Chapter 3

Typage d'un mini-ML

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, qui est un sous-ensemble de Caml-light et de montrer son mécanisme de typage, qui à des simplifications près est celui de Caml-light. Mini-ML correspond au noyau fonctionnel de Caml-light sans le filtrage. Il ne contient ni exceptions ni traits impératifs. Cette dernière restriction simplifie le vérificateur de types sans en changer la nature profonde. L'algorithme d'unification est assez différent de celui déjà vu pour le typage du λ-calcul car il appliquera directement, en faisant une modification physiques sur les types, les substitutions de variables de type. Le programme de vérification de types s'inspire des deux exemples de Michel Mauny que l'on peut lire dans la présentation d'une "spécification CAML d'un sous-ensemble de CAML" et du langage ASL de la documentation Caml-light (voir bibliographie).

3.1 Arbres de syntaxe abstraite de mini-ML

Mini-ML est uniquement un langage d'expression. Il ne contient pas de déclarations de variables globales, d'exceptions ou de types. Néanmoins, les expressions de mini-ML sont assez riches par rapport au λ-calcul. 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. Voici les types Caml-light des expressions et des déclarations :

```
# type ml_expr = Const of ml_const
```

```
    | Var of string
    | Pair of ml_expr * ml_expr
    | Cons of ml_expr * ml_expr
    | Cond of ml_expr * ml_expr * ml_expr
    | App of ml_expr * ml_expr
    | Abs of string * ml_expr
    | Letin of string * ml_expr * ml_expr
    | Letrecin of string * ml_expr * ml_expr
  and ml_const = Int of int
    | Float of float
    | Bool of bool
    | String of string
    | Emptylist
    | Unit;;
```

3.2 Rôle du vérificateur de types de mini-ML

La phase de typage de mini-ML s'insère entre l'analyseur syntaxique (qui construit un arbre de syntaxe abstraite) et le traducteur vers une machine abstraite. Les informations de typage ne seront pas utilisées dans la partie traduction dans un but de simplication du traducteur. Ainsi la fonction `type_check`, correspondant au vérificateur de types, n'aura qu'un rôle de vérification et s'arrêtera à la première erreur de types rencontrée.

3.3 Syntaxe des types de mini-ML

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 $\sigma \ list \in \mathcal{T}$;
- type fonctionnel : si $\sigma, \tau \in \mathcal{T}$ alors $\sigma \ \rightarrow \ \tau \in \mathcal{T}$;

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

```
# type vartype =
    | Unknown of int
    | Instanciated of ml_type
  and consttype =
    | Int_type
```

```
    | Float_type
    | String_type
    | Bool_type
    | Unit_type
  and ml_type =
    | Var_type of vartype ref
    | Const_type of consttype
    | Pair_type of ml_type * ml_type
    | List_type of ml_type
    | 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 :

```
# let new_unknown,reset_unknowns =
    let c = ref 1 in
      ( (function () → c:= !c+1; Var_type( ref(Unknown !c))),
        (function () → c:=1)) ;;
val new_unknown : unit -> ml_type = <fun>
val reset_unknowns : unit -> unit = <fun>
```

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`).

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

3.4 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).

```
# let rec vars_of_type t =
    let rec vars vl = function
      Const_type _ → vl
    | Var_type vt →
        ( match !vt with
```

```
            Unknown n → if List.mem n vl then vl else n::vl
          | Instanciated t → vars vl t
        )
    | Pair_type (t1,t2) → vars (vars vl t1) t2
    | List_type t → vars vl t
    | Fun_type (t1,t2) → vars (vars vl t1) t2
    in
      vars [] t ;;
val vars_of_type : ml_type -> int list = <fun>
```

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

```
# let subtract l1 l2 =
    List.flatten (List.map (function id →
                              if (List.mem id l2) then [] else [id])
                  l1);;
val subtract : 'a list -> 'a list -> 'a list = <fun>
# let free_vars_of_type (bv,t) =
    subtract (vars_of_type t) bv
  and bound_vars_of_type (fv,t) =
    subtract (vars_of_type t) fv ;;
val free_vars_of_type : int list * ml_type -> int list = <fun>
val bound_vars_of_type : int list * ml_type -> int list = <fun>
# let append l1 l2 = l1@l2;;
val append : 'a list -> 'a list -> 'a list = <fun>
# let flat ll = List.fold_right append ll [];;
val flat : 'a list list -> 'a list = <fun>
# let free_vars_of_type_env l =
      flat ( List.map (function (id,Forall (v,t))
                        → free_vars_of_type (v,t)) l) ;;
val free_vars_of_type_env : ('a * quantified_type) list -> int list = <fun>
```

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).

