Année 2019-2020 Session 1

Systèmes de Types et Programmation
IT234

Filière : Informatique, 2ème Année

Date de l'examen : 08/05/2020 Durée de l'examen : 6h

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: 2 points**

Qu'est-ce que la vérification (dans le contexte de la programmation) ? Illustrer votre définition par un exemple et un contre-exemple tiré de la théorie des types.

✓ La vérification peut être définie comme le domaine de l'informatique théorique ayant trait aux preuves de propriétés logiques vérifiées par les comportements (évaluations) de programmes. La théorie des types en est une des facettes. Parmi les propriétés logiques qu'elles permet de vérifier, la plus simple est certainement que les valeurs prises par les programmes restent confinées à des ensembles bien définis, empêchant par exemple d'additionner des locomotives et des fleurs en contraignant les valeurs additionnables. Il existe bien sûr de nombreuses limites, dûes soit à l'indécidabilité desdites propriétés (cf. division par zéro), soit aux limites d'expressivité des systèmes de types (cf. axiomes de l'égalité en Java).

**Exercice 2: 2 points**

Le langage C# permet de définir des classes génériques en utilisant la syntaxe suivante :

```
interface IEatable { void Eat(); }

class FoodBox<T> where T : IEatable {
  List<T> food;
  void EatThemAll() { // Eat all foods in food
      food.ForEach((f) ⇒ f.Eat());
  }
}
```

Exprimer le type de `EatThemAll` en utilisant une formule logique. Comment appelle t'on cette forme de généricité ?

✓ Il s'agit ici de généricité ou de polymorphisme paramétrique contraint. Le type de la fonction `EatThemAll` peut s'écrire :

$$\text{EatThemAll} ::= \forall T \text{ such that } T <: \text{Eatable}, \text{Foodbox}[T] \rightarrow \text{Unit}$$

Il s'agit d'une forme de type faisant intervenir ici à la fois le polymorphisme paramétrique et le polymorphisme d'inclusion. Rappelons que l'on retrouve une telle forme de polymorphisme par exemple dans les fonctions de comparaison en Haskell, qui ne peuvent s'appliquer que sur les valeurs appartenant à la classe `Eq a`.

**Exercice 3: 2 points**

Une bibliothèque propose un système de validation pour des formulaires en utilisant les types algébriques d'OCaml de la manière suivante :

```
type form_result = int
type validation_error = FormOk of form_result | FormBadRequest | FormIncomplete

let validate_authenticity result =
  if   (* do some validations on the parameter *)
  then FormOk result
  else (* compute some error status *)        (* val validate_authenticity : form_result → validation_error *)

let render_html result = match (validate_authenticity result) with
  | FormOk v        → render (Code 200) "<html>Correct : "^(string_of_int v)^"</html>"
  | FormBadRequest  → render (Code 400) "<html>Bad Request</html>"
  | FormIncomplete  → render (Code 200) "<html>Try again</html>"
```

Une personne propose d'implémenter une nouvelle version de la même bibliothèque en Java utilisant le mécanisme des exceptions pour implémenter le couple (`validate_authenticity`, `render_html`) : la validation peut renvoyer des exceptions, qui sont récupérées par la fonction de rendu. Comparer les avantages et les inconvénients des deux implémentations.

✓ La question de la validation d'un objet est liée à l'idée de fonction partielle : si l'objet est valide, les traitements peuvent s'effectuer, sinon il faut gérer les cas d'erreurs. Les deux solutions (types algébriques et exceptions) permettent de répondre à la question. La solution "algébrique" fixe les cas d'erreurs possibles à travers le type `validation_error`. D'un côté, une telle solution est sûre au sens où elle assure de connaître la liste exacte des cas d'erreur, et de vérifier qu'ils sont tous gérés dans le code (*exhaustivité*). D'un autre côté, c'est un mécanisme rigide dans lequel ajouter de nouveaux cas d'erreur demande de reprendre le code existant. La solution "exceptions" est plus flexible, dans la mesure où en définissant en Java une classe `ValidationException`, il est toujours possible d'étendre cette classe a posteriori, mais elle ne possède pas la propriété d'exhaustivité.

**Exercice 4: 2 points**

La norme C++17 offre la possibilité de déclarer des valeurs de type `std::variant<T1, ..., TN>`. De telles valeurs peuvent ensuite être utilisées à l'aide d'une fonction `std::visit` de la manière suivante :

```cpp
// 3 noisy classes with a say() method
class Animal               { public: virtual void say() const { std::cout << "mew ?\n"; } };
class Lion : public Animal { public: virtual void say() const { std::cout << "roar !\n"; } };
class Autobus              { public: virtual void say() const { std::cout << "vroum !\n"; } };
// 1 client acting on these classes
struct PetSomething {
    void operator()(const Animal& a)  { cout << "It says "; a.say(); }
    void operator()(const Autobus& a) { cout << "It makes "; a.say(); }
};

int main(void) {
  std::variant<Animal, Autobus> thing; // Declaration of thing
  thing = Animal();
  std::visit(PetSomething{}, thing);   // Displays "It says mew ?"
  thing = Autobus();
  std::visit(PetSomething{}, thing);   // Displays "It makes vroum !"
  thing = Lion();
  std::visit(PetSomething{}, thing);   // Displays "It says mew ?"
  thing = 666;                       } // Type error
```

Exprimer le type de `thing` par une formule logique. Donner le nom du polymorphisme à l'oeuvre ici, et expliquer le mécanisme (lié aux types) qui permet à ce code de produire de tels résultats.

