ENSIIE

Rapport de projet d'IPF

Projet individuel d'algorithme-programmation IPF

CHEN Zeyu

Enseignant : C. Dubois

03/04/2018

PROJET IPF : TERMES ET RÉÉECRITURE

CHEN ZEYU

Table des matières

Date : 03/04/2018.

1. Introduction

Termes et réécriture : On dois réaliser un système de réécriture permettant dappliquer de façon auto-matique des règles de réécriture. C'est à dire , donne un terme complex et on le simplifie en fonction de certains règle donné.

2. Développement

2.1. **Pre-définition.** On a besion de définir un terme d'abord , un terme est soit :

- *une variable*
- *une constante*
- *un terme de la forme g(t) d'arite 1*
- *un terme de la forme f(x,y) d'arite 2*

Donc, le code pour ce type sera :

```
type terme = Var of string | Const of string |
| Unop of string * terme | Binop of string * terme * terme;;
```

On a aussi deux règle de réécriture :

```
plus(x, zero) -> x
plus(zero, x) -> x
plus(succ(x), y) -> plus(x, succ(y)).
```

et :

```
top(push(x, y))  -> x
pop(push(x, y))  -> y
alternate(vide, z)  -> z
alternate(push(x, y), z)  -> push(x, alternate(y, z))
```

2.2. **Question 1 : Test un terme bien formé.** On doit écrire les fonction qui teste si un terme est bien formé par rapport à une signature.D'abord, il faut creer un type terme et une sigature.

L'idée c'est que on passe deux arguments signature et un string à une fonction appelée **est_dedans** , il retourne true si ce string existe dans signature. Et on a une l'autre fonction appelée **test_terme_s** qui prends deux arguments signature et un terme t , on fait match with pour ce terme, quand t est un variable , il retourne true ,sinon, on itère .

Algorithm 1. Test un terme bien fomé

```
1 : function TEST_TERME(sign, t)
2 :     if t est Var str then
3 :         return true
4 :     end if
5 :     if t est Const str then
6 :         return est_deans (sign,str)
7 :     end if
8 :     if t est Unop(str,t) then
9 :         if est_deans (sign,str) and test_terme (sign,t) sont varis then
10 :            return true
11 :        else
12 :            return false
13 :        end if
14 :    end if
15 :    if t est Binop(str1,t1,t2 then
16 :        if est_deans (sign,str) and test_terme (sign,t1)
17 :            and test_terme (sign,t2) sont varis then
18 :            return true
19 :        else
20 :            return false
21 :        end if
22 :    end if
23 : end function
```

2.3. **Question 2 : Test motif.** Un motif ne contient qu'une seule occurrence de chaque variable.Donc, pour tester un terme est bien un motif, il faut tester si le nombre d'occurrence de chaque variable. Mon idée c'est que je retire tous les variables d'un terme dans une liste, et je crée une fonction pour juger si le nombre d'occurrence de chaque variable est bien 1 en utilisant fold_right, si c'est ce cas la , on retourne true , sinon, false.

Algorithm 2. Test un terme est un motif

```
1 : function IF_MOTIF(t)
2 :     retirer tous les var de t dans une liste l
3 :     if le nombre d'occurrence de chaque var de l = 1 then
4 :         return true
5 :     else
6 :         return false
7 :     end if
8 : end function
```

2.4. **Question 3 : Test un motif filtre un terme.** On prends un motif et un terme, et on veut savoir si ce motif peut bien filtrer t , si oui, on donne une liste de couples de la forme (string*terme), sinon, on donne failwith.

