Année 2018-2019 Session 1

Systèmes de Types et Programmation
IT234

Filière : Informatique, 2ème Année

Date de l'examen : 29/05/2019 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 (même pour le code). La précision des réponses fournies est un critère important de l'évaluation. La syntaxe des langages de programmation utilisés ne sera pas vérifiée à la lettre. Le barème est purement indicatif.

Exercice 1: 1 points

Comparer les aspects statiques et dynamiques des systèmes de types, en invoquant à chaque fois un exemple d'application.

✓ La comparaison se fait principalement sur les formes de vérification offertes. Les vérifications statiques assurent des propriétés (comme la *type safety*) sur *toutes* les exécutions du programme (ex : `static_cast` en C++, algorithme d'inférence de type en OCaml, algorithme de sélection Java). Les vérifications dynamiques assurent aussi de propriétés de sûreté (éventuellement aussi de *type safety*), mais uniquement sur des exécutions particulières (ex : vérification des accès dans les bornes d'un tableau tableau, prédicats de types comme `instanceof` en Java et `dynamic_cast` en C++). Les premières offrent naturellement plus d'assurances que les secondes.

Exercice 2: 1 points

Inférer le type de l'expression suivante : λ`f`.λ`y`.λ`z`.(`f` (`y == z`)) (aucun arbre de dérivation n'est demandé, la fonction `==` est l'opérateur infixe de comparaison standard)

✓ En OCaml, `fun f y z → f (y == z)` renvoie `(bool → 'a) → 'b → 'b → 'a`.

Exercice 3: 2 points

En Racket, la fonction `flatten` possède la documentation suivante :

```
(flatten v) ;; takes a parameter v : any, returns a list
```

Flattens an arbitrary S-expression structure of pairs into a list. More precisely, `v` is treated as a binary tree where pairs are interior nodes, and the resulting list contains all of the non-null leaves of the tree in the same order as an inorder traversal.

Examples :

```
> (flatten '((a) b (c (d) . e) ()))
'(a b c d e)
```

Au vu de cette documentation, proposer (en justifiant) un type pour la fonction `flatten` dans un langage de programmation possédant des types paramétriques.

✓ Classiquement, la fonction `flatten` agit sur une liste et renvoie une liste. Néanmoins, il n'est pas possible d'écrire le type du paramètre sous la forme List[T] pour un T donné, puisqu'une telle liste peut contenir un niveau quelconque de sous-listes. Considérons un type BinaryTree[T] représentant les arbres binaires dont les feuilles sont étiquetées par des valeurs de type T. Les valeurs de ce type sont en bijection avec les listes de profondeur arbitraire. Alors, `flatten` : BinaryTree[T] → List[T] ou `flatten` : BinaryTree[T] → BinaryTree[T].

Exercice 4: 3 points

Définir ce qu'est un type abstrait de données. Illustrer cette définition dans le langage de votre choix parmi {C++, Java, OCaml} en proposant un exemple dans lequel interagissent deux types abstraits de données distincts. L'exemple doit permettre des gérer des joueurs participant à un jeu multi-joueur en ligne (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 `player` et `game` sont tous les deux abstraits, et pourtant interagissent l'un avec l'autre à travers leurs interfaces :

```
module P : sig (* Interface *)
  type player
  val join        : player → game → unit
  val leave       : player → game → unit
  val update_rank : player → int → unit
end = struct   (* Implementation *)
  type player = { name : string, ranking : int }
  (* ... *)
end
```

```
module G : sig (* Interface *)
  type game
  val players : game → player list
  val kick    : game → player → unit
  val finish  : game → unit
end = struct   (* Implementation *)
  type game = { players : player list, scores : int list }
  (* ... *)
end
```

Il devient alors possible de fournir plusieurs implémentations différentes sous forme de différents types concrets. La donnée d'une interface Java et de son implémentation par une classe concrète est un exemple de type abstrait de données.

Exercice 5: 3 points

Le langage OCaml gère les exceptions avec une syntaxe proche de l'exemple suivant (`head`

et `tail` sont les accesseurs classiques sur les listes) :

```
let rec indexOf x xs = try
                         if x == []
                         then raise Not_Found
                         else if (x == head xs)
                              then 0
                              else 1 + (indexOf x (tail xs))
                       with (fun Not_Found → -1);;      (* val indexOf : 'a → 'a list → int = <fun> *)
```

En partant du λ-calcul simplement typé décrit en cours, décrire les modifications du langage nécessaires pour pouvoir travailler avec des exceptions en utilisant `raise` et `try .. with`. (*Remarque* : ces modifications prennent usuellement trois formes distinctes, une certaine latitude est laissée quant à la complétude des modifications proposées).