```
# let type_instance st =
    match st with Forall(gv,t) →
      let unknowns = List.map (function n → n,new_unknown()) gv
      in
        let rec instance = function
          Var_type {contents=(Unknown n)} as t →
            (try List.assoc n unknowns with Not_found → t)
```

```
        | Var_type {contents=(Instanciated t)} → instance t
        | Const_type ct as t → t
        | Pair_type (t1,t2) → Pair_type (instance t1, instance t2)
        | List_type t → List_type (instance t)
        | Fun_type (t1,t2) → Fun_type (instance t1, instance t2)
        in
          instance t ;;
val type_instance : quantified_type -> ml_type = <fun>
```

3.5 Erreurs de typage

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

```
# type typing_error =
    Unbound_var of string
  | Clash of ml_type * ml_type ;;
# 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.

3.6 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 $t1$ et $t2$ vérifient $UP(t1) = UP(t2)$. 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 (Instanciated t))))` deviendra alors seulement `t`. La fonction `shorten` réalise ce travail.

```
# let occurs n t = List.mem n (vars_of_type t);;
val occurs : int -> ml_type -> bool = <fun>
# let rec shorten = function
```

```
      Var_type (vt) as tt →
        (match !vt with
             Unknown _  → tt
           | Instanciated ((Var_type _) as t) →
               let t2 = shorten t in
                 vt := Instanciated t;
                 t2
           | Instanciated t → t
        )
    | t → t;;
val shorten : ml_type -> ml_type = <fun>
```

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

```
# let rec unify_types (t1,t2) =
     let lt1 = shorten t1 and lt2 = shorten t2
     in
       match (lt1,lt2) with
       | Var_type ( {contents=Unknown n} as occn ),
         Var_type {contents=Unknown m} →
           if n=m then () else occn:= Instanciated lt2
       | Var_type ({contents=(Unknown n)} as occn), _ →
           if occurs n lt2
           then raise (Type_error(Clash(lt1,lt2)))
           else occn:=Instanciated lt2
       | _ , Var_type ({contents=(Unknown n)}) → unify_types (lt2,lt1)
       | Const_type ct1, Const_type ct2 →
           if ct1=ct2 then () else raise (Type_error(Clash(lt1,lt2)))
       | Pair_type (t1,t2), Pair_type (t3,t4) →
           unify_types (t1,t3); unify_types(t2,t4)
       | List_type t1, List_type t2 →  unify_types (t1,t2)
       | Fun_type (t1,t2), Fun_type (t3,t4) →
          unify_types (t1,t3); unify_types(t2,t4)
       | _ →  raise(Type_error(Clash(lt1,lt2)));;
val unify_types : ml_type * ml_type -> unit = <fun>
```

3.7 Typage des expressions

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)

$$C \vdash nombre\ entier : Int$$

(ConstFloat)

$$C \vdash nombre\ flottant : Float$$

(ConstString)

$$C \vdash chaine : String$$

(ConstBool)

$$C \vdash booleen : Bool$$

(ConstUnit)

$$C \vdash () : Unit$$

(ConstListeVide)

$$C \vdash [] : \forall\alpha.\alpha\ list$$

Ces règles se traduisent très simplement en Caml :

```
# let type_const = function
    Int _ → Const_type Int_type
  | Float _ → Const_type Float_type
  | String _ → Const_type String_type
  | Bool _ →  Const_type Bool_type
  | Unit →  Const_type Unit_type
  | Emptylist → List_type (new_unknown()) ;;
val type_const : ml_const -> ml_type = <fun>
```

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.

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

```
# let type_instance st =
    match st with Forall(gv,t)  →
    let unknowns = List.map (function n → n,new_unknown()) gv
    in
      let rec instance t  = match t with
          Var_type (vt) as tt →
            ( match !vt with
```

```
                Unknown n →(try List.assoc n unknowns with Not_found → tt)
              | Instanciated ti → instance ti
            )
        | Const_type tc  → t
        | List_type t1 → List_type (instance t1)
        | Fun_type (t1,t2) → Fun_type (instance t1, instance t2)
        | Pair_type (t1,t2) → Pair_type (instance t1, instance t2)
      in
        instance t
  ;;
val type_instance : quantified_type -> ml_type = <fun>

# let generalize_types gamma l =
    let fvg = free_vars_of_type_env gamma
    in
      List.map (function (s,t) →
                  (s, Forall(free_vars_of_type (fvg,t),t))) l
  ;;
val generalize_types :
  ('a * quantified_type) list ->
  ('b * ml_type) list -> ('b * quantified_type) list = <fun>
```

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

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

(Var)

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

(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)

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

(If)

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

(App)

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

(Abs)

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

(Let)

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

(LetRec)

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

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.

