

Typage et Analyse Statique

Cours 3

Emmanuel Chailloux

Spécialité Science et Technologie du Logiciel
Master mention Informatique
Université Pierre et Marie Curie

année 2018-2019

Plan du cours 3

Polymorphisme paramétrique:

- ▶ typage d'un mini-ML
 - ▶ noyau fonctionnel
 - ▶ extension impérative

du λ -calcul aux langages de programmation (1)

Langage d'expressions : λ -calcul auquel on ajoute :

- ▶ des constantes : entiers, booléens
- ▶ des opérateurs arithmétiques et logiques
- ▶ une stratégie d'évaluation : immédiate (ou retardée)
- ▶ une forme spéciale pour la conditionnelle (ITE)
- ▶ nommage d'une expression (LET IN) avec portée statique
- ▶ déclaration récursive (LETREC IN)
- ▶ structures de données :
 - ▶ couples : constructeur et accesseurs (fst, snd)
 - ▶ listes : constructeurs ([], ::), et accesseurs (hd, tl)

On peut y ajouter les déclarations globales, et bien sûr un système de types.

Un compilateur ML se décompose de manière classique en :

- ▶ une analyse lexicale puis syntaxique du texte source en entrée qui construit un arbre de syntaxe ;
- ▶ un vérificateur de types qui infère les types des expressions et détecte donc les erreurs de typage ;
- ▶ un producteur de code assembleur, souvent d'une machine abstraite ;
- ▶ et d'une phase d'assemblage pour exécuter ce code sur un processeur existant.

On se propose dans cette section de définir la syntaxe abstraite d'un langage appelé mini-ML,

mini-ML

Mini-ML est uniquement un langage d'expressions. Elles correspondent à une extension du λ -calcul par les constantes numériques, de chaînes de caractères et des booléens, des couples et des listes, de la structure de contrôle conditionnelle, de la déclaration locale `let` qui introduit le polymorphisme et de la récursion.

```
1  type ml_expr = Const of ml_const
2    | Var of string
3    | Pair of ml_expr * ml_expr
4    | Cons of ml_expr * ml_expr
5    | Cond of ml_expr * ml_expr * ml_expr
6    | App of ml_expr * ml_expr
7    | Abs of string * ml_expr
8    | Letin of string * ml_expr * ml_expr
9    | Letrecin of string * ml_expr * ml_expr
10 and ml_const = Int of int
11   | Float of float
12   | Bool of bool
13   | String of string
14   | Emptylist
15   | Unit;;
```

Syntaxe des types (1)

L'ensemble des types (\mathcal{T}) de mini-ML est construit à partir d'un ensemble de constantes de types (\mathcal{C}), un ensemble de variables de types (\mathcal{P}) et de 3 constructeurs de types (\rightarrow , $*$ et *list*).

- ▶ constante de type :
Int_type, Float_type, String_type, Bool_type et Unit_type
 $\in \mathcal{C}$: si $\alpha \in \mathcal{C}$ alors $\alpha \in \mathcal{T}$;
- ▶ variable de type $\in \mathcal{P}$: si $\alpha \in \mathcal{P}$ alors $\alpha \in \mathcal{T}$;
- ▶ type produit : si $\sigma, \tau \in \mathcal{T}$ alors $\sigma * \tau \in \mathcal{T}$;
- ▶ type liste : si $\sigma \in \mathcal{T}$ alors $\sigma \text{ list} \in \mathcal{T}$;
- ▶ type fonctionnel : si $\sigma, \tau \in \mathcal{T}$ alors $\sigma \rightarrow \tau \in \mathcal{T}$;

Syntaxe des types (2)

La syntaxe abstraite des types de mini-ML est la suivante :

```
1  type vartype =  
2    | Unknown of int  
3    | Instanciated of ml_type  
4  and consttype =  
5    | Int_type  
6    | Float_type  
7    | String_type  
8    | Bool_type  
9    | Unit_type  
10 and ml_type =  
11 | Var_type of vartype ref  
12 | Const_type of consttype  
13 | Pair_type of ml_type * ml_type  
14 | List_type of ml_type  
15 | Fun_type of ml_type * ml_type;;
```

Où une variable de type est une référence sur une inconnue ou un type. Cela permettra d'effectuer une modification de types lors de l'application d'une substitution sur une variable de type.

