Année 2017-2018 — Session 1

Systèmes de Types et Programmation
IT234

Filière : Informatique, 2ème Année

Date de l'examen : 24/05/2018 — Durée de l'examen : 1h30

Sujet de : D. Renault

Documents ☐ autorisés ☒ non autorisés — Calculatrice ☐ autorisée ☒ non autorisée

Cet examen se présente comme un ensemble de questions courtes, demandant des réponses courtes (prenant parfois la forme de code). La syntaxe des langages de programmation utilisés ne sera pas vérifiée à la lettre. Le barème est purement indicatif.

Exercice 1: 1 points
Citer deux limites des systèmes de type, en invoquant à chaque fois un exemple d'application.

✓ Les limites citées dans le cours sont : il existe en général des expressions qui ne sont pas typables (*conservativeness*, par exemple le combinateur de point fixe), et il existe en général des expressions typables mais quand même coincées / stuck (*partial functions*, par exemple les divisions par zéro).

Exercice 2: 1 points
Inférer le type de l'expression suivante : $\lambda x.\lambda y.((x\ y)\ y)$ (aucun arbre de dérivation n'est demandé)

✓ En OCaml, `fun x y → (x y) y` renvoie `('a → 'a → 'b) → 'a → 'b`.

Exercice 3: 3 points
Définir ce qu'est un type abstrait de données. Illustrer cette définition dans le langage de votre choix parmi {C++, Java, OCaml} avec un exemple permettant de gérer des trains arrivant et repartant dans des gares (certains détails d'implémentation peuvent être omis à coups de points de suspension).

✓ Selon la définition du cours, un type abstrait de données ou TAD consiste en la donnée d'une variable de type T, d'un ensemble de types de fonctions agissant sur des valeurs de type T (interface), et d'une implémentation sous la forme d'un type concret S et du code des fonctions associées. Dans l'exemple, les types `train` et `station` sont tous les deux abstraits, et pourtant interagissent l'un avec l'autre à travers leurs interfaces :

```
module T : sig (* Interface *)
  type train
  val departure : train → (int*station)
  val arrival   : train → (int*station)
end = struct   (* Implementation *)
  type train = { tdep : int,     tarr : int,
                 sdep : station, sarr : station }
  (* ... *)
end
```

```
module S : sig (* Interface *)
  type station
  val create     : (station*int) → (station*int) → train
  val handles    : station → train → station
  val departures : station → int → train list
  val arrivals   : station → int → train list
end = struct   (* Implementation *)
  type station = { name : string, trains : train list }
  (* ... *)
end
```

Exercice 4: 3 points
En partant du λ-calcul simplement typé décrit en cours (avec une extension pour gérer les entiers) décrire les modifications du langage nécessaires pour ajouter des *tableaux génériques*, avec accès en lecture et écriture, par indice entier. Les règles d'évaluation ne sont pas demandées. Discuter de la sûreté de vos règles de typage.

✓

Syntax and Types

$\mathsf{t} ::=$	$\ldots$	*terms*
	$[\mathsf{t}_1, \ldots, \mathsf{t}_n]$	*array term* (n may be zero)
	$\mathsf{t}[\mathsf{t}]$	*array access*
	$\mathsf{t}[\mathsf{t}] := \mathsf{t}$	*array update*
$\mathsf{v} ::=$	$\ldots$	*values*
	$[\mathsf{v}_1, \ldots, \mathsf{v}_n]$	*array value* (n may be zero)
$\mathsf{T} ::=$	$\ldots$	*types*
	$\mathsf{Array}[\mathsf{T}]$	*array type*

Typing rules

$$\frac{\forall i, \Gamma \vdash \mathsf{t}_i : \mathsf{T}}{\Gamma \vdash [\mathsf{t}_1, \ldots, \mathsf{t}_n] : \mathsf{Array}[\mathsf{T}]} \qquad \frac{\Gamma \vdash \mathsf{a} : \mathsf{Array}[\mathsf{T}] \qquad \Gamma \vdash \mathsf{i} : \mathsf{Nat}}{\Gamma \vdash \mathsf{a}[\mathsf{i}] : \mathsf{T}}$$

$$\frac{\Gamma \vdash \mathsf{a} : \mathsf{Array}[\mathsf{T}] \qquad \Gamma \vdash \mathsf{i} : \mathsf{Nat} \qquad \Gamma \vdash \mathsf{v} : \mathsf{T}}{\Gamma \vdash \mathsf{a}[\mathsf{i}] := \mathsf{v} : \mathsf{Unit}}$$

Les règles de typage ne vérifient pas que l'indice est correct pour accéder au tableau. La vérification est donc faite de manière dynamique.

Exercice 5: 2 points

Considérons un type $\mathsf{Box}[\mathsf{T}]$ capable de contenir une valeur de type T et de modifier ce contenu. En suivant les principes vus en cours, quelles sont les règles de variance possibles pour le type T ?