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

```
# let rec type_expr gamma =
    let rec type_rec expri =
      match expri with
          Const c → type_const c
        | Var s → let t = try List.assoc s gamma
           with Not_found → raise (Type_error(Unbound_var s))
           in  type_instance t
        | Pair (e1,e2) → Pair_type (type_rec e1, type_rec e2)
        | Cons (e1,e2) →
            let t1 = type_rec e1
            and t2 = type_rec e2 in
              unify_types (List_type t1, t2); t2
        | Cond (e1,e2,e3) →
            let t1 = unify_types (Const_type Bool_type, type_rec e1)
            and t2 = type_rec e2
            and t3 = type_rec e3 in
              unify_types (t2,t3); t2
        | App (e1,e2) →
```

```
            let t1 = type_rec e1
            and t2 = type_rec e2 in
              let u = new_unknown() in
                unify_types (t1, Fun_type (t2,u)); u
        | Abs(s,e) →
            let t = new_unknown() in
              let new_env = (s,Forall ([],t)) :: gamma in
                Fun_type (t, type_expr new_env e)
        | Letin (s,e1,e2) →
            let t1 = type_rec e1 in
              let new_env = generalize_types gamma [ (s,t1) ] in
                type_expr (new_env@gamma) e2
        | Letrecin (s,e1,e2) →
            let u = new_unknown () in
              let new_env = (s,Forall([ ],u)) :: gamma in
                let t1 = type_expr (new_env@gamma) e1 in
                  let final_env = generalize_types gamma [ (s,t1) ] in
                    type_expr (final_env@gamma) e2
    in
      type_rec;;
val type_expr : (string * quantified_type) list -> ml_expr -> ml_type = <fun>
```

3.8 Environnement initial

Le langage mini-ML possède un certain nombre de fonctions prédéfinies. Ces fonctions munies de leur type correspondent à l'environnement initial de typage. Il peut être créé de la manière suivante :

```
# let initial_typing_env =
    let mk_type (ct1,ct2,ct3) =
      Forall([],
       Fun_type (Pair_type(Const_type ct1, Const_type ct2),Const_type ct3))
    in
      let int_ftype = mk_type(Int_type,Int_type,Int_type)
      and float_ftype = mk_type(Float_type,Float_type,Float_type)
      and int_predtype = mk_type(Int_type,Int_type,Bool_type)
      and float_predtype = mk_type(Float_type,Float_type,Bool_type)
      and alpha = Var_type(ref(Unknown 1))
      and beta = Var_type(ref(Unknown 2))
      in
        ("=",Forall([ 1 ],
           Fun_type (Pair_type (alpha,alpha), Const_type Bool_type))) ::
```

```
        (List.map (function s → (s,int_ftype))
           [ "*"; "+"; "-"; "/" ]) @
        (List.map (function s → (s,float_ftype))
           [ "*."; "+."; "-."; "/." ]) @
        (List.map (function s → (s,int_predtype))
           [ "<"; ">"; "<="; ">=" ]) @
        (List.map (function s → (s,float_predtype))
           [ "<."; ">."; "<=."; ">=." ]) @
      [ "^", mk_type (String_type, String_type, String_type) ] @
      [ ("hd",Forall([ 1 ], Fun_type (List_type alpha, alpha)));
      ("tl",Forall([ 1 ], Fun_type (List_type alpha, List_type alpha)));
      ("null",Forall([ 1 ], Fun_type (alpha,
      Fun_type (List_type alpha, Const_type Bool_type))));
      ("fst",Forall([ 1; 2 ], Fun_type (Pair_type (alpha,beta),alpha)));
      ("snd",Forall([ 1; 2 ], Fun_type (Pair_type (alpha,beta),beta))) ]
      ;;
val initial_typing_env : (string * quantified_type) list =
  [("=", Forall ([1], Fun_type (Pair_type (...), Const_type ...)));
    ("*", Forall ([], Fun_type (Pair_type (...), ...))); ...]
```