✓ Les exceptions sont décrites au chapitre 14.3 du Pierce :

Syntax and Types

t ::=	...	*terms*
	try t with t	*handle exception*
	raise t	*throw exception*
T ::=	...	*types*
	Exn	*exception type*

Evaluation rules

$$\frac{\texttt{t} \to_\beta \texttt{t'}}{\texttt{raise t} \to_\beta \texttt{raise t'}}$$

$$(\texttt{raise v})\ \texttt{t} \to_\beta \texttt{raise v}$$

$$\texttt{v}\ (\texttt{raise v'}) \to_\beta \texttt{raise v'}$$

$$\texttt{raise raise v} \to_\beta \texttt{raise v}$$

$$\frac{\texttt{t}_1 \to_\beta \texttt{t}_1'}{\texttt{try t}_1 \texttt{ with t}_2 \to_\beta \texttt{try t}_1' \texttt{ with t}_2}$$

$$\texttt{try v with t} \to_\beta \texttt{v}$$

$$\texttt{try raise v with t} \to_\beta (\texttt{t v})$$

Typing rules

$$\frac{\Gamma \vdash \texttt{t} : \mathsf{Exn}}{\Gamma \vdash \texttt{raise t} : \mathsf{T}}$$

$$\frac{\Gamma \vdash \texttt{t}_1 : \mathsf{T} \qquad \Gamma \vdash \texttt{t}_2 : \mathsf{Exn} \to \mathsf{T}}{\Gamma \vdash \texttt{try t}_1 \texttt{ with t}_2 : \mathsf{T}}$$

La première chose à se demander est la nature des exceptions, ce qui amène à la création d'un type Exn (amené en particulier à être le type du paramètre de `raise`). Les règles d'évaluation sont nombreuses, mais toutes ne sont pas demandées pour réussir l'exercice. La plus grande difficulté est de comprendre que l'expression à droite du `with` est une fonction. Enfin, les règles de typage sont relativement simples (cf. la fonction `failwith : string → 'a` en OCaml).

Exercice 6: 2 points
Considérons un type Factory[T] représentant des objets capables de générer à travers une méthode `create` une valeur de type T. Justifier, en appliquant les principes vus en cours, les règles de variance possibles pour le type T.

✓ Selon le *get-and-put* principle, il n'est possible que d'obtenir des valeurs de type T depuis une Factory[T]. Ce type peut donc être covariant en T, même si ce n'est pas une condition nécessaire. Par contre, il ne peut pas être contravariant, sous peine de perte de la sûreté de typage.

Exercice 7: 3 points
Le langage Eiffel est un langage de programmation orienté-objet classique. Lors d'un héritage, redéfinir une méthode se fait à l'aide du mot-clé `redefine`, comme on peut le voir dans le code suivant :

```
class Sportif feature                                    -- Base class
    meet (s: Sportif) do s.meet_s end                    -- def of method meet in Sportif
    meet_s do io.put_string ("Game␣on,␣Sportif␣!") end
end
class Pongiste inherit Sportif redefine meet end feature -- Inherited class
    meet (p: Pongiste) do p.meet_p end                   -- def of redefined method meet in Pongiste
    meet_p do io.put_string ("Ball␣on,␣Pongiste␣!") end
end
(create {Sportif}).meet(create {Pongiste})  -- prints "Game on, Sportif !"
(create {Pongiste}).meet(create {Pongiste}) -- prints "Ball on, Pongiste !"
(create {Pongiste}).meet(create {Sportif})  -- compilation error : Pongiste.meet takes a Pongiste as param
```

Donner le nom précis de la forme de polymorphisme à l'oeuvre ici. Comparer cet exemple avec l'équivalent Java (le code n'est pas demandé). Expliquer les intérêts et les dangers d'une telle construction en Eiffel, en exhibant un exemple de pseudo-code dangereux.