On peut remarquer t est un terme sans variable

ALGORITHM 3. Test un motif filtre un terme

```
1 : function SUBSITUTION(m, t)
2 :     if m est Var v then
3 :         return [(v,t)]
4 :     end if
5 :     if m est Const s1 and t est Const s2 then
6 :         if s1 = s2 then
7 :             return ∅
8 :         else
9 :             failwith
10 :        end if
11 :    end if
12 :    if m est Unop(s1,t1) and t est Unop(s2,t2) then
13 :        if s1 = s2 then
14 :            return subsitution t1 t2
15 :        else
16 :            failwith
17 :        end if
18 :    end if
19 :    if m est Binop(s1,g1,g2) and t est Binop(s2,g2,d2) then
20 :        if s1 = s2 then
21 :            return subsitution g1 g2 concate subsitution d1 d2
22 :        else
23 :            failwith
24 :        end if
25 :    end if
26 : end function
```

2.5. **Question 4 : Test un systèem de réécriture bien formé.** On rappelle qu'une règle de réécriture sécrit tg -> td ou tg est un terme linéaire avec variables et td un terme avec ou sans variable On remarque que si un système de réécriture est bien formé, il suffit qu'on vérifie tg et td respectent la sigature (cond1) et ils sont motif (cond2). Par ailleurs, on dois vérifier que les variabes dans td sont déjà dans tg (cond3). Voici l'algorithm pour résoudre :

ALGORITHM 4. Test un systèm de réécriture bien formé

```
1 : function TEST_SYS(tg, td)
2 :     if tg et td respect les signature and ils sont des motif then
3 :         On vérifie que les cond3
4 :         if les variabes dans td sont déjà dans tg then
5 :             return true
6 :         else
7 :             return false
8 :         end if
9 :     end if
10 : end function
```

2.6. **Question 5 : Applique des règle sur un terme.** Un système de règle est représenté par une liste, on peut donc parcourir cette liste et filter t par tg de chaque règle dans chaque itération. Le problème c'est que le résultat de filtre est aussi représenté par une liste. Donc, l'idée c'est que j'ai utilisé deux fois List.fold_right pour parcourir respectivement les deux liste, et une fois on a vérifié que une substitution d'un terme est exisist on appelle une fonction appelé remplace qui sert à remplacer s par td dans chaque itération, après on compare notre résultat avec notre terme au début, si ils sont identique, stop ,sinon ,on recommence à le simplifier. Voici l'algorithm pour résoudre :

ALGORITHM 5. Applique des règle sur un terme

```
1 : function APP_REGLE(regle, t)
2 :     Soit r est un règle de tous les règles, et ti est le résultat obtenu
3 :     après filtré t par r
4 :     if ti égale t then
5 :         return ti
6 :     else
7 :         return app_regle regle ti
8 :     end if
9 : end function
```

2.7. **Question 6 : Affichage des étapes de simplification.** Pour imprimer la suite des termes obtenus au cours de la réécriture ainsi que le numéro de la règle qui a permis de lobtenir. L'idée c'est qu'on essaie d'obtenir une liste de type couple (int*terme) contenant un numéro qui indique que le nièm règle peut filtrer t et une terme qui été simplifié par cette règle.On peut donc imprimer chaque étape de simplification en utilisant List.map.

Algorithm 6. Affichage des étapes de simplification

```
1 : function COUPLE_PRINT(regle, t, n)
2 :     while il exsiste une règle r_n peut bien filter t  do
3 :         n est le position de cette règle ti est une terme simplifié par r
4 :         if n != 0 then
5 :             return (n, ti) :: couple_print regle ti 1
6 :         else
7 :             return ∅
8 :         end if
9 :     end while
10 : end function
11 :
12 : function PRINT(regle, t)
13 :     Soit l = couple_print regle t 1
14 :     for y ∈ l do
15 :         imprime y
16 :     end for
17 : end function
```

2.8. **Question 7 : Test deux terme.** C'est une question relativement simple, on juste appelle la fonction précédent et on compare notre résultat, true si ils sont pareil ,false sinon

Algorithm 7. Test deux terme

```
1 : function COMPARE_TERME(t1, t2)
2 :     Soit (t1_s,t2_s) et les termes simplifié pas regle
3 :     if t1_s égale t2_s then
4 :         return true
5 :     else
6 :         return false
7 :     end if
8 : end function
```

2.9. **Question 8 : Amélioration.** On vérifie que si le terme simplifié par les règles est identique que le terme complet, si oui, on vérifie si les règle peuvent s'appliquer sur un des sou-termes du terme complet.

