

# Termes formels, substitutions, filtrage

Solange Coupet Grimal

Cours Logique, Dédution, Programmation 2002–2004\*

## Représentation des termes

Le type `term`, analogue à celui des arbres quelconques, prend en compte 2 types de noeuds :

- les variables (constructeur `Var`) étiquetées par un élément de type `'b`.
- les opérateurs (constructeur `Operation`) étiquetés par un élément de type `'a` avec une liste de fils de type `term`. Cette liste est éventuellement vide si ces éléments sont des constantes.

```
# type ('a,'b) term =  
  Var of 'b  
  | Operation of 'a*( 'a,'b) term list;;  
type ('a, 'b) term = Var of 'b | Operation of 'a * ( 'a, 'b) term li
```

Exemple : considérons le terme  $f(g(x), a)$ , où  $x$  est une variable et  $a$  est une constante. On le représenter par l'expression suivante :

```
# Operation ( "f" , [  
  Operation("g", [Var "x"] );  
  Operation("a", [])  
  ]);;  
- : (string, string) term =  
Operation ("f", [Operation ("g", [Var "x"]); Operation ("a", [])])
```

Une constante peut être considérée comme un symbole de fonction 0-aire. Pour souligner ce point de vue, on écrira une constante avec des parenthèses qui le suivent. Par exemple, le terme précédent devient  $f(g(x), a())$ .

## Itérateur sur les termes

Comme pour les arbres, on définit un itérateur – appelons-le provisoirement `R` – sur les termes. Deux cas sont à envisager :

---

\*Texte révisé par Luigi Santocanale le 28 octobre 2005, et adapté au langage Objective Caml.

- Si  $t = \text{Var } x$ , alors  $R t = v x$ .
- Si  $t = \text{Operation}(r, [t_1; t_2; \dots; t_n])$  alors, comme pour les arbres,  $R t$  est de la forme :

```

      f r g (R t1) (...(g (R tn-1) (g (R tn) a)))
# let rec term_iter funvar funop funcoller premier = fonction
  (Var x) -> funvar x
  | (Operation (oper, fils)) ->
    let
      recur term = term_iter funvar funop funcoller premier term
    in
    let
      coller = fun t x -> funcoller (recur t) x
    in
      funop oper (List.fold_right coller fils premier);;
val term_iter :
  ('a -> 'b) ->
  ('c -> 'd -> 'b) ->
  ('b -> 'd -> 'd) -> 'd -> ('c, 'a) term -> 'b = <fun>

```

### Calcul de l'ensemble des variables d'un terme.

- Si  $t = \text{Var } x$ , c'est  $[x]$ .
- Si  $t = \text{Operation}(r, [t_1; t_2; \dots; t_n])$  alors c'est l'union des variables de  $t_1, \dots, t_n$ .  
On a donc  $v x = [x]$ ,  $f r l = l$ ,  $g = \text{union}$ ,  $a = []$ .

La fonction union est définie (par exemple) par :

```

# let ajouter ens el =
  if List.mem el ens then
    ens
  else el::ens;;
  val ajouter : 'a list -> 'a -> 'a list = <fun>
# let union ens1 ens2 = List.fold_left ajouter ens1 ens2;;
val union : 'a list -> 'a list -> 'a list = <fun>

```

On a donc :

```

# let vars = term_iter
  (fun x -> [x])
  (fun x y -> y) union [];;
val vars : ('_a, '_b) term -> '_b list = <fun>

```

### Tests.

Considérons :

(1)  $x + y*z$  :

```

# let t = Operation("plus",
  [Var "x";

```

```

        Operation("fois", [Var "y";Var "z"])]);;
val t : (string, string) term =
  Operation ("plus",
            [Var "x"; Operation ("fois", [Var "y"; Var "z"])]))
# vars t;;
- : string list = ["y"; "z"; "x"]

(2) t' = x + y*x
# let t' =
  Operation("plus",
            [Var "x";
             Operation("fois",
                       [Var "y";Var "x"])]);;
val t' : (string, string) term =
  Operation ("plus", [Var "x";
                     Operation ("fois", [Var "y"; Var "x"])]))
# vars t';;
- : string list = ["y"; "x"]

(3) t'' = x+y*c ou c est une constante.
# let t'' =
  Operation("plus", [
                    Var "x";
                    Operation("fois", [Var "y";Operation( "c", [])])
                    ]);;
  val t'' : (string, string) term =
  Operation ("plus",
            [Var "x"; Operation ("fois", [Var "y"; Operation ("c", [])])])
# vars t'';;
- : string list = ["y"; "x"]

