

Cours de Compilation-Exercices

Analyse sémantique

Master d'Informatique M1 2009–2010

5 octobre 2009

1 Arbre de syntaxe abstraite-Portée des identificateurs-Typage

Soit un langage de programmation objet dans lequel un programme est constitué d'une suite de déclarations de classes comportant des variables et des méthodes.

Le but de cet exercice est d'explorer des représentations possibles pour les arbres de syntaxe abstraite de ce langage dans le but d'implanter des fonctions de vérification de la bonne formation des programmes.

Types Les variables, paramètres ou résultat de fonctions ont pour type soit `int` le type des entiers, soit `bool` le type des valeurs booléennes soit une classe définie par l'utilisateur.

Déclarations Une *déclaration de variable* s'écrit : `let id = exp` avec *id* le nom de la variable et *exp* la valeur initiale.

Une *déclaration de méthode* s'écrit : `let id (id1 : type1, ..., idn : typen) : type = exp` avec *id* le nom de la méthode, *id_i* le nom du *i*-ème paramètre, *type_i* le type du paramètre et *type* le type du résultat de la fonction.

Les méthodes dans une classe peuvent être mutuellement récursives.

Expressions Une *expression* peut être :

- une variable, déclarée dans une classe ou bien comme paramètre d'une fonction.
- le champs d'un objet *exp.id*
- une valeur entière ou booléenne (`true` ou `false`).
- le mot clé **this** représentant l'objet lui-même.
- une expression *exp₁ op exp₂* formée d'un opérateur binaire prédéfini **op** appliqué à deux expressions *exp₁* et *exp₂* avec **op** ∈ {<, >, ≤, ≥, +, *, /, -, =}
- un appel de fonction *id(exp₁, ..., exp_n)*
- une expression conditionnelle :
if exp then exp₁ else exp₂.
- une expression avec déclaration locale **let id = exp₁ in exp₂**

Grammaire La grammaire du langage est :

Prog	:= Classes
Classes	:= ϵ Classes Class
Class	:= class <i>id</i> extends <i>id</i> { <i>VDecls</i> <i>MDecls</i> }
VDecls	:= ϵ VDecls let <i>id</i> = <i>Exp</i>
MDecls	:= ϵ MDecls let <i>id</i> (TypedId) : Type = <i>Exp</i>
Type	:= int bool <i>id</i>
TypedId	:= ϵ TypedIdNv
TypedIdNv	:= <i>id</i> : Type <i>id</i> : Type, TypedIdNv
Exp	:= <i>id</i> Exp . <i>id</i> <i>valint</i> <i>valbool</i> this Exp <i>op</i> Exp if Exp then Exp else Exp let <i>id</i> = Exp in Exp <i>id</i> (Exps)
Exps	:= ϵ ExpsNv
ExpsNv	:= Exp Exp, ExpsNv

Portée Les *règles de portée* définissent la durée de vie des identificateurs introduits dans les déclarations. Elles permettent de relier chaque utilisation d'un identificateur apparaissant dans le corps du programme à la déclaration dont il dépend.

Les règles de portée de notre langage sont les suivantes :

- dans une déclaration globale de valeur **let** *id* = *exp* la portée de *id* est l'ensemble des déclarations qui suivent dans le programme, en particulier *id* n'est pas visible dans *exp*.
- dans une déclaration globale de méthode
let *id* (*id*₁ : *type*₁, ..., *id*_{*n*} : *type*_{*n*}) : *type* = *exp*
la portée des paramètres formels *id*_{*i*} est limitée à l'expression *exp*; la portée de *id* est l'ensemble des déclarations de méthodes dans le programme; *id* est en particulier visible dans *exp*;
- dans une déclaration locale **let** *id* = *exp*₁ **in** *exp*₂, la portée de l'identificateur *id* introduit est limitée à *exp*₂
- un paramètre ou une variable peut être caché par la redéfinition d'un paramètre ou d'une variable de même nom dans un bloc plus profond.

Questions

1. Dire pour chacune des déclarations du programme suivant si elles respectent les règles de portée et si oui à quelle déclaration correspond chaque utilisation d'un identificateur.

```
let x = 1
let y = x+1
let f(y:int,z:int):int= x + y + let y = y+3 in g(z+y)
let g(x:int,y:int):int = f(x-1,y-1)
let h(x) = let z = x+z in f(y)
```

2. Pour représenter les arbres de syntaxe abstraite de ce langage, on introduit les types CAML suivants :

```
type ident = string
type asa_typ = TypInt | TypBool | Typid of ident
type cte = Int of int | Bool of bool
type op = Plus | Mult | Div | Minus | Leq | Geq | Lt | Gt | Eq
```

Compléter cette définition par les définitions des types `asa_class`, `asa_decl`, `asa_meth`, `asa_exp` et `asa_prog` permettant de coder les arbres de syntaxe abstraite des déclarations, des expressions et des programmes.

3. Lorsqu'une erreur de portée ou de typage se produit, il est nécessaire de produire un message d'erreur significatif. Pour cela on souhaite pouvoir associer à un arbre de syntaxe abstraite, la suite de caractères dans le fichier initial correspondante.

