauses :
4 :   ('a, 'b, 'c) Clause.clause list ->
5 :   ('a, 'b, 'c) Clause.clause list -> ('a, 'b, 'c) Clause.clause list
```

simplifier.ml

```
1 : let tautologies clauses = List.filter (fun c  -> not (Clause.taut c)) clauses;;
2 :
3 : let susclauses listepivot clauses =
4 :   List.filter
5 :     (fun cl -> not
6 :        (List.exists (fun pivot -> Clause.sous pivot cl) listepivot))
7 :     clauses;;
8 :
```

saturer.mli

saturer.ml

```
 1 : let saturer theorie =
 2 :   let wo = ref []
 3 :   and us = ref theorie
 4 :   in
 5 :   let
 6 :     sature = ref false
 7 :   and
 8 :     contradictoire = ref false
 9 :   in
10 :     begin
11 :       while not !sature do
12 :         if !us = [] then
13 :           (sature := true; contradictoire := false)
14 :         else
15 :           if List.mem Clause.vide !us then
16 :             (sature := true; contradictoire := false)
17 :           else
18 :             let c = List.hd !us in
19 :               wo:= c::!wo;
20 :               us := List.tl !us;
21 :               let nouveaux = ref (Produire.faire_reagir c !wo)  in
22 :                 nouveaux := Simplifier.tautologies !nouveaux;
23 :                 wo := Simplifier.susclauses !nouveaux !wo;
24 :                 nouveaux := Simplifier.susclauses !wo !nouveaux;
25 :                 us := Simplifier.susclauses !nouveaux !us;
26 :                 nouveaux := Simplifier.susclauses !us !nouveaux;
27 :                 us := (!us)@(!nouveaux)
28 :       done;
29 :       match !contradictoire with
30 :           true -> None
31 :         | false -> Some(!wo)
32 :     end;;
```