```

### Occurrence d'une variable dans un terme.

```

# let occurs v t = List.mem v (vars t);;
val occurs : string -> (string, string) term -> bool = <fun>
# occurs "x" t;;
- : bool = true
# occurs "w" t;;
- : bool = false

```

## Substitutions

On se propose de substituer des variables par des termes. Une substitution est une fonction partielle qui associe à une variable un terme, par exemple :

$$\begin{array}{l} x \rightarrow 2 + 3 \quad ( = +(2(),3()) ) \\ y \rightarrow 4 \quad \quad ( = 4() ) \end{array}$$

En Caml une substitution  $s$  peut être représentée par une liste de couples :

```
# let subst =
  [ ("x", Operation ("plus", [Operation("2", []); Operation("3", [])]));
    ("y", Operation("4", [])) ];;
  val subst : (string * (string, 'a) term) list =
  [("x", Operation ("plus", [Operation ("2", []); Operation ("3", [])]))
   ("y", Operation ("4", []))]
```

Soit  $t$  un terme et  $R\ t$  le résultat des substitutions de  $s$  dans  $t$ .

- Si  $t = \text{Var } x$ , alors  $R\ t = \text{try (List.assoc } x\ s) \text{ with } \_ \rightarrow (\text{Var } x)$ , c'est-à-dire que si  $x$  est une variable, soit elle est dans  $s$  et on la remplace par l'élément correspondant, soit elle n'y est pas et on la laisse intacte.
- Si  $t = \text{Operation}(r, [t_1; t_2; \dots; t_n])$  alors :

```
R t = Operation(r, [(R t1); (R t2); ... ; (R tn)])=
      f r g (R t1) (...(g (R tn-1) (g (R tn) a)))
```

où

```
g = fun x l -> x::l
a = []
et f r l = Operation(r,l)
```

Par suite :

```
# let apply_subst s =
  term_iter
  (fun x -> try (List.assoc x s) with _ -> (Var x))
  (fun r l -> Operation(r,l))
  (fun x l -> x::l) [];;
val apply_subst :
  ('a * ('b, 'a) term) list ->
  ('b, 'a) term -> ('b, 'a) term =
<fun>
```

### Remarque

On aura pu définir un itérateur sur les termes `iter_term` en remplaçant le `List.fold_right` avec un `List.fold_left` dans la définition de `term_iter`.

On s'aperçoit alors que la substitution peut être définie de façon semblable, à l'aide de l'itérateur `iter_term`. En effet, si un terme est une variable `Var x`, alors il faut chercher la valeur de la variable à l'aide de la fonction

```
(fun x -> try (List.assoc x s) with _ -> (Var x))
```

Rien ne change dans ce cas. Si le terme `t` est une `Operation`, on a

```
apply_subst s t = Operation(r, [(apply_subst s t1); (apply_subst s t2);
  ... ; (apply_subst s tn)]) =
  f r (g ... (g (g premier (apply_subst s t1)) (apply_subst s t2))
    ... (apply_subst s tn))
```

où

```
g = fun x l -> l@[x]
a = []
et f r l = Operation(r,l) .
```

On remarque que la fonction `g`, qui ajoute l'élément `x` en queue de la liste, est bien coûteuse. Comment peut-on améliorer le code, en sauvegardant le principe qu'on veut utiliser `List.fold_left` et `iter_term` qui sont récursif terminaux ?

**Test.**

```
# apply_subst subst t;;
- : (string, string) term =
Operation ("plus",
  [Operation ("plus", [Operation ("2", []); Operation ("3", [])]);
  Operation ("fois", [Operation ("4", []); Var "z"])])
```

**Composée de substitutions.**

On se donne deux substitutions `s1` et `s2`, sous forme de listes et on calcule la liste `s` correspondant à  $(s1) \circ (s2)$ . On calcule pour cela la liste de tous les couples  $(x, s1(u))$  ou  $(x,u)$  est dans `s2`, puis on concatène la liste des variables transformées par `s1` mais pas par `s2`.