Pour construire la nouvelle structure d'arbres localisés en CAML, on peut procéder de la manière suivante :

- Le type CAML `Lexing.position` permet de conserver les informations relatives à la localisation;
- introduire un type polymorphe `'a loc` permettant d'ajouter une localisation à n'importe quelle valeur de type `'a`;
- introduire pour chaque type d'arbre de syntaxe abstraite `asa`, deux types distincts, l'un `asa_raw` qui contient un objet non-localisé (mais dont les sous expressions sont localisées) et un `asa` qui est défini comme `asa_raw loc`

La structure CAML sera donc :

```
(* position du premier et dernier caractere de l'expression *)
type location = Lexing.position * Lexing.position
(* Un type generique pour annoter un asa par une position *)
type 'a loc = {loc : location;asa : 'a}

type asa = asa_raw loc
and asa_raw = C1 of ... | C2 of ...
```

- (a) Que faut-il changer aux définitions de la question précédente pour introduire les localisations?
- (b) On suppose que la grammaire comporte les règles suivantes :

```
decl:
    LET ID EQ expr          {}
expr:
    expr OP expr           {}
```

Compléter les actions pour produire des arbres localisés.

- (c) On introduit une exception pour les erreurs de typage qui prend en argument une chaîne de caractères et une localisation.

```
exception TypeError of string * location
```

On suppose qu'une liste d'identificateurs `primitifs` est donnée qui correspondent à des fonctions primitives qui ne sont pas autorisées à être redéfinies. Écrire une fonction CAML qui prend en argument un programme, vérifie que les déclarations globales respectent la condition et renvoie un message d'erreur approprié si ce n'est pas le cas.

4. On se propose de décrire les règles de typage pour le langage. Pour cela on introduira un environnement qui associe aux noms de variables déclarées un type et aux noms de fonctions un profil. Un profil est de la forme $[\tau_1, \dots, \tau_n] \rightarrow \tau$, avec τ_i le type des arguments et τ le type du résultat. Un environnement s'écrit $id_1 : \pi_1, \dots, id_n : \pi_n$ où les π_i sont soit des types soit des profils. On écrira $f : [int; int] \rightarrow int, x : bool$ un environnement qui définit une fonction f qui attend deux arguments de type entier et renvoie un entier ainsi qu'une variable x . Si d est une déclaration, alors on peut lui associer $id(d)$ qui est l'identifiant introduit par la déclaration et $\pi(d)$ qui est le type ou le profil de $id(d)$. En itérant ces opérations, on peut associer à tout programme p un environnement qui sera noté $\Gamma(p)$.
 - (a) Identifier pour les différentes classes syntaxiques (expressions, déclarations, programme) la forme du jugement qui dit qu'ils sont correctement formés.
 - (b) Donner les règles de typage pour les programmes, les classes et les expressions.
5. Écrire une fonction qui vérifie si un programme satisfait les règles de typage. Pour cela on doit utiliser une table qui comporte tous les identificateurs visibles en un point du

programme avec pour chacun son type ou son profil (types des arguments et du résultat) dans le cas d'une fonction. On se contentera pour cette question de spécifier les opérations nécessaires sur cette structure de données.

6. Proposer plusieurs manières d'implanter la structure de données utilisée dans la question précédente et décrire les avantages et les inconvénients de chaque méthode.

Implanter les différentes solutions proposées en utilisant les bibliothèques CAML suivantes :

- (a) `List` en utilisant des listes d'associations et les fonctions

```
assoc : 'a -> ('a*'b) list -> 'b
mem_assoc : 'a -> ('a*'b) list -> bool
```

- (b) `Hashtbl` qui implante des tables de hachage

```
create : int -> ('a,'b) t
add : ('a,'b) t -> 'a -> 'b -> unit
find : ('a,'b) t -> 'a -> 'b
mem : ('a,'b) t -> 'a -> bool
remove : ('a,'b) t -> 'a -> unit
```

- (c) `Map` qui implante des ensembles d'association sous forme d'arbres AVL. Le module `Map` définit un module qui peut être instancié sur n'importe quel ensemble de clés muni d'une fonction de comparaison. Pour l'utiliser, on peut faire :

```
module Key = struct type t=ident let compare = compare end
module Env = Map.Make(Key)
Env.empty : 'b Env.t
Env.add : ident -> 'b -> 'b Env.t -> 'b Env.t
Env.find : ident -> 'b Env.t -> 'b
Env.mem : ident -> 'b Env.t -> bool
```

7. Afin de relier chaque utilisation de symbole à la déclaration correspondante, on choisit d'associer une table des symboles qui permet de stocker les déclarations et on transforme l'arbre de syntaxe abstraite pour que chaque utilisation soit remplacée par une référence à la table des symboles.

On choisit d'associer un nom unique à chaque déclaration et de remplacer l'utilisation d'un identificateur par le nom de la déclaration correspondante.

- (a) Préciser les informations stockées dans la table des symboles.
- (b) Construire une fonction qui transforme l'arbre issu de l'analyse syntaxique en un arbre avec des identificateurs uniques et qui remplit la table des symboles avec pour chaque déclaration, l'information connue sur les types et profil des déclarations. On supposera donnée une fonction `store` qui étant donné un identificateur et une information à stocker, engendre un nom unique, stocke la déclaration dans la table des symboles et renvoie le nom unique.
- (c) Proposer une implantation de la fonction `store`.

2 Surcharge à la ADA

On suppose qu'en ADA, on a l'opérateur `*` usuel de multiplication sur les entiers dont le profil est `int × int → int` et que l'on surcharge cet opérateur en le redéfinissant avec les profils `int × int → complexe` et `complexe × complexe → complexe`. On suppose que les valeurs `2, 3, 4, ...` ont un seul type, le type `int` et que `z` est une variable de type `complexe`.