3.9 Impression des types et des erreurs

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 crée 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` :

```
# let print_consttype = function
    Int_type →  print_string "int"
  | Float_type → print_string "float"
  | String_type → print_string "string"
  | Bool_type → print_string "bool"
  | Unit_type → print_string "unit";;
val print_consttype : consttype -> unit = <fun>
```

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

```
# let string_of_char c =
    let s = " " in s.[0] <- c; s;;
val string_of_char : char -> string = <fun>
```

```
# let var_name n =
    let rec name_of n =
      let q,r = ((n / 26), (n mod 26)) in
        if q=0 then string_of_char(Char.chr (96+r))
        else (name_of q)^(string_of_char(Char.chr (96+r)))
    in
      "'"^(name_of n);;
val var_name : int -> string = <fun>
```

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.

```
# let print_quantified_type (Forall (gv,t)) =
    let names =
      let rec names_of = function
        (n,[]) → []
      | (n,(v1 :: lv)) → (var_name n) :: (names_of (n+1,lv))
      in (names_of (1,gv))
    in
      let var_names = List.combine (List.rev gv) names in
        let rec print_rec = function
          Var_type {contents=(Instanciated t)} → print_rec t
        | Var_type {contents=(Unknown n)} →
            let name =
              ( try List.assoc n var_names
                with Not_found →
                  raise (Failure "Non quantified variable in type"))
            in print_string name
        | Const_type ct → print_consttype ct
        | Pair_type(t1,t2) → print_string "("; print_rec t1;
            print_string " * ";  print_rec t2;  print_string ")"
        | List_type t →
            print_string "(("; print_rec t; print_string ") list)"
        | Fun_type(t1,t2) → print_string "("; print_rec t1;
            print_string " -> "; print_rec t2; print_string ")"
    in
     print_rec t;;
val print_quantified_type : quantified_type -> unit = <fun>
```

L'impression d'un type simple utilise l'impression des chémas de type en créant

pour l'occasion une quantification sur toutes les variables libres du type.

```
# let print_type t =
    print_quantified_type (Forall(free_vars_of_type ([],t),t));;
val print_type : ml_type -> unit = <fun>
```

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.

```
# let typing_handler typing_fun env expr =
    reset_unknowns();
    try typing_fun env expr
    with
      Type_error (Clash(lt1,lt2)) →
        print_string "Type clash between "; print_type lt1;
        print_string " and "; print_type lt2;  print_newline();
        failwith "type_check"
    | Type_error (Unbound_var s) →
        print_string "Unbound variable ";
        print_string s; print_newline();
        failwith "type_check";;
val typing_handler : ('a -> 'b -> 'c) -> 'a -> 'b -> 'c = <fun>
```

3.10 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`.

```
# let type_check e =
    let t =
      typing_handler type_expr initial_typing_env e in
      let qt =
        snd(List.hd(generalize_types initial_typing_env ["it",t])) in
          print_string "it : "; print_quantified_type qt; print_newline();;
val type_check : ml_expr -> unit = <fun>
```

3.11 Exemples

```
# type_check (Const (Int 3));;
it : int
- : unit = ()
# type_check (Const (Float 3.2));;
it : float
- : unit = ()
# type_check (Abs ("x", Var "x"));;
it : ('a -> 'a)
- : unit = ()
# type_check (Abs ("x", Pair(Var "x", Var "x")));;
it : ('a -> ('a * 'a))
- : unit = ()
# type_check (Letin ("f", Abs ("x", Pair(Var "x", Var "x")),
                    ( Pair ((App (Var  "f", Const (Int 3))),
                            (App (Var "f", Const (Float 3.14)))))));;
it : ((int * int) * (float * float))
- : unit = ()
# type_check (App (Var "*", Pair (Const (Int 3), Const (Float 3.1))));;
Type clash between int and float
Exception: Failure "type_check".
# type_check (Cond (
       App (Var "=", Pair(Const(Int 0), Const (Int 0))),
       Const(Int 2),
       Const(Int 5)));;
it : int
- : unit = ()
# type_check (
    Letrecin ("fact",
      Abs ("x",
           Cond (App (Var "=",Pair(Var "x",Const(Int 1))),
                 Const(Int 1),
                 App (Var "*",
                      Pair(Var "x",
                           App (Var "fact",
                                App (Var "-",Pair(Var "x",Const(Int 1) ))
                               )
                          )
                     )
                )
          ),
      App (Var "fact", Const (Int 4))));;
```

```
it : int
- : unit = ()
```

3.12 Exercices

1. Écrire la fonction `parse_ml_expr` qui lit dans un fichier des expressions `ml_expr` et construit leur arbre de syntaxe correspondant.
2. Ajouter les déclarations globales dans le langage mini-ML et modifier le vérificateur de types en conséquence.
3. Comment typer les exceptions, le filtrage de motif et les références ?