✓ Selon le *get-and-put* principle, il est possible à la fois de soutirer et d'insérer des valeurs de type T à l'intérieur d'un $\mathsf{Box}[\mathsf{T}]$. Ce type doit donc nécessairement être invariant en T.
Au delà de la question, ce résultat est vexant parce qu'il est censé s'appliquer à n'importe quelle référence générique, pour laquelle on aimerait une propriété de covariance.

Exercice 6: 2 points

Une bibliothèque propose une implémentation d'une fonction en OCaml de type `f: 'a → 'a`. Une personne désireuse de la réimplémenter en Java propose de l'écrire avec le type `Object f(Object o)`. Les deux types sont-ils équivalents ? Proposer une explication.

✓ La fonction OCaml est de type $\forall \mathsf{T}.\mathsf{T} \to \mathsf{T}$, tandis que la fonction Java est de type $(\exists \mathsf{U} <: \mathsf{Object}, \mathsf{U}) \to (\exists \mathsf{V} <: \mathsf{Object}, \mathsf{V})$. Rien dans le type Java n'assure que les types d'entrée et de sortie sont les mêmes.

Exercice 7: 2 points

Justifier ou réfuter l'assertion suivante : "Inclusion polymorphism is also known as runtime polymorphism. Parametric polymorphism is also known as compile-time polymorphism."

✓ Cette assertion sous-entend que le polymorphisme d'inclusion est un polymorphisme décidé à l'exécution, et l'oppose au polymorphisme paramétrique qui serait décidé à la compilation. C'est faux en toute généralité. L'algorithme de sélection de méthode Java discuté en cours montre qu'une partie du polymorphisme d'inclusion au moins peut être décidée à la compilation, et le langage C++ propose des redéfinitions de méthodes sans `virtual` qui sont décidées statiquement. Par contre, il est vrai que le polymorphisme paramétrique est en général décidé statiquement.

Exercice 8: 2 points
Dans un langage de programmation donné sans construction `instance_of` ou équivalent, comment tester statiquement si un type concret T est sous-type d'un autre type concret U ? Quelle propriété des types (qui n'est pas la propriété de sous-type) utilise t'on pour assurer la validité d'un tel test ?

✓ La façon la plus simple de tester la propriété de sous-type est d'utiliser la propriété de substitution : si $U <: T$, alors toute valeur de type U peut être utilisée à la place d'une autre valeur de type T. Une simple affectation `T t = new U()` permet donc de répondre à la question.

Exercice 9: 2 points
L'extension C 11 du langage C a introduit une construction `_Generic` pouvant être utilisée de la manière suivante :

```c
const char* cos_s(const char* x) { return x; }

#define cos(X) _Generic((X), \
    long double: cosl, \
    float: cosf, \
    const char*: cos_s, \
    default: cos \
    )(X)

int main(void) {
    long double x = 1.0L;
    float y = 5.0;
    const char* z = "sept";
    printf("cos(x) = %.40Lf\n", cos(x));  // cos(x) = 0.5403...2404
    printf("cos(y) = %.40f\n",  cos(y));  // cos(y) = 0.2835...0000
    printf("cos(z) = %s\n",     cos(z));  // cos(z) = sept
}
```

Que peut-on dire de la fonction `cos` ? Quelle propriété des systèmes de types est mise en valeur dans cet exemple ? Proposer une application (mais pas un exemple) d'une telle construction.

✓ Le fonction `cos` rassemble plusieurs implémentations de types différents sans liens entre eux. Il s'agit d'un polymorphisme de surcharge. Disposer d'une telle construction dans le langage C permet d'appliquer des techniques de programmation générique (cf. `tgmath.h` pour les fonctions mathématiques), mais aussi de disposer d'un opérateur `instance_of` statique (qui n'a vraiment de sens qu'en conjonction avec les techniques précédentes)

Exercice 10: 2 points
Construire le type en Java d'une fonction `sort` prenant en paramètre un prédicat de comparaison et une liste, et triant cette liste. Faire en sorte que le type soit le plus précis possible. Quelles sortes de vérifications sont-ici impossibles à écrire en Java mais pourraient l'être dans un langage avec un système de types plus sophistiqué ?

✓ `Interface Comparator<T> {int compare(T o1, T o2) }`
`static <T> void sort(List<T> list, Comparator<? super T> c)`
Les vérifications faites en Java sont de simples vérifications de types, mais n'assure en rien de propriétés plus générales. Ainsi, l'objet `Comparator` n'est en rien assuré de vérifier les propriétés classiques d'une relation de comparaison. Assurer ce genre de propriété n'est pas possible dans les langages classiques, et demanderait de passer par de la preuve de programme, possiblement réalisée de manière automatique mais en toute généralité assistée.