1. Appliquer l'algorithme de résolution de surcharge aux expressions suivantes et dire si elles sont correctement typées et non ambiguës : $z * (3 * 5)$, $1 * (z * 2)$, $2 * (3 * 5)$, $z * (2 * (3 * 5))$, $((1 * 2) * (3 * 5)) * z$.
2. A quelle condition sur le type de a , b et c , l'expression $(a * b) * c$ admet-elle un type unique ?

3 Surcharge à la Java

Dans Java, la surcharge est résolue statiquement en ne prenant en compte que le type des arguments de la méthode. À cause du sous-typage, plusieurs méthodes peuvent s'appliquer aux mêmes arguments. L'ambiguïté est levée s'il existe parmi toutes les méthodes applicables exactement une méthode dont le profil est plus petit que tous les autres. Le profil d'une méthode à n arguments est formé de la classe dans laquelle la méthode est déclarée et des types des arguments. Un profil $(C, \tau_1, \dots, \tau_n)$ est plus petit qu'un profil $(D, \sigma_1, \dots, \sigma_p)$ si $n = p$, que C est une sous-classe de D et que chaque τ_i est un sous-type de σ_i .

Parmi les exemples suivants, dire s'ils sont correctement formés et si oui indiquer à quelle déclaration fait référence chaque utilisation de méthode.

1.

```
class A {
    void g(A a) { g(a,a); }
    void g(A a, A b) {g(a); }
    void g(B b, B a) {g(a); }
}
class B extends A {
    void g(B b) { g(b,b); }
    void g() { g(this); }
    void g(int n) { g(new A()); }
}
```
2.

```
class A { }
class B extends A {
    void g() { g(this,this); }
    void g(A a, B b) { }
    void g(B b, A a) { }
}
```
3.

```
class A {
    void f(B b) { }
}
class B extends A {
    void f(A a) { }
}
class Main {
    static void main(A a) { a.f(a); }
}
```

On se propose d'implanter une fonction qui étant donné un profil p et un ensemble de profils lp (représenté par exemple par une liste de profils) cherche s'il existe un seul profil minimal p' dans lp qui est plus grand que p . Ce problème peut se formuler de manière plus générale, on se donne un ordre partiel \sqsubseteq sur un ensemble A , en pratique une fonction booléenne. Étant donné une liste l d'éléments de A , construire une liste l' des éléments minimaux de l , ie $l' \subseteq l$, deux éléments différents de l' sont incomparables et tout élément de l est plus grand qu'un élément dans l' .

1. Ecrire une fonction `minimal` qui effectue cette opération.
2. On suppose donnée une fonction qui compare deux types de Java suivant la relation de sous-typage. Un profil est représenté par une liste de types. Écrire la fonction `resoud_surcharge` qui étant donné un profil d'utilisation d'une méthode m et une liste de profils possibles pour une méthode m , résoud la surcharge lorsque cela peut être fait de manière non ambiguë.
3. Surcharge sur les types de base. Dans Java les types de base suivent la hiérarchie suivante `byte ≤ short ≤ int`, `char ≤ int` et `int ≤ long ≤ float ≤ double`. La conversion d'un type vers un type plus grand est faite implicitement par le compilateur qui introduit si nécessaire des opérations de conversion.

Ecrire une fonction qui étant donné un arbre de syntaxe abstraite $\text{Op}(e_1, \text{Plus}, e_2)$ produit un arbre avec l'opération adéquate (PlusInt , Pluschar , Plusfloat ...) et les conversions associées (Int2Float ...)

4 Analyse de Flots

L'objet de cet exercice est l'étude d'un cas simple d'analyse de flots. Il s'agit de repérer la durée de vie des variables dans un langage intermédiaire d'affectations dans le but de décider de l'allocation des variables dans les registres.

Le langage que nous manipulons contient une suite d'instructions; chaque instruction est éventuellement précédée d'une étiquette. Une expression est soit une variable, soit une constante entière soit une opération arithmétique appliquée à deux variables ou constantes. Une instruction est

- soit l'affectation d'une expression E à une variable x qui s'écrit $x \leftarrow E$,
- soit une conditionnelle qui s'écrit $\text{if } E_1 \phi E_2 \text{ goto } L$ avec E_i une variable ou une constante, ϕ un opérateur relationnel (égalité, inégalité), et L une étiquette qui peut être symbolique ou bien être le numéro d'une instruction,
- soit une valeur de retour qui s'écrit $\text{return } x$ avec x une variable ou une constante.

On désire calculer le graphe d'analyse de flots dont les nœuds sont les instructions et une arête entre un nœud i et un nœud j signifie que l'instruction j peut avoir lieu juste après l'instruction i . Dans notre langage les instructions sont exécutées de manière séquentielle, sauf lors des branchements conditionnels, où, si la condition est vraie, alors l'instruction exécutée ensuite sera l'instruction associée à l'étiquette. L'instruction return marque la fin du programme.

