

Examen Partiel 27 octobre 2010

Les questions sont largement indépendantes. L'énoncé est composé de 8 pages. L'examen dure 3 heures. Les notes de cours et de TD manuscrites et les supports de cours distribués cette année sont les seuls documents autorisés.

Consigne Lorsqu'il est demandé de décrire une méthode, vous n'êtes pas obligés de donner du code dans tous ses détails (vous pouvez utiliser des fonctions auxiliaires dont l'implantation est directe en vous contentant de spécifier leur comportement). Vous devez par contre décrire les données manipulées et les algorithmes utilisés de manière précise.

Le langage considéré: HA

Dans tout l'énoncé, on s'intéresse à un langage de programmation HA qui s'inspire du langage Haskell.

Conventions lexicales

Identifiants Les identifiants sont formés de caractères alphabétiques minuscules ou majuscules, de caractères numériques, ils commencent par une minuscule.

Mots-clés Les mots-clés (mots réservés) sont **if**, **then**, **else**, **let**, **in**

Caractères spéciaux Les caractères spéciaux sont: () , et ; (parenthèse gauche, droite, virgule et point-virgule).

Symboles infixes Les symboles infixes représentent des fonctions binaires op pour lesquelles on utilise la syntaxe $e_1 \text{ op } e_2$ pour représenter l'application de op aux expressions e_1 et e_2 . Dans HA, les symboles infixes sont formés d'une suite de caractères appartenant à l'ensemble: $\{!, \&, +, *, /, <, =, >, |, -, : \}$. Parmi ces suites de caractères on distingue les *symboles réservés* qui sont: $:$, $:$, $=$, $->$ et les suites d'au moins deux symboles $-$ qui servent pour les commentaires.

Commentaires Les commentaires sont de deux sortes:

- les *commentaires simples* commencent par une suite $--$ de deux ou plus symboles $-$ et finissent au premier retour à la ligne.
- les *commentaires imbriqués* sont délimités par les suites de symboles $\{-$ et $- \}$.

Grammaire

Types Les types de ce langage sont soit des types de base: **int** ou **bool**, soit le type (noté $\tau_1 \rightarrow \tau_2$) des fonctions qui prennent un argument de type τ_1 et renvoient un résultat de type τ_2 , soit le type (noté $(\tau_1, \dots, \tau_n)$) des n-uplets $(v_1, \dots, v_n)$ où chaque v_i a pour type τ_i .

Ainsi une fonction qui prend en argument un booléen et un couple d'entiers et calcule un entier aura pour type: `bool -> (int, int) -> int`.

La flèche $\rightarrow$ associe à droite et des parenthèses peuvent être utilisées pour lever les ambiguïtés: ainsi une fonction de type `(int -> int) -> int` prend en argument une fonction de type `int -> int` et calcule un entier.

Expressions simples Une expression simple est soit une constante entière, soit une constante booléenne (`true` ou `false`), soit un identifiant, soit un symbole infixé utilisé de manière préfixe en le parenthésant: (*infix*), soit une expression parenthésée (*e*), soit un n-uplet formé d'expressions séparées par des virgules: $(e_1, \dots, e_n)$.

Expressions générales les expressions générales sont soit des expressions simples, soit l'application notée $e_1 e_2$ d'une expression e_1 à une expression simple e_2 , soit une expression infixé $e_1 \text{ op } e_2$ avec op une suite de symboles représentant un symbole infixé, soit une déclaration locale de la forme **let** *ident* = e_1 **in** e_2 , soit une expression conditionnelle **if** e_1 **then** e_2 **else** e_3 .

Déclarations et Programme Les déclarations peuvent être de trois formes:

- déclaration du type d'une fonction: *ident* : *typ*
- définition de valeur de la forme: *ident* = *expr*
- définition de fonction de la forme: *ident* $x_1 \dots x_n$ = *expr*

Les déclarations sont séparées par des points virgules. Un programme est une suite de déclarations.

Un exemple de programme (bien formé) est:

```

x = 2+3;
p = f 2 4;
f :: int -> int -> (int,int,int);
f x y = (1,x+y,x-y);
g :: ((int,int,int) -> bool) -> bool;
g h = h p || h (0,1,0)

```

Arbres de syntaxe abstraite

Dans la suite, pour représenter les arbres de syntaxe abstraite des programmes on introduit les types Ocaml suivants:

```

type cst = Int of int | Bool of bool
type typ = TInt | TBool | Prod of typ list | Arr of typ * typ

```

qui représentent respectivement les arbres de syntaxe abstraite pour les constantes (entières ou booléennes) et les expressions de types. Le type des n-uplets $(\tau_1, \dots, \tau_n)$ est représenté en Ocaml par `Prod([ $\tau_1; \dots; \tau_n$ ])` et le type des fonctions $\tau_1 \rightarrow \tau_2$ est représenté par l'expression Ocaml `Arr( $\tau_1, \tau_2$ )`.