La fonction de création de variables de type est la suivante :

```
1 let new_unknown, reset_unknowns =  
2   let c = ref 1 in  
3     ( function () -> c:= !c+1; Var_type( ref(Unknown !c))),  
4     ( function () -> c:=1) ;;
```

L'ensemble des types qui vient d'être décrit correspondant aux types simples étendus par les constantes de type et les constructeurs pour les paires et les listes. Il est encore nécessaire de définir les schémas de type (type `quantified_type`).

```
1 type quantified_type = Forall of (int list) * ml_type;;
```


Environnement de typage

L'environnement de typage, nécessaire lors du typage pour les variables libres (comme les primitives ou les paramètres formels) d'expressions est représenté par une liste d'association

(string * quantified_type) list.

Il est nécessaire de faire une distinction entre les variables d'un type si elles sont quantifiées (variables liées) ou si elles ne le sont pas (variables libres).

```
1 let rec vars_of_type t =
2   let rec vars vl = function
3     Const_type _ -> vl
4   | Var_type vt ->
5     ( match !vt with
6       Unknown n -> if List.mem n vl then vl else n::vl
7     | Instanciated t -> vars vl t
8     )
9   | Pair_type (t1,t2) -> vars (vars vl t1) t2
10  | List_type t -> vars vl t
11  | Fun_type (t1,t2) -> vars (vars vl t1) t2
12  in
13  vars [] t ;;
```

Variables libres et liées

Pour calculer les variables libres et liées d'un schéma de type,

```
1  let subtract l1 l2 =
2    List.flatten (List.map (function id ->
3                          if (List.mem id l2) then [] else [id])
4                          l1);;
5  let free_vars_of_type (bv,t) =
6    subtract (vars_of_type t) bv
7  and bound_vars_of_type (fv,t) =
8    subtract (vars_of_type t) fv ;;
9  let append l1 l2 = l1@l2;;
10 let flat ll = List.fold_right append ll [];;
11 let free_vars_of_type_env l =
12   flat ( List.map (function (id,Forall (v,t))
13                   -> free_vars_of_type (v,t)) l) );;
```

Instance d'un schéma de type

Lors du typage d'un identificateur dans un environnement C , le vérificateur cherche dans l'environnement le type quantifié associé à cet identificateur et en retourne une instance, où les variables quantifiées valent de nouvelles variables (ou inconnues).

```
1 let type_instance st =
2   match st with Forall(gv,t) ->
3     let unknowns = List.map (function n -> n,new_unknown()) gv
4     in
5     let rec instance = function
6       Var_type {contents=(Unknown n)} as t ->
7         (try List.assoc n unknowns with Not_found -> t)
8     | Var_type {contents=(Instancié t)} -> instance t
9     | Const_type ct as t -> t
10    | Pair_type (t1,t2) -> Pair_type (instance t1, instance t2)
11    | List_type t -> List_type (instance t)
12    | Fun_type (t1,t2) -> Fun_type (instance t1, instance t2)
13  in
14    instance t ;;
```

Erreurs de typage

Les exceptions suivantes sont levées quand une erreur de type est détectée :

```
1 type typing_error =  
2   Unbound_var of string  
3 | Clash of ml_type * ml_type ;;  
4 exception Type_error of typing_error;;
```

Elles correspondent soit à une variable d'une expression qui n'a pas été déclarée, soit à une erreur proprement dite de typage.

Unification

La grande différence dans l'algorithme de typage du λ -calcul et celui présenté ici provient de l'algorithme d'unification. dans le premier cas, la fonction TYPE retournait la liste des substitutions (l'unificateur principal UP) à appliquer pour que deux termes t_1 et t_2 vérifient $UP(t_1) = UP(t_2)$. Cet algorithme bien que juste n'est pas forcément très efficace. En mini-ML, Les substitutions trouvées sont immédiatement appliquées aux types en effectuant une modification physique de ces types.

Deux problèmes se posent :

- ▶ la vérification d'occurrences, quand une variable de type est remplacée par un autre type, ce second type ne doit pas contenir d'occurrence de cette première variable. La fonction occurs réalise ce test.
- ▶ Lorsqu'une inconnue se voit attribuer comme valeur une autre inconnue, on crée une chaîne d'indirection qu'il faut alors simplifier : `Var_type (ref (Var_type (ref (Instancié t))))` deviendra alors seulement `t`.