1. Donner un type d'arbre de syntaxe abstraite pour représenter les programmes de ce langage.
2. Définir une fonction qui étant donné un programme, calcule la correspondance entre étiquette de branchement symbolique et numéro d'instruction.
3. Écrire une fonction qui transforme le programme contenant des étiquettes symboliques en une suite d'instructions ne comportant que des étiquettes numériques correspondant aux numéros des instructions.
4. A quel moment peut-on détecter qu'une étiquette est déclarée plus d'une fois ou bien qu'elle n'est pas déclarée ?
5. On suppose dorénavant que les instructions du programme ne comportent que des étiquettes numériques. Écrire un programme qui étant donné une instruction de numéro i calcule la liste des instructions qui peuvent être exécutées après i .
6. Le *graphe de flots* d'un programme a pour sommets les numéros des instructions du programme et une arête de i à j si l'instruction j peut être exécutée juste après i . Soit le programme :

```

a <- 0
L : b <- a+1
c <- c+b
a <- b*2
if a < N goto L
return c

```

Construire le graphe de flots de ce programme.

7. Une instruction d'affectation définit une variable. Cette variable peut être utilisée dans la partie droite d'une affectation, un test conditionnel ou une instruction de retour. Définir une fonction qui associe à chaque instruction l'ensemble des variables définies et l'ensemble des variables utilisées.

8. Une variable est vivante sur une arête du graphe de flots s'il existe un chemin commençant par cette arête qui s'arrête à un nœud qui utilise la variable et ne passe pas par un nœud qui définit cette variable.

Indiquer dans le graphe de flots calculé précédemment les arêtes où les variables a, b et c sont vivantes.

9. À chaque nœud n du graphe, on associe $In(n)$ l'ensemble des variables vivantes sur au moins une arête arrivant vers n et $Out(n)$ l'ensemble des variables vivantes sur au moins une arête issue de n . Donner une équation qui définit $In(n)$ en fonction de $Out(n)$ et des variables définies ou utilisées en n , puis donner une définition de $Out(n)$ en fonction de $In(p)$ pour p les successeurs de n dans le graphe. En déduire une manière systématique de calculer $In(n)$ et $Out(n)$ et l'appliquer au programme précédent.

5 Typage d'un langage avec unités de mesure

D'après sujet de TP de compilation 96-97 (C. Marché).

Soit un langage de programmation appelé UNIT (à la ML mais sans fonction d'ordre supérieur et avec typage explicite) dans lequel un programme est constitué d'une suite de déclarations de fonctions ou de valeurs. La particularité de ce langage est de pouvoir associer une unité de mesure à une valeur numérique et de vérifier que les opérations s'effectuent de manière cohérente par rapport aux dimensions associées aux valeur manipulées. Ainsi on ne pourra pas additionner des mètres et des degrés. Par contre ce langage permet d'additionner des grandeurs de même nature mais représentées dans des unités différentes par exemple des heures avec des secondes ou des années.

Déclarations Une *déclaration de variable* s'écrit : **let** $id = exp$ avec id le nom de la variable et exp la valeur initiale.

Une *déclaration de fonction* s'écrit : **let** $id (id_1 : type_1, \dots, id_n : type_n) : type = exp$ avec id le nom de la fonction, id_i le nom du i -ème paramètre, $type_i$ le type du paramètre et $type$ le type du résultat de la fonction. La déclaration de fonction peut être récursive.

Une *déclaration d'unité* s'écrit : **unit** $id = unité$ elle introduit l'identificateur id comme abréviation de l'unité $unité$ (cf paragraph sur les unités).

Expressions Une *expression* peut être :

- une variable, déclarée dans un **let** ou bien comme paramètre d'une fonction.
- une valeur booléenne (**true** ou **false**).
- une valeur numérique associée à une unité : comme **3 m** ou **5 s**
- une expression exp_1 **op** exp_2 formée d'un opérateur binaire prédéfini **op** appliqué à deux expressions numériques exp_1 et exp_2 avec $op \in \{<, >, \leq, \geq, +, *, /, -, =\}$
- un appel de fonction $id(exp_1, \dots, exp_n)$
- une expression conditionnelle : **if** exp **then** exp_1 **else** exp_2 .
- une expression avec déclaration locale **decl in** exp où $decl$ est une déclaration de variable ou de fonction éventuellement récursive avec la même syntaxe que les déclaration globales.

Unités On introduit les 7 *unités* fondamentales du système ISO : m (mètre), kg (kilogramme), s (seconde) A (ampère), K (kelvin), mol (mole) et cd (candela).

Les unités se composent. Si u, u_1 et u_2 sont des unités, c est une constante numérique et n un entier (positif ou négatif) alors les grandeurs suivantes sont des unités :

- $c u$
- $u_1 \cdot u_2$
- u^n

L'unité u^0 correspond à des constantes sans grandeur comme π . On pourra écrire simplement la valeur numérique n pour représenter $n u^0$.

On peut introduire un nom *id* pour représenter l'unité u par la construction syntaxique :

```
unit id = u
```

On peut par exemple introduire des abréviations pour les unités kilomètre, litre, minutes et Newton :

```
unit km = 1000 m
```

```
unit l = 0.001 m3
```

```
unit mn = 60 s
```

```
unit N = m kg s2
```

Une constante numérique se définit associée à une unité de mesure. On pourra introduire une constante représentant la contenance d'une canette de 33 cl par :

```
let contenance_canette = 0.33 l
```

Dimension Chacune de ces unités mesure une *dimension*, la dimension peut se comprendre comme le *type* associé à une unité de mesure.

Les dimensions élémentaires sont : longueur (L), masse (M), temps (T), intensité électrique (C), température (D), quantité de matière (Q) et intensité lumineuse (I).