Algorithm 8. Amélioration

```
1 : function NEW_APP_REGLE(regle, t)
2 :     Soit t_a et les termes simplifié pas règle
3 :     if t_a égale t then
4 :         on applique les règle sur les sou-termes
5 :     else
6 :         return t_a
7 :     end if
8 : end function
```

3. Conclusion

L'objectif de ce projet était de simplifier une terme à l'aide d'un système de réécriture. Pour ces huit question , j'ai bien réalisé et testé correctement. Cependant, le système de réécriture qu'on utilise ne sert qu'à simplifier le terme qu'on a défini, Var , Const , Unop et Binop, mais pas pour les fonction avec plus d'arguments.

4. Annexes

4.1. **Les interfaces et les tests de Q1.** Quatre fonctions

```
1
2  (** fst
3  @param un couple (a,b)
4  @return  le premier element d'un couple
5  *)
6  let fst (a,b) = a;;
7
8  (** est_dedans
9  @param un list l un string str
10 @return true si str est dans l false sinon
11 *)
12 let rec est_dedans l str = match l with
13   [] -> false
14 | h::r -> if str = (fst h) then true
15           else est_dedans r str;;
16
17 (** test_terme_s
18 @param un signature sign , un terme t
19 @return : true si t satisfait le signature false sinon
20 *)
21 let rec test_terme_s sign t = match t with
```

```
    |Var str -> true
    |Const str -> est_dedans sign str
    |Unop(str,t)-> if est_dedans sign str && test_terme_s sign t
                   then true else false
    |Binop(str,t1,t2)->
                   if (est_dedans sign str) &&
                   (test_terme_s sign t1) && (test_terme_s sign t2)
                   then true else false;;

(** test_terme
@param un terme t
@return true si t est dans terme bien forme
*)
let test_terme t = test_terme_s signature t;;
```

```
(**----------test----------
let terme1 =
Binop("f", Binop("f", Var "x",Binop("f",Unop("g",Var "t"),Const "a")), Var

test_terme terme1;; (*--true--**)
let terme2 = Binop("t",Const "a",Var "e");;

test_terme terme2;; (*--false--**)
*)
```

4.2. **Les interfaces et les tests de Q2.** Quatre fonctions

```
(** list_var
@param     un terme t
@return  : une liste qui contient tous les variable de terme t
*)
let rec list_var t  = match t with
| Var str ->  [str]
| Const str -> []
| Unop(str,t)-> list_var t
| Binop(str,t1,t2)-> (list_var t1 )@(list_var t2 );;

(** nb_occ
@param     une liste t et un 'a s
@return  : le nombre d'occurance de s dans l
*)
let rec nb_occ l s = List.fold_right
```

```
(function x->function y-> if x =s then y+1 else y) l 0;;

(** if_doule
@param     une liste l
@return : true si le nombre de tous les element de l est 1 false sinon
*)
let if_doule l  = let l1 = List.map
(function x -> if nb_occ t x =1 then true else false) l in
if nb_occ l1 false >=1 then false else true ;;

(** if_motif
@param   : une terme t
@return : true si t est motif false sinon
*)
let rec if_motif t = let l = list_var t in if_doule l;;
```

```
(**----------test----------*
let terme1 = Binop ("f",Var "x", Unop ("g", Var "x"));;
if_motif terme1;; (* bool = false*)
let terme2 = Binop ("f",Var "u", Unop ("g", Var "y"));;
if_motif terme2;; (* bool = true*)
let terme3 = Unop ("g",Var "x");;
if_motif terme3;; (* bool = true*)
*)
```