Fonctions auxiliaires (1)

La fonction shorten réalise ce travail.

```
1 let occurs n t = List.mem n (vars_of_type t);;
2 let rec shorten = function
3   Var_type (vt) as tt ->
4     (match !vt with
5       Unknown _ -> tt
6       | Instanciated ((Var_type _) as t) ->
7         let t2 = shorten t in
8           vt := Instanciated t;
9           t2
10      | Instanciated t -> t
11    )
12 | t -> t;;
```

Fonctions auxiliaires (2)

La fonction `unify_types` prend deux types, modifie leur taille (par la fonction `shorten`) et les rend égaux ou échoue.

```
1 let rec unify_types (t1,t2) =
2   let lt1 = shorten t1 and lt2 = shorten t2
3   in
4     match (lt1,lt2) with
5     | Var_type ( {contents=Unknown n} as occn ),
6       Var_type {contents=Unknown m} ->
7       if n=m then () else occn:= Instanciated lt2
8     | Var_type ({contents=(Unknown n)} as occn), _ ->
9       if occurs n lt2
10      then raise (Type_error(Clash(lt1,lt2)))
11      else occn:=Instanciated lt2
12     | _ , Var_type ({contents=(Unknown n)}) -> unify_types (lt2,lt1)
13     | Const_type ct1, Const_type ct2 ->
14       if ct1=ct2 then () else raise (Type_error(Clash(lt1,lt2)))
15     | Pair_type (t1,t2), Pair_type (t3,t4) ->
16       unify_types (t1,t3); unify_types(t2,t4)
17     | List_type t1, List_type t2 -> unify_types (t1,t2)
18     | Fun_type (t1,t2), Fun_type (t3,t4) ->
19       unify_types (t1,t3); unify_types(t2,t4)
20     | _ -> raise(Type_error(Clash(lt1,lt2)));;
```

Typage des expressions (1)

Le typage des constantes est très simple à l'exception de la liste vide. Ces règles de typage sont toutes considérées comme des axiomes.

(ConstInt)

$C \vdash \text{nombre entier} : \text{Int}$

(ConstFloat)

$C \vdash \text{nombre flottant} : \text{Float}$

(ConstString)

$C \vdash \text{chaîne} : \text{String}$

(ConstBool)

$C \vdash \text{booléen} : \text{Bool}$

(ConstUnit)

$C \vdash () : \text{Unit}$

Typage des expressions (2)

Ces règles se traduisent très simplement en OCaml :

```
1 let type_const = function
2   Int _ -> Const_type Int_type
3 | Float _ -> Const_type Float_type
4 | String _ -> Const_type String_type
5 | Bool _ -> Const_type Bool_type
6 | Unit -> Const_type Unit_type
7 | Emptylist -> List_type (new_unknown()) ;;
```

On introduit deux fonctions *instance* qui retourne une instance de type à partir d'un schéma de types et *generalize* qui crée un schéma de types à partir d'un type et d'un environnement.

Instance et généralisation (1)

La fonction *instance* correspond à la fonction OCaml `type_instance`. Elle remplace les variables de types quantifiées d'un schéma de type par de nouvelles variables de types.

```
1 let type_instance st =
2   match st with Forall(gv,t) ->
3   let unknowns = List.map (function n -> n,new_unknown()) gv
4   in
5     let rec instance t = match t with
6       Var_type (vt) as tt ->
7         ( match !vt with
8           Unknown n ->(try List.assoc n unknowns with Not_found -> tt)
9           | Instanciated ti -> instance ti
10          )
11       | Const_type tc -> t
12       | List_type t1 -> List_type (instance t1)
13       | Fun_type (t1,t2) -> Fun_type (instance t1, instance t2)
14       | Pair_type (t1,t2) -> Pair_type (instance t1, instance t2)
15   in
16     instance t
17 ;;
```

Instance et généralisation (2)

La fonction *generalize* correspond à la fonction OCaml suivante :

```
1 let generalize_types gamma l =  
2   let fvg = free_vars_of_type_env gamma  
3   in  
4     List.map (function (s,t) ->  
5               (s, Forall(free_vars_of_type (fvg,t),t))) l  
6   ;;
```

Elle crée un schéma de type en quantifiant les variables libres d'un type qui n'appartiennent pas aux variables libres de l'environnement.