Pour cela on définit la fonction `filter`<sup>1</sup> qui ne garde d'une liste que les éléments satisfaisant un certain prédicat `p`.

```
# let rec filter p = function
  [] -> []
  | (a::l) -> if (p a) then (a::filter p l) else (filter p l);;
  val filter : ('a -> bool) -> 'a list -> 'a list = <fun>
```

puis on définit la composée des substitutions de la façon suivante :

```
# let compsubst s1 s2 =
  (List.map (fun (x,u) -> (x,apply_subst s1 u)) s2)@
  let var_s2=(List.map fst s2) in
  (List.filter (fun (x,t)-> not (List.mem x var_s2)) s1);;
```

---

<sup>1</sup>En Ocaml, cette fonction est définie dans le module `List`, et donc on peut s'en servir en la nommant `List.filter`.

```

    val compsubst :
('a * ('b, 'a) term) list ->
('a * ('b, 'a) term) list -> ('a * ('b, 'a) term) list = <fun>

```

On remarque que l'on peut garder entièrement `s1` puisque dans l'utilisation des listes de substitutions, on ne s'intéresse qu'à la première occurrence d'une variable. On obtient alors une définition plus simple :

```

# let compsubst s1 s2 =
  (List.map (fun (x,u) -> (x,apply_subst s1 u)) s2)@s1;;
val compsubst :
('a * ('b, 'a) term) list ->
('a * ('b, 'a) term) list -> ('a * ('b, 'a) term) list = <fun>

```

## Filtrage

Le filtrage d'un terme `t` par un terme `f` (le filtre) consiste à trouver une substitution `s` telle que `s(f) = t`.

- Si `f` est une variable `x`, alors il y a une substitution et une seule qui convient :  
`s = [(x, t)]`.
- Si `f` est de la forme `r(f1, ... ,fn)` alors le filtrage est possible si :
  1. `t` est de la forme `r(t1, ... ,tn)`,
  2. pour tout `i` le filtrage de `ti` par `fi` est possible,
  3. les substitutions engendrées par ces `n` filtrages ne sont pas incompatibles, c'est-à-dire si lorsque l'on ajoute un couple `(x,t)` à la substitution courante elle ne contient pas déjà un couple de la forme `(x,t')` avec `t` différent de `t'`.

Pour traiter la troisième condition, on écrit une fonction `add_subst` qui rajoute à une substitution `s` un couple `(x,t)` lorsque il est possible et sinon renvoie une exception. On rappelle que l'exception renvoyée par `(List.assoc x l)` quand `x` n'a pas été trouvé est `Not_found`. On définit ainsi :

```

# exception Match_exc;;
exception Match_exc
# let add_subst s (x,t) =
  try
    let t' = (List.assoc x s) in
      if t = t' then s
      else raise Match_exc
  with
    Not_found -> (x,t)::s;;
val add_subst : ('a * 'b) list ->
'a * 'b -> ('a * 'b) list = <fun>

```

Le filtrage (en anglais « matching ») se fait alors comme suit. On rappelle que la fonction `combine` est définie dans le module `List`, qu'elle transforme un couple de listes en une liste

de couples et renvoie l'exception `Invalid_argument "List.combine"` si les 2 listes ont différentes longueurs. Ainsi, si `sons_f=[f1; ...; fn]` et `sons_t = [t1; ...;tn]`, on obtient

```
List.combine sons_f sons_t = [(f1,t1); ... ;(fn,tn)]
```

La fonction `match_rec s (f,t)` ci-dessous rajoute à une substitution `s` successivement le résultat des filtrages des `ti` par les `fi`. Ainsi,

```
match_rec s
  (Operation(op,[f1; ...; fn]), Operation(op,[t1; ... ;tn])) =
  (match_rec( ... (match_rec(match_rec s (f1,t1)) (f2,t2))
    ... (fn,tn))
```

La fonction ci-dessous renvoie une exception si les 2 opérateurs `op_f` et `op_t` n'ont pas le même nombre d'arguments.

```
# let matching =
  let rec
    match_rec s = function
      (Var(x), t) -> add_subst s (x,t)
    | (Operation(op_f,sons_f),Operation(op_t,sons_t)) ->
      if op_f = op_t then
        (List.fold_left match_rec s
          (List.combine sons_f sons_t))
      else raise Match_exc
    | _ -> raise Match_exc
  in
    match_rec [];;
  val matching :
    ('_a, '_b) term * ('_a, '_c) term ->
    ('_b * ('_a, '_c) term) list = <fun>
```

## Tests.

Testons sur  $f = x + y * z$  et  $t = (a+b) + (x*y) * (a+b)$ , on doit obtenir

$$s = [(x, a+b); (y, x*y); (z, a+b)] .$$

```
# let f = Operation("plus",
  [ Var("x");
    Operation("fois", [Var("y");Var("z")])]);;
  val f : (string, string) term =
  Operation ("plus", [Var "x"; Operation ("fois", [Var "y"; Var "z"])]])
# let t = Operation("plus",
  [ Operation("plus",
    [Var("a");Var("b")])];;
```

```

        Operation("fois", [
            Operation("fois",
                [Var("x");Var("y")]);
            Operation("plus",
                [Var("a");Var("b")])])]);];
    val t : (string, string) term =
Operation ("plus",
  [Operation ("plus", [Var "a"; Var "b"]);
   Operation ("fois",
     [Operation ("fois", [Var "x"; Var "y"]);
      Operation ("plus", [Var "a"; Var "b"])]))]
# matching (f,t);;
- : (string * (string, string) term) list =
[("z", Operation ("plus", [Var "a"; Var "b"]));
 ("y", Operation ("fois", [Var "x"; Var "y"]));
 ("x", Operation ("plus", [Var "a"; Var "b"]))]