Plusieurs unités peuvent correspondre à la même dimension : par exemple le mètre et le kilomètre sont des unités de longueur.

Les dimensions se composent comme les mesures, ainsi si δ_1 et δ_2 sont des dimensions, il en est de même de δ_1^n et de $\delta_1.\delta_2$, comme pour les unités, la dimension δ^0 représente une dimension *nulle* pour les constantes sans grandeur.

Ainsi une vitesse sera de dimension $L^2.T^{-1}$ Les dimensions δ sont construites de la manière suivante :

- Les dimensions de base : L, M, T, C, D, Q, I
- Les dimensions produit : $\delta_1.\delta_2$
- Les dimensions puissance : δ^n avec $n \in \mathbb{Z}$

Les dimensions vérifient les égalités suivantes :

$$\begin{aligned} \delta_1.(\delta_2.\delta_3) &= (\delta_1.\delta_2).\delta_3 \\ \delta_1.\delta_2 &= \delta_2.\delta_1 \\ \delta^n.\delta^m &= \delta^{n+m} \end{aligned}$$

Type Les types de base du langage sont le type `bool` des valeurs booléennes et le type des valeurs numériques `num δ` où δ est une dimension (si c'est la dimension nulle on écrira simplement `num`).

Les opérations arithmétiques et logiques vérifient les règles de typage suivantes :

$$\frac{e_1 : \text{num } \delta \quad e_2 : \text{num } \delta \quad \text{op} \in \{+, -\}}{e_1 \text{ op } e_2 : \text{num } \delta} \quad \frac{e_1 : \text{num } \delta \quad e_2 : \text{num } \delta \quad \text{op} \in \{<, >, =, \leq, \geq\}}{e_1 \text{ op } e_2 : \text{bool}}$$

$$\frac{e_1 : \text{num } \delta_1 \quad e_2 : \text{num } \delta_2}{e_1 * e_2 : \text{num } \delta_1.\delta_2} \quad \frac{e_1 : \text{num } \delta_1 \quad e_2 : \text{num } \delta_2}{e_1 / e_2 : \text{num } \delta_1.(\delta_2)^{-1}}$$

Exemple Le programme suivant est bien typé

```
unit h = 60mn
let dist_marathon = 42.195 km
let record_marathon = 2 h + 8mn
let moyenne_marathon = distance_marathon / record_marathon
```

Le type des objets défini est : `dist_marathon : num L`, `record_marathon : num T`, `moyenne_marathon : num L T-1`.

Par contre la déclaration suivante provoque une erreur de type

```
let x = dist_marathon + record_marathon
```

Questions

1. Le type des arbres de syntaxe abstraite des programmes est le suivant :

```
type num = float
type ident = string
type op = Plus | Mult | Div | Minus | Leq | Geq | Lt | Gt | Eq
```

```
type dim = ....
type typ = ....
type unite = ....
type cte = Cte_bool of bool
          | Cte_num of ...
```

```
type asa_decl =
  Tvar of ident * asa_exp
  | Tfun of ident * typ * (ident*typ) list * asa_exp
  | Tunit of ident * unite
and asa_exp =
  Let of asa_decl * asa_exp
  | Call of ident * asa_exp list
  | Var of ident
  | Cte of cte
  | Bin of op * asa_exp * asa_exp
  | If of asa_exp * asa_exp * asa_exp
type asa_prog = asa_decl list
```

Compléter les définitions de `dim`, `unite`, `typ` et `cte` .

2. Écrire un algorithme de typage pour les programmes, on supposera que l'analyse de portée a déjà été réalisée et que chaque identificateur du programme est unique.
3. On suppose maintenant que l'on n'écrit plus les dimensions dans les types numériques mais que l'on veut laisser le système les inférer automatiquement.

Le programme suivant est bien typé avec $\text{pi} : \text{num}$, $\text{surface} : \text{num } L \rightarrow \text{num } L^2$, $s : \text{num } L^2$

```
let pi = 3.14
let surface (d:num) : num = pi (d * d) / 4
let s = let d = 4 cm in surface d
```

- (a) Les programmes suivants sont-ils bien typés ? Si oui décrire l'ensemble des types possibles.

```
let f (x : num, y : num) : num = 1 + ((x * x) / 2 * y)
let g (x : num, y : num) : num = 1 + x * y
let h (x : num, y : num, z : num) : num = 1 + x * y * z
let j (x : num, y : num) : num = let l = 3 m in x + y + l * x * y
```

- (b) Proposer un algorithme pour vérifier qu'un programme est bien formé et calculer le type des expressions et fonctions définies.

On pourra simplement spécifier les fonctions qui seront utiles telles que l'unification de deux dimensions avec variable (on admettra que dans ce formalisme, s'il existe une substitution qui unifie deux dimensions, alors il en existe une principale) ou l'application d'une substitution à un ensemble de dimensions.

- Proposer une traduction des programmes (bien typés) de ce langage vers des programmes CAML.

6 Modules

(partiel 2008)

L'objet de cet exercice est d'étudier l'analyse de portée et le typage d'un mini-langage avec modules.

Le langage considéré est un mini-langage fonctionnel typé.

- Les types (notés τ) sont soit des types de base (dont `bool`) soit des types fonctionnels $\tau_1 \rightarrow \tau_2$ qui est le type des fonctions qui prennent un argument de type τ_1 et renvoient un résultat de type τ_2 .