Règles de typage (1)

Les règles de typages des expressions sont les règles de typage des constantes et les règles suivantes :

(Var)

$$x : \sigma, C \vdash x : \tau \quad \tau = \text{instance}(\sigma)$$

(Pair)

$$\frac{C \vdash e_1 : \tau_1 \quad C \vdash e_2 : \tau_2}{C \vdash (e_1, e_2) : \tau_1 * \tau_2}$$

(List)

$$\frac{C \vdash e_1 : \tau \quad C \vdash e_2 : \tau \text{ list}}{C \vdash (e_1 :: e_2) : \tau \text{ list}}$$

(If)

$$\frac{C \vdash e_1 : \text{Bool} \quad C \vdash e_2 : \tau \quad C \vdash e_3 : \tau}{C \vdash (\text{if } e_1 \text{ then } e_2 \text{ else } e_3) : \tau}$$

(App)

$$\frac{C \vdash e_1 : \tau' \rightarrow \tau \quad C \vdash e_2 : \tau'}{C \vdash (e_1 e_2) : \tau}$$

(Abs)

$$\frac{x : \tau_1, C \vdash e : \tau_2}{C \vdash (\text{function } x \rightarrow e) : \tau_1 \rightarrow \tau_2}$$

Règles de typage (2)

(Let)

$$\frac{C \vdash e_1 = \tau_1 \quad \sigma = \text{generalize}(\tau_1, C) \quad (x : \sigma), C \vdash e_2 : \tau_2}{C \vdash \text{let } x = e_1 \text{ in } e_2 : \tau_2}$$

(LetRec)

$$\frac{(x : \alpha), C \vdash e_1 = \tau_1 \quad \alpha \notin V(C) \quad \sigma = \text{generalize}(\tau_1, C) \quad (x : \sigma), C \vdash e_2 : \tau_2}{C \vdash \text{letrec } x = e_1 \text{ in } e_2 : \tau_2}$$

Le polymorphisme est introduit dans les règles *Let* et *LetRec*. La règle *LetRec* suppose que la variable définie récursivement (x) est du type α , nouvelle variable de type qui n'apparaît pas libre dans l'environnement. Ce qui ne fait aucune supposition sur le type de x , mais dans la mesure où le type de x n'est pas un schéma de type, les contraintes de type sur x dans e_1 seront bien répercutées sur le type final de e_1 .

Fonction type_expr (1)

La fonction type_expr suivante suit les règles de typage définies ci-dessus.

```
1 let rec type_expr gamma =
2   let rec type_rec expri =
3     match expri with
4       Const c -> type_const c
5     | Var s -> let t = try List.assoc s gamma
6                 with Not_found -> raise (Type_error(Unbound_var s))
7                 in type_instance t
8     | Pair (e1,e2) -> Pair_type (type_rec e1, type_rec e2)
9     | Cons (e1,e2) ->
10      let t1 = type_rec e1
11      and t2 = type_rec e2 in
12      unify_types (List_type t1, t2); t2
13    | Cond (e1,e2,e3) ->
14      let t1 = unify_types (Const_type Bool_type, type_rec e1)
15      and t2 = type_rec e2
16      and t3 = type_rec e3 in
17      unify_types (t2,t3); t2
```

Fonction type_expr (2)

```
1 | App (e1,e2) ->
2   let t1 = type_rec e1
3   and t2 = type_rec e2 in
4     let u = new_unknown() in
5       unify_types (t1, Fun_type (t2,u)); u
6 | Abs(s,e) ->
7   let t = new_unknown() in
8     let new_env = (s,Forall ([],t))::gamma in
9       Fun_type (t, type_expr new_env e)
10 | Letin (s,e1,e2) ->
11   let t1 = type_rec e1 in
12     let new_env = generalize_types gamma [ (s,t1) ] in
13       type_expr (new_env@gamma) e2
14 | Letrecin (s,e1,e2) ->
15   let u = new_unknown () in
16     let new_env = (s,Forall([ ],u))::gamma in
17     let t1 = type_expr (new_env@gamma) e1 in
18     let final_env = generalize_types gamma [ (s,t1) ] in
19     type_expr (final_env@gamma) e2
20 in
21   type_rec;;
```