4.3. **Les interfaces et les tests de Q3.** une fonctions

```
(*compare directement m et t*)

let rec substitution m t  = match (m, t) with
  | (Var v1, str) -> [(v1, str)]
  | (Const s1, Const s2) ->if s1 = s2 then []
                           else failwith "Ne filtre pas"
  | (Unop(s1, t1), Unop(s2, t2)) -> if s1 = s2 then (substitution t1 t2)
                           else failwith "Ne filtre pas "
  | (Binop(s1, g1, d1), Binop(s2, g2, d2)) -> if s1 = s2
                           then (substitution g1 g2)@(substitution d1 d2)
                           else failwith "Ne filtre pas"
  | _ -> failwith "Ne filtre pas";;
```

```
(**----------test----------*
substitution motif1 terme1;;
```

```
(*motif1=Binop ("f", Var "x", Const "a")*)
(*terme1=Binop ("f", Binop ("f", Const "a", Const "b"), Const "b"))*)
(*(string * terme) list = [("x", Binop ("f", Const "a", Const "b"))]*)

substitution motif2 terme1;;
(*motif2=Binop ("f", Binop ("f", Var "x", Var "y"), Const "a")*)
(*Exception: Failure "m ne filtre pas t"*)

substitution motif2 terme3;;

(*terme3=Binop ("f",Binop ("f", Binop ("f", Const "a",
 Const "b"),Binop ("f", Const "a", Const "b")),Const "a")*)
(*(string * terme) list =
[("x", Binop ("f", Const "a", Const "b"));("y", Binop ("f", Const "a", Con
*)
```

4.4. **Les interfaces et les tests de Q4.** deux fonctions

(** compare
@param : deux liste l1 l2
@return : true si l2 appartient à l1 false sinon
*)

```
let rec compare l1 l2 = match (l1,l2) with
| ([],[]) ->true
| (_,[]) -> true
| ([],_) -> false
| (h1::r1,h2::r2)-> if h1 = h2 then (compare r1 r2) else false;;
```

(** test_sys
@param : deux deux terme tg td
@return : true si un système de réécriture est bien formé
*)

```
 let test_sys tg td = (test_terme_s sign_plus tg) &&
 (test_terme_s sign_plus td) && (if_motif tg)
 && (if_mo tif td) && let (l1,l2) = (list_var tg,list_var td)
in compare l1 l2 ;;
```

```

(**----------test----------*
let tg1 = Binop("plus",Var "x",Const "zero");;
let td1 = Var "x";;
let tg2 = Binop("plus",Const "zero", Var "x");;
```

```
let td2 = Var "x";;
let tg3 = Binop("plus",Unop ("succ",Var "x"),Var "y");;
let td3 = Binop("plus",Var "x",Unop ("succ",Var "y"));;

test_sys tg1 td1;; (*bool = true*)
test_sys tg2 td2;; (*bool = true*)
test_sys tg3 td3;; (*bool = true*)
*)
```

4.5. **Les interfaces et les tests de Q5.** huit fonctions

```
(** test_sys
@param   : un couple l
@return  : return premier element de l
*)
let fst l = match l with (c1,c2) -> c1;;

(** test_sys
@param   : un couple l
@return  : return 2ieme element de l
*)
let snd l = match l with (c1,c2) -> c2;;

(** sub_override
* -  : terme -> terme -> (string * terme) list = <fun>
* - the override for substitution so that we can
    trait the result for the case of "fail"
@param   : un motif m un terme t
@return  : une liste de type (string*terme)
*)
let rec sub_override m t  = match (m, t) with
  | (Var v1, str) -> [(v1, str)]
  | (Const s1, Const s2) ->if s1 = s2 then []
                           else [("false",Var "false")]
  | (Unop(s1, t1), Unop(s2, t2)) -> if s1 = s2 then (sub_override t1 t2)
                           else [("false",Var "false")]
  | (Binop(s1, g1, d1), Binop(s2, g2, d2)) -> if s1 = s2
                           then (sub_override g1 g2)@(sub_override d1 d2)
                           else [("false",Var "false")]
  | _ -> [("false",Var "false")];;