1 Analyse lexicale (6 points)

On se donne le fichier `ocamllex` (incomplet) suivant.

```

{
  let kwd_tbl =
    [ "int",INT; "bool",BOOL; "true",CST(Bool true); "false",CST(Bool false);
      "let",LET; "in",IN; "if",IF; "then",THEN; "else",ELSE ]
  let id_or_kwd s = try List.assoc s kwd_tbl with _ -> IDENT s
}
let low = ['a'-'z']
let letter = ['a'-'z' 'A'-'Z']
let digit = ['0'-'9']
let integer = ['0'-'9']+
let space = [' ' '\t' '\n']
let symb = ['!' '&' '*' '+' '/' '-' ':' '>' '<' '=' '|']
let ident = low (letter | digit)*
let dashes = "---"(' - ')*

rule token = parse
  | space+           { token lexbuf }
  | ident as id      { id_or_kwd id }
  | integer as s     { CST (Int(int_of_string s)) }
  | '('              { LP }
  | ')'              { RP }
  | ','              { VIRG }
  | ';'              { PV }
  | eof              { EOF }
  | _ as c           { raise (Lexing_error c) }

```

1. Compléter les règles d'analyse lexicale pour traiter les commentaires simples et les commentaires imbriqués.

Correction :

```

rule token = parse
  ...
  | dashes           { simplecomment lexbuf; token lexbuf }
  | "{-"             { comment lexbuf; token lexbuf }

and comment = parse
  | "{-"             { comment lexbuf; comment lexbuf }
  | "-}"             { () }
  | _                { comment lexbuf }

and simplecomment = parse

```

```

| '\n'          { }
| _             { simplecomment lexbuf }

```

2. Le langage HA autorise la définition de fonctions binaires qui seront notées de manière infixe. Si op est une telle fonction, et e_1 et e_2 sont deux expressions, on écrira $e_1 \text{ op } e_2$ pour représenter l'expression $(\text{op } e_1 e_2)$. D'un point de vue lexical, op est une suite de *symboles ordinaires* (voir liste ci-dessus). Les symboles arithmétiques et logiques comme $+$, $*$, $>=$, $||$ sont des cas particuliers de cette convention.

- (a) Ajouter à l'analyseur lexical, une règle permettant de reconnaître de tels opérateurs.

Le lexème associé sera noté `INFIX` et sa valeur sera la chaîne de caractères correspondant à l'opérateur. Ainsi le symbole d'addition $+$ devra produire le lexème `INFIX("+")` tandis que le symbole logique de disjonction $||$ produira le lexème `INFIX("||")`.

Correction :

```

rule token = parse
| symb+ as s      { INFIX(s) }

```

- (b) Certaines suites de symboles jouent un rôle particulier dans la grammaire, on leur associe des lexèmes spécifiques. Pour cela, on ajoute dans l'analyseur lexical (avant la règle générique pour les symboles infixes) les règles suivantes:

```

| " :: "          { DDP }
| '='             { DEF }
| '! '           { NEG }
| "->"           { ARR }
| '- '           { MINUS }

```

Donner la suite de lexèmes produite dans l'analyse de l'entrée suivante (justifier brièvement le résultat):

```

3 + x -- foo
3 || x - - foo
3 ||+ x --- foo
3 -> x --+ foo

```

Correction :

- `CTE(Int(3)) INFIX("+") IDENT("x")` le reste est un commentaire et ne produit pas de token
- `CTE(Int(3)) INFIX("||") IDENT("x") MINUS MINUS IDENT("foo")`
- `CTE(Int(3)) INFIX("||+") IDENT("x")` le reste est un commentaire. La règle du lexème de taille maximale d'applique: on reconnaît `INFIX("||+")` et non pas `INFIX("||") INFIX("+")`
- `CTE(Int(3)) ARR IDENT("x") INFIX("--+") IDENT("foo")` la règle du lexème de taille maximale d'applique: on reconnaît `INFIX("--+")` au lieu d'un commentaire commençant par `--`

2 Analyse Syntaxique (6 points)

Le prélude du fichier `ocaml yacc` pour la syntaxe de HA déclare les tokens suivants:

```

%token <cst> CST
%token <string> IDENT
%token <string> INFIX
%token NEG MINUS
%token LET IN IF THEN ELSE
%token LP RP VIRG PV DDP ARR DEF
%token INT BOOL
%token EOF

```

1. (a) Donner les règles de grammaires pour reconnaître les types du langage HA tels qu'ils sont décrits en préambule. On construira dans l'action l'arbre de syntaxe abstraite correspondant.