Environnement initial

```
1  let mk_type (ct1,ct2,ct3) =
2    Forall([],
3      Fun_type (Pair_type(Const_type ct1, Const_type ct2),Const_type ct3)) in
4  let int_ftype = mk_type(Int_type,Int_type,Int_type)
5  and float_ftype = mk_type(Float_type,Float_type,Float_type)
6  and int_predtype = mk_type(Int_type,Int_type,Bool_type)
7  and float_predtype = mk_type(Float_type,Float_type,Bool_type)
8  and alpha = Var_type(ref(Unknown 1))
9  and beta = Var_type(ref(Unknown 2)) in
10 ("=",Forall([ 1 ],
11   Fun_type (Pair_type (alpha,alpha), Const_type Bool_type)))):
12 (List.map (function s -> (s,int_ftype)) [ "*" ; "+" ; "-" ; "/" ]) @
13 (List.map (function s -> (s,float_ftype)) [ "*." ; "+." ; "-." ; "/" ]) @
14 (List.map (function s -> (s,int_predtype)) [ "<" ; ">" ; "<=" ; ">=" ]) @
15 (List.map (function s -> (s,float_predtype)) [ "<." ; ">." ; "<=." ; ">=." ]) @
16 [ "^", mk_type (String_type, String_type, String_type) ] @
17 [ ("hd",Forall([ 1 ], Fun_type (List_type alpha, alpha)));
18 ("tl",Forall([ 1 ], Fun_type (List_type alpha, List_type alpha)));
19 ("null",Forall([ 1 ], Fun_type (alpha,
20 Fun_type (List_type alpha, Const_type Bool_type))));
21 ("fst",Forall([ 1; 2 ], Fun_type (Pair_type (alpha,beta),alpha)));
22 ("snd",Forall([ 1; 2 ], Fun_type (Pair_type (alpha,beta),beta)) ] ;;
```


Impression des types et des erreurs (1)

Comme les variables de types sont codées à l'aide d'entiers, il serait particulièrement désagréable d'avoir un affichage de types dépendant de cette numérotation. Pour cela, pour chaque expression globale est créée une liste d'association des numéros de variables de types et des symboles pour les types polymorphes.

Les constantes de types sont affichées directement par la fonction `print_consttype` :

```
1 let print_consttype = function
2   Int_type -> print_string "int"
3 | Float_type -> print_string "float"
4 | String_type -> print_string "string"
5 | Bool_type -> print_string "bool"
6 | Unit_type -> print_string "unit";;
```

Impressions des types et des erreurs (2)

La fonction `var_name` retourne une chaîne de caractères unique pour chaque entier passé en argument :

```
1 let string_of_char c =  
2   let s = " " in s.[0] <- c; s;;  
3 let var_name n =  
4   let rec name_of n =  
5     let q,r = ((n / 26), (n mod 26)) in  
6     if q=0 then string_of_char(Char.chr (96+r))  
7     else (name_of q)^(string_of_char(Char.chr (96+r)))  
8   in  
9     ""^(name_of n);;
```

La fonction principale est l'impression d'un schéma de type. Le seul problème est de trouver le nom d'une variable de types. la fonction locale `names_of` retourne les variables quantifiées, chacune associée à une chaîne de caractères particulière. Si une variable de type n'est pas donc cet ensemble une exception est déclenchée, car il ne peut avoir d'expression globale contenant des variables de type non quantifiées.

Impression des types et des erreurs (3)

```
1 let print_quantified_type (Forall (gv,t)) =
2   let names =
3     let rec names_of = function
4       (n,[]) -> []
5       | (n,(v1::lv)) -> (var_name n)::(names_of (n+1,lv))
6     in (names_of (1,gv))
7   in
8     let var_names = List.combine (List.rev gv) names in
9     let rec print_rec = function
10      Var_type {contents=(Instanciated t)} -> print_rec t
11      | Var_type {contents=(Unknown n)} ->
12        let name =
13          ( try List.assoc n var_names
14            with Not_found ->
15              raise (Failure "Non quantified variable in type"))
16        in print_string name
17      | Const_type ct -> print_consttype ct
18      | Pair_type(t1,t2) -> print_string "("; print_rec t1;
19        print_string " * "; print_rec t2; print_string ")"
20      | List_type t ->
21        print_string "("; print_rec t; print_string ") list)"
22      | Fun_type(t1,t2) -> print_string "("; print_rec t1;
23        print_string " -> "; print_rec t2; print_string ")"
24   in print_rec t;
```

Impression des types et des erreurs (4)

L'impression d'un type simple utilise l'impression des schémas de type en créant pour l'occasion une quantification sur toutes les variables libres du type.

```
1 let print_type t =  
2   print_quantified_type (Forall(free_vars_of_type ([],t),t));;
```

Impression des types et des erreurs (5)