```

```
(** if_exsist_sub
- : (string * 'a) list -> bool = <fun>
@param  : une liste l qui conient le resultat de sub_override
@return : true si il n'y a pas de false, false sinon
*)
let rec if_exsist_sub l =  match l with
| [] ->true
| h::r -> if (fst h) = "false" then false else if_exsist_sub r;;

(** remplace
- : string * terme -> terme -> terme = <fun>
@param  : un couple s de type string*terme , un terme td
@return : remplace td en fonction de s
*)
let rec remplace s td = match td with
| Var x -> if x = (fst s) then (snd s) else Var x
| Const a-> Const a
| Unop (u,t) -> Unop(u,(remplace s t))
| Binop(b,t,d) -> Binop(b,(remplace s t),(remplace s d) );;

(** app_une_regle
- : terme * terme -> terme -> terme = <fun>
- : n'applique qu'une regle pour un terme t
@param  : une regle regle_1 un terme t
@return : un terme qui est transforme par regle_1
*)
let app_une_regle regle_1 t = let l = (sub_override (fst regle_1) t) in
if (if_exsist_sub l) then List.fold_right
(function s -> function td-> (remplace s td)) l (snd regle_1) else t;;

(** app_de_la_regle
- : (terme * terme) list -> terme -> terme = <fun>
- : applique tous les regles dans regle pour t
@param  : une liste de regle regle un terme t
@return : un terme qui est transforme par regle une fois
*)
let app_de_la_regle regle t = List.fold_right
(function r-> function t-> let l =  (sub_override (fst r) t) in
if (if_exsist_sub l) then app_une_regle r t else t ) regle t;;

(** app_regle
```

```
72 - : (terme * terme) list -> terme -> terme = <fun>
73 @param  : une liste de regle regle un terme t
74 @return : un terme qui est transforme par regle jusqu'a qu'aucune des
75 regles de la liste ne s'applique plus
76 *)
77 let rec app_regle regle t =
78   let t_i = app_de_la_regle regle t in if ( t_i = t ) then t else app_regle
```

```
1
2  (**----------test----------*
3
4  let regle_plus = [(Binop("plus",Var "x",Const "zero"),Var "x");
5  (Binop("plus",Const "zero", Var "x"),Var "x");
6  (Binop("plus",Unop ("succ",Var "x"),Var "y"),
7  Binop("plus",Var "x",Unop ("succ",Var "y")))];;
8
9  let regle_piles = [(Unop("top",Binop("push",
10 Unop("succ",Const "zero"),Var "y")),Var "x");
11 (Unop("pop",Binop("push",Unop("succ",Const "zero"),Var "y")),Var "y");
12
13 (Binop("alternate",Const "vide",Var "z"),Var "z");
14 (Binop("alternate",Binop("push",Unop("succ",Const "zero"),
15 Var "y"),Var "z"),Binop("push",Unop("succ",Const "zero"),
16 Binop("alternate",Var "y",Var "z"))) ];;
17
18 let t = Binop("plus",Const "zero",Binop("plus",
19 Unop("succ",Const "zero"),Unop("succ",Const "zero")));;
20
21 let p_1= Unop("top",Binop("push",
22 Unop("succ",Const "zero"),Var "y"));;
23 let p_2 =Binop("alternate",Binop("push",
24 Unop("succ",Const "zero"),Var "y"),Var "z");;
25
26 app_regle regle_plus t (* Unop ("succ",
27  Unop ("succ", Const "zero"))*)
28
29 app_regle regle_piles p_1;; (*Var "x"*)
30 app_regle regle_piles p_2;; (* Binop ("push", Unop ("succ", Const "zero"),
31  Binop ("alternate", Var "y", Var "z"))*)
32
33 sub_override (fst r1) t;;
```

```
34 if_exsist_sub (sub_override (fst r2) t);;
35 app_de_la_regle regle t;;
36
37 app_une_regle r2 t2;;
38 sub_override  (fst r2) terme;;
39 *)
```