```

Testons sur  $f' = x+y*x$  et  $t$  inchangé.

```

# let f' = Operation("plus", [
    Var("x");
    Operation("fois", [Var("y");Var("x")])]);;
    val f' : (string, string) term =
    Operation ("plus", [Var "x"; Operation ("fois", [Var "y"; Var "x"])]
# matching (f',t);;
- : (string * (string, string) term) list =
[("y", Operation ("fois", [Var "x"; Var "y"]));
 ("x", Operation ("plus", [Var "a"; Var "b"]))]

```

Puis sur  $f'$  et  $t' = (a+b)+(x+y)*z$  :

```

# matching (f',t');;
- : (string * (string, string) term) list =
[("y", Var "y"); ("x", Var "x")]

```

Puis sur  $f$  et  $t'' = x+(y*z)+u$  avec un opérateur "+" ternaire.

```

# let t'' = Operation("plus", [
    Var("x");
    Operation("fois", [Var("y");Var("z")]);
    Var("u")]);;
    val t'' : (string, string) term =
    Operation ("plus",
      [Var "x"; Operation ("fois", [Var "y"; Var "z"]); Var "u"])
# matching (f,t'');;
Exception: Invalid_argument "List.combine".

```

Il vaut mieux ré-écrire `matching` pour avoir une exception appropriée dans ce cas-la.



```

# let matching' =
let rec
  match_rec s = function
    (Var(x), t) -> add_subst s (x,t)
  | ( Operation(op_f,sons_f),Operation(op_t,sons_t)) ->
    if op_f = op_t then
      try
        (List.fold_left match_rec s
          (List.combine sons_f sons_t))
          with _ -> raise Match_exc
      else raise Match_exc
    | _ -> raise Match_exc
in
  match_rec [];;
val matching' :
('a, 'b) term * ('a, 'c) term ->
('b * ('a, 'c) term) list = <fun>

```

### Traitement plus précis des erreurs.

On peut affiner la définition des exceptions pour prendre en compte les différents types d'erreurs.

```

# matching' (f,t'');;
Exception: Match_exc.
# exception Match_exc of string;;
exception Match_exc of string
# let add_subst s (x,t) =
  try
    let t' = (List.assoc x s) in
      if t = t' then s
      else raise (Match_exc ("impossibility with "^x))
  with
    Not_found -> (x,t)::s;;
val add_subst : (string * 'a) list ->
string * 'a -> (string * 'a) list =
<fun>
# let matching =
let rec
  match_rec s = function
    (Var(x), t) -> add_subst s (x,t)
  | (Operation(op_f,sons_f),Operation(op_t,sons_t)) ->
    if op_f = op_t then
      try
        (List.fold_left match_rec s
          (List.combine sons_f sons_t))
          with
            with

```

```

        (Invalid_argument "List.combine") ->
            raise (Match_exc("the operator "^op_f^"
                             " has different numbers of arguments
                             else raise (Match_exc "operator incompatibility")
    | _ -> raise (Match_exc "operator incompatibility")
in
    match_rec [];;
val matching :
(string, string) term * (string, '_a) term ->
(string * (string, '_a) term) list = <fun>

```

On peut noter que le type de matching est plus restrictif qu'auparavant.

```

# matching (f,t);;
- : (string * (string, string) term) list =
[("z", Operation ("plus", [Var "a"; Var "b"]));
 ("y", Operation ("fois", [Var "x"; Var "y"]));
 ("x", Operation ("plus", [Var "a"; Var "b"]))]
# matching (f,t');;
- : (string * (string, string) term) list =
[("z", Var "x"); ("y", Var "y"); ("x", Var "x")]
# matching (f',t);;
- : (string * (string, string) term) list =
[("y", Operation ("fois", [Var "x"; Var "y"]));
 ("x", Operation ("plus", [Var "a"; Var "b"]))]
# matching (f',t');;
- : (string * (string, string) term) list =
[("y", Var "y"); ("x", Var "x")]
# matching (f',t''');;
Exception:
Match_exc "the operator plus has different numbers of arguments".

```