3.13 Bibliographie

- R. Milner. "A theory of type polymorphism in programming". J. Comput. Syst. Sci. 1978.
- Jean-Jacques Lévy "Langages de Programmation"
  Notes de cours - M2 (X)
  `http://www.enseignement.polytechnique.fr/informatique/M2/lp/`
  chapitres 3 et 4.

Chapter 4

Typage des traits impératifs

4.1 Remarques sur le typage d'Objective Caml

Le typeur de mini-ML ne tenait pas compte des valeurs mutables (physiquement modifiables). Objective Caml est un langage fonctionnel muni d'effets de bord. Or ceci si le mécanisme de généralisation (des types en schémas de types) variables de types

4.1.1 Polymorphisme et valeurs mutables

Le *typeur* de Objective Caml utilise des variables de types faibles pour rendre sûr la manipulation de valeurs mutables. En effet 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 :

```
let x = ref (fun x -> x)
in
  x:=(fun x -> x + 1);
  !x true;;
```

et entraîne une erreur de typage.

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).

4.1.2 Expressions non-expansives

Voici le texte de la fonction de reconnaissance d'expression non expansives du compilateur Objective Caml 3.0 :

```
let rec is_nonexpansive exp =
  match exp.exp_desc with
    Texp_ident(_,_) -> true
  | Texp_constant _ -> true
  | Texp_let(rec_flag, pat_exp_list, body) ->
      List.for_all (fun (pat, exp) -> is_nonexpansive exp) pat_exp_list &
      is_nonexpansive body
  | Texp_apply(e, None::el) ->
      is_nonexpansive e &&
      List.for_all (function None -> true | Some exp -> is_nonexpansive e) el
  | Texp_function _ -> true
  | Texp_tuple el ->
      List.for_all is_nonexpansive el
  | Texp_construct(_, el) ->
      List.for_all is_nonexpansive el
  | Texp_variant(_, Some e) -> is_nonexpansive e
  | Texp_variant(_, None) -> true
  | Texp_record(lbl_exp_list, opt_init_exp) ->
      List.for_all
        (fun (lbl, exp) -> lbl.lbl_mut = Immutable & is_nonexpansive exp)
        lbl_exp_list &&
      (match opt_init_exp with None -> true | Some e -> is_nonexpansive e)
  | Texp_field(exp, lbl) -> is_nonexpansive exp
  | Texp_array [] -> true
  | Texp_new (_, cl_decl) when Ctype.class_type_arity cl_decl.cty_type > 0 ->
      true
  | _ -> 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, 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'estm 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.

Toutes les autres expressions sont expansives.

En Objective Caml 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).

4.1.3 Exemples

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

avec effets de bord

Objective Caml 3.0	commentaires
expression non expansive	
`let nref x = ref x;;` `nref : 'a -> 'a ref = <fun>`	abstraction schéma de type
expression expansive	

`let x = ref [];;`
`x : '_a list ref = ref []`
`x:=[3];;`
`x;;`
`int list ref`
	ref est un constructeur mutable
	variable de type faible `_a`
	affinage du type de `x`

sans effets de bord

Objective Caml 3.0	commentaires

`let a x y = x y;;`
`('a -> 'b) -> 'a -> 'b`
`let g = a id;;`
`'_a -> '_a`
`g 3;;`
`int = 3`
`g;;`
`int -> int`
	le combinateur A
	son schéma de type
	une application partielle
	son type non généralisé
	affinage du type de `g`

4.1.4 Exercices

1. Introduire les références dans mini-ML.
2. Modifier le typeur de mini-ML pour tenir compte des expressions expansives.

4.1.5 Bibliographie

La thèse de Xavier Leroy décrit le typage polymorphe pour les effets de bords, les canaux de communications et les continuations dans un langage algorithmique fonctionnel.

La thèse d'Emmanuel Engel compare différents systèmes de typage polymorphe pour les effets de bords en présence de modules poru le langage Caml.