✓ Il s'agit de redéfinition de méthode (*overriding* en anglais), dans un cadre de polymorphisme d'inclusion. Mais dans cet exemple, la méthode `meet(Pongiste)` n'est pas un sous-type de `meet(Sportif)`, du fait de la contravariance du paramètre de la fonction. Le même exemple en Java type correctement parce que la fonction `meet` devient surchargée. L'erreur de type dans l'exemple montre qu'une telle surcharge n'existe pas en Eiffel. Par conséquent, le code produit n'est plus *type-safe*, comme le montre l'exemple suivant :

```
s: Sportif; p: Pongiste
p := create {Pongiste}     -- initialize p
s := p                     -- cast p into a Sportif
s.meet (create {Sportif}) -- call to inexistant meet_p method for the argument
```

En Eiffel, c'est appelé un CAT-call (Changing Availability or Type), et cela mène à une erreur à l'exécution. Néanmoins, et si l'on accepte cette faiblesse, cette construction pourrait avoir un intérêt si l'on désirait empêcher des sportifs de types différents (dont le type est connu statiquement) de se rencontrer, par exemple un `Pongiste` et un `Rugbyman`.

Exercice 8: 2 points
L'item 29 d'*Effective Java* de J. Bloch propose un motif de programmation nommé "type-safe heterogeneous containers". Ce motif permet par exemple à un client de gérer des

conteneurs rassemblant plusieurs instances de différentes classes sans relations les unes avec les autres.

```java
// Typesafe heterogeneous container pattern - API
public class Container {
  public <T> void put(Class<T> type, T instance);
  public <T> T    get(Class<T> type);
}
Container c = new Container();
c.put(String.class, "Java");       // inserts a value of type String
c.put(Integer.class, 0xcafebabe);  // inserts a value of type Integer
String fav = c.get(String.class);  // retrieve the value of type String
System.out.println(fav);           // prints "Java"
```

A quoi se réfère l'adjectif *type-safe* dans ce contexte ? Expliquer les avantages et inconvénients d'un tel motif, en comparant ses aspects statiques et dynamiques.

✓ Il s'agit d'une technique permettant de gérer des enregistrements/records extensibles. La sûreté ici est assurée par le polymorphisme paramétrique des méthodes `get`/`put`, qui assurent que le type de l'élément stocké dans les `Container` est le même que celui qui en est retiré : il est préservé au cours de l'exécution. Un tel conteneur est dynamique, et donc flexible car il peut être étendu à la demande. Néanmoins, aucune vérification statique n'est possible sur son contenu réel. En comparaison, un enregistrement OCaml (par ex. `type hostinfo = { host : string; time : Time.t; arch : string }`) n'est pas un conteneur flexible (ses 3 champs sont fixes), mais il assure statiquement leur présence à l'intérieur.

Exercice 9: 3 points

La bibliothèque Dyno en C++ permet d'écrire le code suivant :

```cpp
// Define the interface of something that can be drawn
DYNO_INTERFACE(Drawable, (draw, void (ostream&) const));

struct Square   { void draw(ostream& out) const { out << "Square"; } };
struct Circle   { void draw(ostream& out) const { out << "Circle"; } };
struct Triangle { };

DYNO_EXTEND(Drawable, vector<int>,
            [](vector<int> const& v, ostream& out) { // a lambda-function that outputs the contents of v
            });

void f(Drawable const& d) { cout << "I␣am␣drawing␣a␣"; d.draw(cout); }

int main() {
  f(Square{});              // prints "I am drawing a Square"
  f(Circle{});              // prints "I am drawing a Circle"
  f(vector<int>{1, 2, 3}); // prints "I am drawing a 1 2 3 "
  // f(Triangle{});         // type error
}
```

Les auteurs de cette bibliothèque affirment "résoudre le problème du polymorphisme à l'exécution mieux que le C++ standard". Quelle est la manière standard/classique d'écrire un tel code en C++ ? En quoi Dyno le résout-il différemment ? A quelle construction vue en cours cette bibliothèque peut-elle être comparée ? *Bonus* : en quoi cette construction

est-elle intéressante ?

✓ Classiquement, un tel code s'écrirait en programmation orientée objet classique, avec une interface (méthode virtuelle en C++) implémentée dans des classes dérivées. En Dyno, il n'y a pas de lien explicite entre l'interface et les classes dérivées. Au vu de l'erreur de type, le compilateur se débrouille pour vérifier que les méthodes de l'interface existent dans les classes utilisées. Un mécanisme (équivalent à celui de sélection de méthode en objet) doit permettre de sélectionner le code à utiliser pendant l'exécution. Une telle construction, parfois aussi appelée *retroactive extension*, est apparentée au *type classes* en Haskell, aux *interfaces* en Go, et aux *trait objects* en Rust. Contrairement à l'héritage classique, il est possible d'implémenter de telles interfaces même si la classe originale ne l'implémentait pas au moment de sa définition, ce qui est plus flexible qu'en POO classique.