On introduit le type `ocaml` noté `asaType` pour représenter les arbres de syntaxe abstraite des types.

```
type asaType = Base of string | Bool
              | Arrow of (asaType * asaType)
```

- Les expressions (notées e) sont soit des constantes (`cte`), soit des identificateurs (`ident`), soit des fonctions simples `fun (ident : τ) \rightarrow e` , soit des objets récursifs `rec (ident : τ). e` , soit des conditionnelles `if e_1 then e_2 else e_3` , soit des applications $e_1 e_2$. Les expressions peuvent être parenthésées pour lever les ambiguïtés.

Si l'environnement Γ associe un type à chaque identificateur et la fonction δ donne le type de chaque constante, les règles de typage des expressions sont :

$$\frac{c \in \text{cte}}{\Gamma \vdash c : \delta(c)} \quad \frac{x \in \text{ident} \quad (x : \tau) \in \Gamma}{\Gamma \vdash x : \tau}$$

$$\frac{\Gamma, x : \tau_1 \vdash e : \tau_2}{\Gamma \vdash (\text{fun } (x : \tau_1) \rightarrow e) : \tau_1 \rightarrow \tau_2} \quad \frac{\Gamma \vdash e : \tau_1 \rightarrow \tau_2 \quad \Gamma \vdash e_1 : \tau_1}{\Gamma \vdash (e e_1) : \tau_2}$$

$$\frac{\Gamma, x : \tau \vdash e : \tau}{\Gamma \vdash (\text{rec } (x : \tau).e) : \tau} \quad \frac{\Gamma \vdash e_1 : \text{bool} \quad \Gamma \vdash e_2 : \tau \quad \Gamma \vdash e_3 : \tau}{\Gamma \vdash \text{if } e_1 \text{ then } e_2 \text{ else } e_3 : \tau}$$

On suppose donnés des types `ident` et `cte` pour représenter les identificateurs et les constantes, et un type fonctionnel `env` pour représenter les environnements avec les fonctions suivantes :

Nom	Type	Commentaire
<code>empty</code>	<code>env</code>	environnement vide
<code>mem</code>	<code>ident \rightarrow env \rightarrow bool</code>	teste si l'identifiant est déclaré dans l'environnement
<code>find</code>	<code>ident \rightarrow env \rightarrow asaType</code>	renvoie le type associé à l'identifiant dans l'environnement
<code>add</code>	<code>ident \rightarrow asaType \rightarrow env \rightarrow env</code>	renvoie un nouvel environnement dans lequel l'identifiant est associé au type passé en argument
<code>type_cte</code>	<code>cte \rightarrow asaType</code>	renvoie le type d'une constante

- Proposer un type `ocaml` (noté `asaExpr`) pour représenter les arbres de syntaxe abstraite des expressions de ce langage.
- Écrire une fonction `ocaml` (notée `type_expr`) qui étant donnés un environnement et une expression vérifie que l'expression est bien formée dans l'environnement et calcule son type.
- On suppose maintenant que dans notre langage, les déclarations sont organisées en modules. Plus précisément une déclaration peut être
 - une déclaration simple de variable : `let x = e`

- une déclaration de module : `module m = struct ldecl end`, avec m un identificateur et $ldecl$ une liste de déclarations (soit des variables introduites par `let` soit d'autres modules). Une déclaration de module a pour effet d'introduire dans l'environnement des noms *qualifiés* (`nom_de_module.nom_de_la_variable`) pour les variables du module. Un nom de module peut lui-même être formé d'un simple identificateur ou bien d'une suite de noms de modules séparés par des points dans le cas de modules imbriqués. Par exemple si on suppose que le langage comporte des types de base `real` et `real2` pour les réels et les paires de réels avec des fonctions de base `pair:real→real→real2` pour constituer une paire, `fst` et `snd` de type `real2→real` pour récupérer la première et la seconde composante et `plus`, `mult`, `mean`, `norm`, `angle` de type `real→real→real` pour additionner, multiplier, calculer la moyenne arithmétique ($\frac{x+y}{2}$), la moyenne géométrique ($\sqrt{x^2 + y^2}$), le rapport angulaire de deux réels, une constante 0 réelle, des fonctions `cos`, `sin` de type `real→real` pour calculer les cosinus, sinus, on peut construire un module pour les complexes :

```

module complexe = struct
  let reel = fun (r:real)→pair r 0
  module cartesien = struct
    let add = fun (c1:real2)→fun (c2:real2)→
      pair (plus (fst c1) (fst c2)) (plus (snd c1) (snd c2))
    let fromPol = fun (p:real2)→
      pair (mult (fst p) (cos (snd p)))
          (mult (fst p) (sin (snd p)))
  end
  module polaire = struct
    let add = fun (p1:real2)→fun (p2:real2)→
      pair (norm (fst p1) (fst p2)) (mean (snd p1) (snd p2))
    let fromCart = fun (c:real2)→
      pair (norm (fst c) (snd c)) (angle (fst c) (snd c))
  end
end

```

La déclaration de ce module introduit dans l'environnement les identificateurs suivants :

- `complexe.reel : real→real2`
- `complexe.cartesien.add : real2→real2→real2`
- `complexe.cartesien.fromPol : real2→real2`
- `complexe.polaire.add : real2→real2→real2`
- `complexe.polaire.fromCart : real2→real2`

Tout identificateur déclaré est visible dans la suite des déclarations avec son nom qualifié. On suppose que le type `ident` utilisé dans les expressions correspondent maintenant à des noms qualifiés c'est-à-dire qui sont formés d'une liste de noms de modules et d'un nom de variable. On introduit le type `simpleid` pour représenter un nom simple (de module ou de variable). Un nom qualifié sera juste une liste (non vide) de noms simples.

```
type ident = simpleid list
```

On décide de la représenter en sens inverse de la notation concrète (par exemple le nom qualifié `complexe.cartesien.add` est représenté par la liste `[add; cartesien; complexe]`).