Correction : Une expression de type parenthésée (τ) est interprétée comme le type τ , les types produits ont au moins deux types séparés par une virgule.

```

typ :
| INT                { TInt }
| BOOL               { TBool }
| LP typ RP          { $2 }
| LP ptyp2 RP        { Prod(List.rev $2) }
| typ ARR typ        { Arr($1, $3) }

```

;

```
ptyp2 :  
| typ    VIRG typ    { [$3; $1] }  
| ptyp2 VIRG typ    { $3 :: $1 }
```

- (b) Votre grammaire est-elle ambiguë ? si oui indiquer comment régler la précedence des tokens pour obtenir le comportement attendu.

Correction : La flèche est ambiguë : $int \rightarrow int \rightarrow int$ s'interprète comme $int \rightarrow (int \rightarrow int)$ ou $(int \rightarrow int) \rightarrow int$. L'énoncé demande une associativité droite. Il suffit donc de déclarer:

%right ARR

2. La grammaire (partielle) des expressions comporte les déclarations suivantes (les symboles infixes ont tous la même priorité et associent à gauche):

```
%left INFIX  
%%  
expr :  
| aexpr                { $1 }  
| expr aexpr           { App($1, $2) }  
| NEG aexpr            { App(Var "neg", $2) }  
| MINUS aexpr          { App(Var "uminus", $2) }  
| expr INFIX expr      { App(App(Var $2, $1), $3) }  
| LET IDENT DEF expr IN expr { Letin ($2, $4, $6) }  
| IF expr THEN expr ELSE expr { If($2, $4, $6) }  
  
aexpr :  
| CST                  { Cst($1) }  
| IDENT                { Var($1) }  
| LP expr RP           { $2 }  
| LP INFIX RP          { Var($2) }  
;
```

Un des états de cette grammaire indique les conflits suivants:

```
35: shift/reduce conflict (shift 16, reduce 14) on CST  
35: shift/reduce conflict (shift 17, reduce 14) on IDENT  
35: shift/reduce conflict (shift 21, reduce 14) on LP  
state 35  
expr : expr . aexpr (11)  
expr : expr . INFIX expr (14)  
expr : expr INFIX expr . (14)
```

- (a) Donner un exemple d'une entrée pour laquelle cette situation de conflit se produit.

Correction : Le conflit se présente entre une réduction de la règle $expr ::= expr \text{ INFIX } expr$ et la lecture de *CST*, *IDENT* ou *LP* qui sont le premier terminal possible pour une expression simple *aexpr*. Par exemple : $x + y \ 3$ ou $x + y \ z$ ou $x + y \ (3)$

- (b) Indiquer comment ordonner les précédences pour que l'application (*f e*) soit prioritaire par rapport aux opérateurs infixes.

Correction : Il faut lire le prochain token pour donner la priorité à l'application. Il faut donc déclarer les token *CST*, *IDENT* et *LP* plus prioritaires que *INFIX*:

```
%left INFIX  
%nonassoc CST IDENT LP
```

Le fait que la règle de grammaire impose que l'argument de l'application soit une expression simple évite les conflits de la forme $f \ 3 + 5$. Il n'est donc pas utile de forcer la priorité de la règle d'application.

- (c) Expliquer pourquoi ocaml yacc n'indique pas dans cet état de conflit pour la lecture du token *INFIX*.

Correction : Le terminal *INFIX* est déclaré associatif gauche donc la situation $expr \text{ INFIX } expr \text{ INFIX } expr$ n'est pas un conflit. Dans l'état donné, si le prochain token de l'entrée est *INFIX* alors on procède à une réduction.