4.6. **Les interfaces et les tests de Q6.** cinq fonctions

```
1  (**terme_to_string
2   - : terme -> string = <fun
3   @param : un terme t
4   @return : un string
5   *)
6  let rec terme_to_string t = match t with
7   |Const a -> a
8   |Var x ->x
9   |Unop (a,b) ->a ^"(" ^ (terme_to_string b) ^")"
10  |Binop(a,b,c)->
11  a ^ "(" ^ (terme_to_string b) ^ ", "^ (terme_to_string c) ^ ")";;
12
13 (**print_terme
14  - : terme -> int -> unit
15  @param : un terme t un entier n
16  @imprimer le terme lisible
17  *)
18
19 let print_terme t n =
20 Printf.printf "par r%d : %s \n" n (terme_to_string t);;
21
22 (**count_n
23  - : (terme * 'a) list -> terme -> int -> int
24  @param : une regle un terme un entier n
25  @return un entier qui indique que niem regle peut filtre t
26  *)
27 let rec count_n regle t n = match regle with
28 | [] -> (0,t)
29 | h::r -> let l = (sub_override (fst h) t) in
30           if (if_exsist_sub l) then (n,(app_une_regle h t))
31           else count_n r t (n+1);;
32
```

```
(**couple_print
 - : (terme * terme) list -> terme -> int -> (int * terme) list
 @param : une regle un terme un entier n
 @return une liste de tpye (int*terme) qu'on va print
 *)
let rec couple_print regle t n= let (n,ti) = (count_n regle t n) in
        if (n != 0) then (n,ti)::(couple_print regle ti 1)
        else [];;


(**print
 - : (terme * terme) list -> terme -> unit list
 @param : une regle un terme
 @return une liste de tpye unit qui contient les etape de
    simplicaiton
 *)
let rec print regle t = let l = (couple_print regle t 1) in List.map
        (function x-> print_terme (snd x) (fst x)) l ;;


(**----------test----------*
print regle t ;;

(*t =Binop ("plus", Const "zero",Binop ("plus", Unop ("succ",
Const "zero"), Unop ("succ", Const "zero")))*)

(*par r2 : plus(succ(zero), succ(zero))
par r3 : plus(zero, succ(succ(zero)))
par r2 : succ(succ(zero))*)
*)
```

4.7. **Les interfaces et les tests de Q7.** une fonctions

```
(**compare_terme
 - : terme -> terme -> bool
 @param : deux terme
 @return true si ils sont identiques apres simplification false sinon
 *)
let compare_terme t1 t2 = let (ts_1,ts_2) =
(app_regle regle t1,app_regle regle t2) in ts_1 = ts_2 ;;

```

```
(**----------test----------*

(**
* t = Binop("plus",Const "zero",Binop("plus",
Unop("succ",Const "zero"),Unop("succ",Const "zero")));;
* app_regle regle t;;
* t1 =Binop ("plus", Unop ("succ", Const "zero"),
 Unop ("succ", Const "zero"));;
*)
compare_terme t1 t;; (*- : bool = true*)

(**
  * t2 = Unop("succ",Const "zero")
*)
let t2 = Unop("succ",Const "zero");;
compare_terme t2 t;; (*- : bool = false*)
*)
```

4.8. **Les interfaces et les tests de Q8.** une fonctions

```
let p= Binop("plus",Binop("plus",Const "zero",
Unop("succ",Const "zero")),Unop("succ",Const "zero"));;

let rec new_app_regle regle t =
  let t_a = app_regle regle t in if t_a <> t then t_a
    else match t_a with
    | Var v -> Var v
    | Const a -> Const a
    | Unop(u,s)-> Unop(u,s)
    | Binop(b,s1,s2) ->
      new_app_regle regle (Binop (b,new_app_regle regle s1,
      new_app_regle regle s2));;

(**----------test----------*
new_app_regle regle p;;
(*terme = Unop ("succ", Unop ("succ", Const "zero"))*)
```