- Proposer un type `ocaml` pour représenter les arbres de syntaxe abstraite des déclarations.
- Écrire une fonction `type_decl` qui étant donnés un environnement et une liste de déclarations vérifie que ces déclarations sont bien typées et renvoie l'environnement complété par les nouvelles déclarations.

Dans cette question, on suppose que tout objet introduit est utilisé dans la suite avec son nom qualifié, c'est-à-dire qu'on écrira :

```

module M = struct
  let x = 3
  let y = M.x + 1

```

end

On pourra penser à passer en argument de `type_decl` une liste de noms de modules correspondant à la suite de modules dans lesquels les déclarations sont faites.

- (c) Il est parfois possible de ne pas utiliser le nom qualifié d'une variable. Par exemple à l'intérieur d'un module, une déclaration peut être réutilisée avec son nom court, sans mentionner les modules dans lequel elle est définie. Par exemple dans le module `polaire`, on peut utiliser le nom court `reel` au lieu de `complexe.reel`, le nom court `add` au lieu de `complexe.polaire.add` et le nom (partiellement qualifié) `cartesien.add` au lieu de `complexe.cartesien.add` (car `cartesien` est une abréviation de `complexe.cartesien`)

On ajoute également une déclaration `open m` dans laquelle `m` est un nom (éventuellement qualifié) de module. L'effet de cette déclaration est que tout nom `x` (de variable ou de module) déclaré dans `m` peut être utilisé directement comme une abréviation pour le nom qualifié `m.x`. De fait, `m` peut lui-même être une abréviation pour un nom de module qualifié `m'`, la variable `x` sera alors une abréviation pour le nom qualifié `m'.x`.

Pour traiter cette question on introduit une phase d'analyse de portée dans laquelle une suite de déclarations comportant des noms abrégés est transformée en des déclarations comportant des noms qualifiés. Pour cela on utilise une table d'abréviations (de type `abbrev`) qui associe à un identificateur simple visible (nom d'une variable ou d'un module de type `simpleid`) la suite de modules dans lequel il est déclaré (un objet de type `simpleid list`).

Dans le cas de notre exemple, au moment de la déclaration de `complexe.polaire.fromCart`, on aura dans la table d'abréviations :

$$\text{reel} \mapsto [\text{complexe}], \text{cartesien} \mapsto [\text{complexe}], \text{add} \mapsto [\text{polaire}; \text{complexe}]$$

On pourra réutiliser pour manipuler cette table les noms d'opérations `empty`, `mem`, `find`, `add` avec des comportements analogues au cas des environnements.

Questions.

- i. Écrire une fonction `transform_id` qui étant donné un nom de type `ident` (qui peut être partiellement qualifié) et une table d'abréviations renvoie le nom qualifié correspondant.
- ii. Écrire une fonction qui étant donnée une liste de déclarations (comportant également des déclarations `open`) utilisant des noms abrégés, reconstruit les déclarations avec des noms entièrement qualifiés. On suppose donnée la fonction `transform_expr` (de type `abbrev → asaExpr → asaExpr`, construite à partir de `transform_id`) qui étant donnée une table d'abréviations, renvoie l'expression correspondante ne comportant que des noms qualifiés. Afin de traiter les déclarations `open`, on pourra également utiliser une table qui à chaque module déclaré associe la liste des identificateurs définis dans ce module.

7 Inférence de types polymorphes

Nous considérons ici un mini langage fonctionnel `PtiCaml` comportant uniquement les expressions suivantes :

$$\begin{aligned} \langle \text{expr} \rangle & := \langle \text{simple_expr} \rangle \mid \text{function } \langle \text{ident} \rangle \text{ -} \rangle \langle \text{expr} \rangle \mid \langle \text{expr} \rangle \langle \text{op} \rangle \langle \text{expr} \rangle \\ & \quad \mid \langle \text{simple_expr} \rangle \langle \text{simple_expr} \rangle^+ \mid \text{let } \langle \text{ident} \rangle = \langle \text{expr} \rangle \text{ in } \langle \text{expr} \rangle \\ & \quad \mid \text{if } \langle \text{expr} \rangle \text{ then } \langle \text{expr} \rangle \text{ else } \langle \text{expr} \rangle \\ \langle \text{simple_expr} \rangle & := (\langle \text{expr} \rangle) \mid \langle \text{ident} \rangle \mid \langle \text{const} \rangle \\ \langle \text{op} \rangle & := + \mid - \mid +. \mid -. \mid = \\ \langle \text{const} \rangle & ::= \text{true} \mid \text{false} \mid \langle \text{entier} \rangle \mid \langle \text{réel} \rangle \\ \langle \text{type} \rangle & := \text{bool} \mid \text{int} \mid \text{float} \mid \alpha \mid \langle \text{type} \rangle \text{ -} \rangle \langle \text{type} \rangle \end{aligned}$$

1. Définir en OCaml l'arbre de syntaxe abstraite des expressions et des types.

7.1 Typage monomorphe

Dans cette première partie, nous nous limitons au typage sans polymorphisme.