3. En pratique tous les symboles infixes n'ont pas la même priorité. Les symboles prédéfinis comme $+$, $>=$, ... ont des priorités relatives fixées que l'on numérote de 1 à 9, la priorité 9 étant la plus forte. L'utilisateur peut fixer la priorité d'un nouvel opérateur qu'il introduit par une déclaration spéciale, de la forme **infixl** n op avec n un entier de 1 à 9. En l'absence de déclaration explicite la priorité d'un nouvel opérateur est fixée par défaut à 9. En pratique, on introduit dans la grammaire neuf tokens différents `INFIX1`, ..., `INFIX9` correspondant aux différents niveaux de priorité. On ne considère ici que des opérateurs qui associent à gauche. Lorsqu'on rencontre une suite de symboles correspondant à un opérateur infix, il faut produire le lexème `INFIX $i$` qui convient.

Que faut-il modifier dans l'analyseur lexical et l'analyseur syntaxique pour prendre en compte ces priorités?

Dans la phase d'analyse lexicale, on pourra penser à enregistrer dans une table les priorités déclarées à l'aide du mot clé **infixl**. Les opérateurs infixes pourront ensuite être traités en s'inspirant du traitement des mot-clés.

Correction : Dans l'analyseur lexical, on introduit une table pour stocker les priorités déclarées à l'aide de **infixl**. On associe à chaque symbole le token associé. La fonction `add_infix` ajoute le symbole s avec la priorité n . On peut préremplir la table avec les fonctions prédéfinies.

```
let tinfix = Hashtbl.create 14
```

```
let mkinfix n s = match n with
  1 -> INFIX1(s) | 2 -> INFIX2(s) | 3 -> INFIX3(s) | 4 -> INFIX4(s)
  | 5 -> INFIX5(s) | 6 -> INFIX6(s) | 7 -> INFIX7(s) | 8 -> INFIX8(s)
  | 9 -> INFIX9(s) | _ -> failwith "incorrect_level"
```

```
let add_infix s n = Hashtbl.add tinfix s (mkinfix n s)
```

Une fonction `infix` prend en argument un symbole et renvoie soit le token associé dans la table soit `INFIX9`.

```
let infix s = try Hashtbl.find tinfix s with Not_found -> INFIX9(s)
```

On transforme l'analyseur lexical pour interpréter les commandes **infixl** et reconnaître les symboles dans la bonne catégorie.

```
rule token = parse
  | "infixl" space+ (digit as n) space+ (symb+ as s)
    { add_infix s (int_of_char n - 48);
      token lexbuf }
  | symb+ as s
    { infix s }
```

Dans l'analyseur lexical, on introduit les neuf tokens `INFIX $i$` avec les bonnes priorités et on duplique les règles contenant `INFIX`. On ne peut pas mettre l'indice en valeur du token `INFIX` car la grammaire ne permettrait pas alors de distinguer les priorités.

```
%left INFIX1
%left INFIX2
%left INFIX3
...
expr:
...
| expr INFIX1 expr           { App (App(Var $2, $1), $3) }
| expr INFIX2 expr           { App (App(Var $2, $1), $3) }
| expr INFIX3 expr           { App (App(Var $2, $1), $3) }
...
aexpr:
...
| LP INFIX1 RP               { Var $2 }
| LP INFIX2 RP               { Var $2 }
| LP INFIX3 RP               { Var $2 }
```

3 Analyse sémantique (6 points)

Les règles de portées autorisent l'ensemble des fonctions à être définies de manière mutuellement récursives. On demande que pour chaque fonction du programme définie par:

$$f\ x_1 \dots x_n = e$$

le programme contienne également une déclaration du type de cette fonction de la forme:

$f :: \tau$

Par ailleurs un certain nombre de fonctions sont prédéfinies, en particulier les opérateurs arithmétiques, booléens et les fonctions de comparaison. On suppose qu'une table de visibilité `prim` contient toutes les fonctions prédéfinies avec pour chacune son type. Par exemple `+` a pour type `int -> int -> int`, `<=` a pour type `int -> int -> bool`, `||` a pour type `bool -> bool` et `uminus` a pour type `int -> int`.

Contrairement aux fonctions, les définitions de variables:

$x = e$

ne sont pas forcément associées à des déclarations et ne peuvent être utilisés qu'après leur définition.

1. Donner un type Ocaml pour représenter les arbres de syntaxe abstraite pour les expressions (on complètera le type utilisé dans la grammaire précédente pour prendre en compte les n-uplets), les déclarations et les programmes.