Il est fort intéressant d'afficher les causes d'un échec de typage, comme par exemple les types mis en cause lors d'une erreur de typage ou le nom d'une variable indéfinie. La fonction `typing_handler` reçoit en argument une fonction de typage, un environnement et une expression à typer. Elle capture les exceptions dues à une erreur de typage, affiche un message plus clair et déclenche une nouvelle exception.

```
1 let typing_handler typing_fun env expr =
2   reset_unknowns();
3   try typing_fun env expr
4   with
5     Type_error (Clash(lt1,lt2)) ->
6       print_string "Type clash between "; print_type lt1;
7       print_string " and "; print_type lt2; print_newline();
8       failwith "type_check"
9   | Type_error (Unbound_var s) ->
10     print_string "Unbound variable ";
11     print_string s; print_newline();
12     failwith "type_check";;
```

Fonction principale

La fonction `type_check` type l'expression `e` par la fonction `type_expr` avec l'environnement de typage `initial` et affiche le type calculé de `e`.

```
1 let type_check e =  
2   let t =  
3     typing_handler type_expr initial_typing_env e in  
4   let qt =  
5     snd(List.hd(generalize_types initial_typing_env ["it",t])) in  
6     print_string "it : "; print_quantified_type qt; print_newline();;
```

Exemples

```
1 type_check (Const (Float 3.2));;
2 type_check (Abs ("x", Var "x"));;
3 type_check (Letin ("f", Abs ("x", Pair(Var "x", Var "x")),
4             ( Pair ((App (Var "f", Const (Int 3))),
5                     (App (Var "f", Const (Float 3.14))))));;
6 type_check (App (Var "*", Pair (Const (Int 3), Const (Float 3.1))));;
7 type_check (Cond (
8     App (Var "=", Pair(Const(Int 0), Const (Int 0))),
9     Const(Int 2), Const(Int 5))));;
10 type_check (
11   Letrecin ("fact",
12     Abs ("x",
13       Cond (App (Var "=",Pair(Var "x",Const(Int 1))),
14             Const(Int 1),
15             App (Var "*",
16                 Pair(Var "x",
17                     App (Var "fact",
18                         App (Var "-",Pair(Var "x",Const(Int 1) ))
19                     )
20                 )
21             )
22         )
23     ),
24   App (Var "fact", Const (Int 4))));;
```

Remarques sur le typage d'OCaml

Le typeur de mini-ML ne tenait pas compte des valeurs mutables (physiquement modifiables). OCaml est un langage fonctionnel muni d'effets de bord.

Si les primitives de constructions de valeurs mutables étaient traités comme des valeurs polymorphes classiques le *typeur* ne détecterait pas les inconsistances de types. Le programme suivant en est un exemple :

```
1 let x = ref (function x -> x)
2 in
3   x:=(function x -> x + 1);
4   !x true;;
```

et entraîne une erreur de typage.

Polymorphisme et valeurs mutables

- ▶ Une solution serait de ne construire que des valeurs mutables monomorphes.
- ▶ Comme cette contrainte est trop forte, il est possible de créer des valeurs mutables polymorphes spéciales, en utilisant des variables de types faibles pour celles-ci. L'idée est d'assurer lors de l'instanciation de cette variable de type, dans d'une application, que l'on obtient un type simple ou une variable de type liée au contexte de typage.
- ▶ Pour cela on différencie les expressions en deux classes : les expressions *non expansives* incluant principalement les variables, les constructeurs et l'abstraction et les expressions *expansives* incluant principalement l'application. Seules les expressions *expansives* peuvent engendrer une exception ou étendre le domaine (incohérence de types).

Expressions non-expansives (1)

Voici le texte de la fonction de reconnaissance d'expression non expansives du compilateur OCaml 3.0 (fichier typecore.ml) :

```
1  let rec is_nonexpansive exp =
2    match exp.exp_desc with
3      Texp_ident(_,_) -> true
4    | Texp_constant _ -> true
5    | Texp_let(rec_flag, pat_exp_list, body) ->
6      List.for_all (fun (pat, exp) -> is_nonexpansive exp) pat_exp_list &
7        is_nonexpansive body
8    | Texp_apply(e, None::el) ->
9      is_nonexpansive e &&
10     List.for_all (function None -> true | Some exp -> is_nonexpansive e) el
11   | Texp_function _ -> true
12   | Texp_tuple el ->
13     List.for_all is_nonexpansive el
14   | Texp_construct(_, el) ->
15     List.for_all is_nonexpansive el
```