2. Quel est le type de l'expression suivante :

```
let id = fonction x -> x in
let a = id 1 in
id true
```

3. Définir le système de type pour PtiCaml monomorphe.
4. Écrire un typeur pour PtiCaml avec typage monomorphe et dans lequel les paramètres de fonctions sont annotés par leur type (`fonction (<ident> : <type>) -> <expr>`).

Nous voulons maintenant réaliser un typeur avec inférence de type (les arguments des fonctions ne sont plus annotés avec leur type).

5. Donner précisément la dérivation de typage de l'expression suivante :

```
fonction g -> fonction h -> fonction x -> g ((h x) + 1) + h (x + 2)
```

6. On représentera une substitution par une liste d'association. Définir une fonction `subst` qui applique une substitution à un type, une fonction `subst_env` qui applique une substitution à tous les types d'un environnement de typage et une fonction `compose` qui compose deux substitutions.
7. Définir une fonction `unify` qui calcule la substitution qui unifie deux types. `unify t1 t2` retourne une substitution `s` telle que `subst s t1 = subst s t2`.
8. Écrire un typeur pour PtiCaml avec typage monomorphe.

7.2 Typage polymorphe

Nous voulons maintenant introduire du typage polymorphe. Pour cela, l'environnement de typage associe à chaque variable un schéma de type (de la forme $\forall \alpha_1, \dots, \alpha_n. \tau$) et les règles de typage des variables et du `let` deviennent :

$$\frac{\tau \in \text{inst}(\Gamma(x))}{\Gamma \vdash x : \tau} \quad \frac{\Gamma \vdash e_1 : \tau_1 \quad \Gamma, x : \forall \bar{\alpha}. \tau_1 \vdash e_2 : \tau_2 \quad \bar{\alpha} = \text{vars}(\tau_1) \setminus \text{fv}(\Gamma)}{\Gamma \vdash \text{let } x=e_1 \text{ in } e_2 : \tau_2}$$

où `vars` est l'ensemble des variables apparaissant dans un type, `fv` est l'union des ensembles des variables libres des schémas de type présents dans un environnement et où `inst` ($\forall \alpha_1, \dots, \alpha_n. \tau$) est l'ensemble des types obtenus en substituant dans τ les variables α_i par des types quelconques.

9. Définir un type `schema` qui représente un schéma de type.
10. Écrire une fonction `specialize` qui à partir d'un schéma de type calcule un nouveau type où toutes les variables de type quantifiées universellement sont remplacées par des variables fraîches.
11. Écrire une fonction `generalize` qui à partir d'un environnement de typage et d'un type va calculer le schéma de type correspondant.
12. Définir la fonction `subst_sc` qui applique une substitution à un schéma de type.
13. Écrire un typeur pour PtiCaml avec typage polymorphe.

7.3 Typage avec niveaux

Avec le système de type précédent, l'opération de généralisation peut coûter cher. En effet, il faut parcourir tous les types de l'environnement pour déterminer les variables que l'on peut généraliser. Nous allons étudier ici un système de type où il suffit de parcourir le type que l'on souhaite généraliser.

Chaque variable de type est associée au niveau auquel elle a été créée (α^n). Les jugements de typage sont maintenant établis à un niveau courant. Ainsi, ils ont la forme suivante : $\Gamma \vdash_n e : \tau$.

Un type τ est dit de niveau n et noté $\tau \in \tau^n$ s'il ne comporte que des variables créées à un niveau au plus égal à n .

On note $\mathcal{V}^{\geq n}(\tau)$ l'ensemble des variables de type présentes dans τ dont le niveau est supérieur ou égal à n .

Les règles de typage des variables, des fonctions et du **let** sont les suivantes dans ce système :

$$\frac{\tau \in \mathbf{inst}(n, \Gamma(x))}{\Gamma \vdash_n x : \tau} \quad \frac{\Gamma, x : \tau_1 \vdash_n e : \tau_2 \quad \tau_1 \in \tau^n}{\Gamma \vdash_n \mathbf{function} \ x \rightarrow e : \tau_1 \rightarrow \tau_2}$$

$$\frac{\Gamma \vdash_{n+1} e_1 : \tau_1 \quad \Gamma, x : \forall \bar{\alpha}. \tau_1 \vdash_n e_2 : \tau_2 \quad \bar{\alpha} = \mathcal{V}^{\geq n+1}(\tau_1)}{\Gamma \vdash_n \mathbf{let} \ x=e_1 \ \mathbf{in} \ e_2 : \tau_2}$$

où $\mathbf{inst}(n, \forall \alpha_1, \dots, \alpha_n. \tau)$ est l'ensemble des types obtenus en substituant dans τ les variables α_i par des types quelconques de niveau au plus n .

14. Redéfinir l'arbre de syntaxe abstraite des types.
15. Redéfinir la fonction **unify** de telle sorte que lors de l'unification d'une variable α^n avec un type τ , toutes les variables de τ soient mises à un niveau au plus n .
16. Redéfinir les fonctions **specialize** et **generalize**.
17. Écrire un typeur pour PtiCam1 avec typage polymorphe avec niveaux.