Correction : La distinction entre expressions et expressions simples ne sert que pour éviter les conflits dans l'analyse syntaxique. De même dans ce langage, les opérateurs infixes sont considérés comme des fonctions ordinaires, il n'est donc pas nécessaire d'avoir un opérateur spécifique.

```
type program = decl list
and decl =
  Fdecl of string * typ
  | Fdef of string * (string list) * expr
and cst = | Int of int | Bool of bool
and expr =
  Cst of cst
  | Var of string
  | Tuple of expr list
  | App of expr * expr
  | Letin of string * expr * expr
  | If of expr * expr * expr
```

2. Préciser les fonctions dont vous aurez besoin pour manipuler la table de visibilité.

Correction : Il faut associer à chaque identificateur (fonction ou variable ordinaire) son type.

```
module Key = struct type t = string let compare = compare end
module Env = Map.Make(Key)
Env.add : ident -> typ -> Env.t -> Env.t
Env.mem : ident -> Env.t -> bool
Env.find : ident -> Env.t -> typ
```

3. Ecrire une fonction qui étant donné un programme p , complète la table de visibilité des opérations primitives `prim` avec toutes les fonctions déclarées.

Correction : On collecte dans le programme toutes les déclarations de type $\varepsilon :: \tau$ et on les ajoute à la table des opérations primitives.

```
let add_decl vis = function Fdecl(id, t) -> Env.add id t vis | _ -> vis
let add_decls p = List.fold_left add_decl prim p
```

4. Ecrire une fonction qui étant donnés une table de visibilité comportant les objets visibles (fonctions et variables), et une expression e , vérifie que l'expression e est bien typée et calcule son type.

Correction :

```
let rec type_expr vis e = match e with
| Cst(Int n) -> TInt
| Cst(Bool _) -> TBool
| Var x -> if Env.mem x vis then Env.find x vis
  else raise (TypeError ("ident_not_declared"))
| Tuple(le) -> let lt = List.map (type_expr vis) le in
  Prod(lt)
| Letin(x, e1, e2) ->
  let taul = type_expr vis e1 in
  let vis' = Env.add x taul vis in
  type_expr vis' e2
```

```

| If(e1, e2, e3) ->
    let tau1 = type_expr vis e1 in
    if tau1 <> TBool then
        raise (TypeError ("Expecting_condition_of_type_bool"))
    else let tau2 = type_expr vis e2 and tau3 = type_expr vis e3 in
    if tau2 <> tau3 then
        raise (TypeError ("Expecting_branches_of_same_type"))
    else tau2
| App(e1, e2) ->
    let t1 = type_expr vis e1 in
    match t1 with
    | Arr(t2, t) -> if type_expr vis e2 = t2 then t
    | _ -> raise (TypeError ("type_of_argument_does_not_correspond_to_expected_type"))
| _ -> raise (TypeError ("not_a_function"))

```

5. Ecrire une fonction qui vérifie qu'une déclaration de fonction est bien formée et que son type correspond bien à la déclaration.

Correction : Il faut utiliser le type déclaré dans la table pour inférer les types des arguments et du résultat.

```

let rec check_fun vis lx e tau =
    match (lx, tau) with
    | ([], _) -> let tau' = type_expr vis e in
    if tau' <> tau
    then raise (TypeError ("type_of_body_does_not_correspond_to_expected_type"))
    | (x::l, Arr(tau1, tau2)) ->
        check_fun (Env.add x tau1 vis) l e tau2
    | _ -> raise (TypeError ("type_of_function_does_not_match_declaration"))

```

6. En déduire une fonction qui vérifie qu'un programme est bien formé.

```

let rec check_decls vis = function
    [] -> ()
  | Fdecl(_, _)::p -> check_decls vis p (* type declarations are already in vis *)
  | Fdef(id, [], e)::p ->
    let tau = type_expr vis e in
    let vis' = Env.add id tau vis in check_decls vis' p
  | Fdef(id, lx, e)::p -> let tau = Env.find id vis in
    check_fun vis lx e tau; check_decls vis p

let check_program p = let vis = add_decls p in check_decls vis p

```

4 Filtrage (4 points)

On étend le langage HA pour inclure des *motifs* qui permettent d'associer des noms de variables aux composantes d'un n-uplet. Un motif est soit une variable x , soit un n-uplet de motifs $(p_1, \dots, p_n)$ soit le motif universel noté $_$. Le type des arbres de syntaxe abstraite des motifs sera défini par:

```
let pattern = Pvar of string | Puniv | Ptuple of pattern list
```

En pratique on étend la construction **let** $x = e_1$ **in** e_2 au cas **let** $p = e_1$ **in** e_2 où p est un motif.