Expressions non-expansives (2)

```
1 | Texp_variant(_, Some e) -> is_nonexpansive e
2 | Texp_variant(_, None) -> true
3 | Texp_record(lbl_exp_list, opt_init_exp) ->
4 |   List.for_all
5 |     (fun (lbl, exp) -> lbl.lbl_mut = Immutable & is_nonexpansive exp)
6 |     lbl_exp_list &&
7 |     (match opt_init_exp with None -> true | Some e -> is_nonexpansive e)
8 | Texp_field(exp, lbl) -> is_nonexpansive exp
9 | Texp_array [] -> true
10 | Texp_new (_, cl_decl) when Ctype.class_type_arity cl_decl.cty_type > 0 ->
11 |   true
12 | _ -> false ;;
```

Les expressions suivantes sont non-expansives (donc pourront être généralisées :

- ▶ les constantes, les identificateurs,
- ▶ les n-uplet si toutes les sous-expressions le sont, les constructeurs constants, les constructeurs d'arité 1 si ce constructeur n'est pas mutable et si l'expression passée au constructeur est elle aussi non-expansive,

Expressions non-expansives (3)

- ▶ les déclarations locales (`let` et `let rec`) si l'expression finale et toute les expressions des déclarations locales sont non-expansives,
- ▶ l'abstraction,
- ▶ la récupération d'exceptions si l'expression à calculer et toutes les expressions de partie droite du filtrage le sont, la séquence si le dernier élément de la séquence l'est,
- ▶ la conditionnelle si les branches *then* et *else* le sont, les expressions contraintes par un type si la sous-expression l'est,
- ▶ le vecteur vide,
- ▶ les enregistrements si tous ces champs sont non-mutables et si chaque expression associée à un champ l'est,
- ▶ l'accès à un champ d'un enregistrement si l'expression correspondante à l'enregistrement l'est, les filtrages de flots et les gardes si l'expression associée l'est.

Expressions non-expansives (4)

Toutes les autres expressions sont expansives.

En OCaml on pourra généraliser (c'est-à-dire construire un schéma de type) les variables de type rencontrées dans des expressions *non expansives* et ne pas généraliser (variable de type faible notée `_ 'a`) celles rencontrées dans des expressions *expansives*.

Ces variables de type faible pourront être instanciées ultérieurement dans le typage d'un module. Il est à noter que ces variables de type faible ne peuvent sortir d'un module (sans le risque de réemplois ultérieurs incompatibles).

Exemples (1)

Voici deux séries d'exemples de typage d'expressions non-expansives et d'expressions expansives :

avec effets de bord

OCaml 4.0	commentaires
expression non expansive	
<pre>let nref x = ref x;; nref : 'a -> 'a ref = <fun></pre>	abstraction schéma de type
expression expansive	
<pre>let x = ref [];; x : '_a list ref = ref [] x:= [3];; x;; int list ref</pre>	ref est un constructeur mutable variable de type faible _a affinage du type de x

Exemples (2)

sans effets de bord

OCaml 4.0	commentaires
<pre>let a x y = x y;; ('a -> 'b) -> 'a -> 'b let g = a id;; '_a -> '_a g 3;; int = 3 g;; int -> int</pre>	<p>le combinateur A son schéma de type une application partielle son type non généralisé affinage du type de g</p>

Bibliographie

- ▶ R. Milner. “A theory of type polymorphism in programming”. J. Comput. Syst. Sci. 1978.
- ▶ X. Leroy. “Typage polymorphe d'un langage algorithmique”, Thèse de doctorat université Paris 7, 1992. décrit le typage polymorphe pour les effets de bords, les canaux de communications et les continuations dans un langage algorithmique fonctionnel.
- ▶ Wright (Andrew K.). “Polymorphism for Imperative Languages without Imperative Types”. Rapport technique n 93-200, Rice University, 1993. introduit la «value restriction» pour le typage des traits impératifs.
- ▶ Engel (Emmanuel). “Extensions sûres et praticables du système de types de ML en présence d'un langage de modules et de traits impératifs”. Thèse de doctorat université Paris 11, 1998. compare différents systèmes de typage polymorphe pour les effets de bords en présence de modules pour le langage Caml.
- ▶ Pierce (Benjamin). “Types and Programming Languages”. MIT Press, 2002.