Par exemple, on pourra implanter l'addition de deux points ou encore l'accès à la première composante d'une paire d'entiers par les fonctions:

```

addp :: (int, int, int) -> (int, int, int) -> (int, int, int);
addp p1 p2 =
    let (x1, y1, z1) = p1 in let (x2, y2, z2) = p2 in
    (x1+x2, y1+y2, z1+z2);
fsti :: (int, int) -> int;
fsti p = let (x, _) = p in x

```

Les motifs et les expressions associées doivent se correspondre. Par exemple si e est une expression qui a pour type $((int, int), int)$, il peut lui correspondre le motif universel $_$, un motif formé d'une seule variable x qui sera de type $((int, int), int)$, un motif formé d'un couple de variables (x_1, x_2) avec x_1 de type (int, int) et x_2 de type int ou un

motif $((x_1, y_1), x_2)$ avec x_1 de type `int`, y_1 de type `int` et x_2 de type `int`. Par contre le motif $(x_1, (x_2, -))$ ne pourra pas être associé à l'expression e car le motif $(x_2, -)$ de la seconde composante ne correspond pas au type attendu `int`.

On définit formellement une relation de bonne formation $(p : \tau) \rightsquigarrow l$ qui étant donnés un motif p et un type τ définit quand le motif p est bien formé par rapport au type τ et correspond à la liste d'associations l entre les variables du motif et leur type.

$$\frac{}{\langle - : \tau \rangle \rightsquigarrow []} \quad \frac{}{\langle x : \tau \rangle \rightsquigarrow [(x, \tau)]} \quad \frac{\langle p_1 : \tau_1 \rangle \rightsquigarrow l_1 \dots \langle p_n : \tau_n \rangle \rightsquigarrow l_n}{\langle (p_1, \dots, p_n) : (\tau_1, \dots, \tau_n) \rangle \rightsquigarrow l_1 @ \dots @ l_n}$$

1. Dire si les motifs et les types suivants sont bien formés et si oui construire l'association des variables et des types correspondante.

(a) motif: $(x, -, (y, z))$ type: $(\text{int} \rightarrow \text{int}, (\text{bool}, \text{bool}), (\text{int}, \text{bool}))$

(b) motif: (x, y) type: $(\text{int} \rightarrow \text{int}, \text{bool}, \text{int})$.

Correction : Le premier motif est correct et construit les liaisons $(x:\text{int} \rightarrow \text{int}) (y:\text{int}) (z:\text{bool})$ le second est incorrect: le motif est un n-uplet de taille 2 et le type un produit de taille 3.

2. Ecrire une fonction qui étant donnés un motif p et un type τ vérifie que le motif correspond au type et calcule la liste d'associations l telle que $\langle p : \tau \rangle \rightsquigarrow l$ et échoue sinon.

Correction :

```
let rec check_pattern p tau = match (p, tau) with
| Pvar (x), _ -> [(x, tau)]
| Pall, _ -> []
| Ptuple(lp), Prod(lt) ->
  (try (List.fold_left2 (fun v p t -> v @ check_pattern p t) [] lp lt)
   with Invalid_argument _ -> raise (TypeError ("incorrect_pattern"))
  )
| _ -> raise (TypeError ("incorrect_pattern"))
```

3. La règle de typage des expressions **let in** avec motif peut alors s'écrire:

$$\frac{\Gamma \vdash e_1 : \tau_1 \quad (p : \tau_1) \rightsquigarrow l \quad \Gamma + l \vdash e_2 : \tau_2}{\Gamma \vdash \text{let } p = e_1 \text{ in } e_2 : \tau_2}$$

En déduire le cas correspondant de la définition de la fonction de typage d'une expression:

```
let rec type_expr vis = function
...
LetIn(p, e1, e2) ->
...
```

Correction : On introduit une fonction `add_pat` qui ajoute dans la table de visibilité les liaisons de variables associées au motif. Il suffit ensuite d'utiliser cette fonction à la place de `Env.add`.

```
let rec add_pat vis p tau =
  List.fold_left (fun v (x, t) -> Env.add x t v) vis (check_pattern p tau)

let rec type_expr vis = function
...
| LetIn(pat, e1, e2) ->
  let taul = type_expr vis e1 in
  let vis' = add_pat vis pat taul in
  type_expr vis' e2
```