✓ Le type `std::variant` est un type *union*, que l'on peut décrire de la manière suivante :

$$\text{thing} ::= \text{Animal} \cup \text{Autobus}$$

Cela peut se voir à travers la manière dont on peut affecter à `thing` n'importe quelle valeur de l'un de ces types, mais pas d'autres.

La fonction `std::visit` relie la valeur `thing` à la valeur `PetSomething`. Au vu des résultats, `std::visit` appelle l'une des méthodes de `PetSomething`, manifestement en fonction du type de `thing`. Il s'agit d'un mécanisme de *sélection*, dans le cadre d'un *polymorphisme de surcharge*. Ici, la méthode surchargée est l'opérateur `()` de `PetSomething`. Pour un bonus, noter que ce polymorphisme est décidé statiquement, et donc que le `Lion` ne peut appeler sa propre méthode `say()`.

Noter que le type de `PetSomething` peut se décrire indifféremment comme :

$$\text{PetSomething} ::= (\text{Animal} \cup \text{Autobus}) \to \text{Unit} \quad \text{ou} \quad (\text{Animal} \to \text{Unit}) \cap (\text{Autobus} \to \text{Unit})$$

... parce que les deux formules logiques sont équivalentes.

**Exercice 5: 5 points**

La plupart des langages de programmation autorisent d'affecter des valeurs dans des références, par un opérateur d'affectation. La syntaxe classique consiste à écrire une affectation `a = b` pour affecter la valeur `b` à l'intérieur de `a`. Pour les besoins de cet exercice, on considère cette affectation comme une fonction `assign`, et on écrit `a = b` comme `assign(a,b)`.

1. Proposer un type générique pour `assign` mettant en jeu le polymorphisme *paramétrique* (par exemple en OCaml). Expliquer en quelques mots le genre de vérifications que l'on peut attendre d'un tel type.

En C++, lorsque l'on affecte des références, il peut se produire un phénomène appelé *slicing*, qui peut provoquer des comportements inattendus. Par exemple, dans l'exemple suivant, les deux affectations amènent à un objet dans un état incohérent :

```
struct A { int x;
           A(int x) : x(x) {}
};
struct B : public A {
           int y;
           B(int x, int y) : A(x), y(y) {}
};

int main(void) {
  B b1 = B(1, 2);
  B b2 = B(3, 4);
  A& a_ref = b2;    // assign(a_ref, b2) -- b1 contains (1,2), b2 contains (3,4)
  a_ref    = b1; } // assign(a_ref, b1) -- b1 contains (1,2), b2 contains (1,4)
```

2. Expliquer le problème apparaissant dans le code précédent.
3. Une personne vous affirme que "par défaut, les opérateurs de copie en C++ sont statiques". Expliquer ce qu'elle entend par là, montrer le fonctionnement de ces opérateurs sur l'exemple précédent, et proposer une manière de se prémunir des problèmes de slicing.

✓

1. En OCaml, l'opérateur d'affectation s'écrit (`:=`), et indique clairement que l'on stocke une valeur à l'intérieur d'une référence.

   $$\texttt{assign} : \forall T,\ \mathsf{Ref}[T] \to T \to \mathsf{Unit}$$

   ```
   (:=);;  (* 'a ref → 'a → unit = <fun> *)
   ```

   Le véritable opérateur d'affectation dans d'autres langages, comme en C++, est usuellement plus compliqué, du fait des opérations de conversions multiples.
   Un tel type assurerait que lorsqu'on écrit une affectation `assign(x,y)`, le type de `y` est forcément soit le même type que celui de `x`, soit un sous-type dans un langage avec polymorphisme d'inclusion.
2. En C++, il existe des différences entre les références et les pointeurs, néanmoins dans l'exemple précédent, la référence `a_ref` agit comme un pointeur sur la zone mémoire correspondant à `b2`. Lorsque l'on affecte `b1` à l'intérieur de la référence, on copie le contenu de `b1` à l'intérieur de `b2`. Néanmoins, la copie ici ne copie que la composante de `b1` faisant partie du type A. Ce ne serait pas grave si on n'avait accès qu'à `a_ref`, mais la valeur `b2` devient dans un état incohérent.
3. L'opérateur d'affectation est une méthode comme une autre dans les classes C++, et donc il ne dispose pas du mot-clé `virtual`. Cela signifie que, sans modification explicite du type de ces opérateurs, il s'agit d'un polymorphisme de surcharge. Dans l'exemple proposé, l'affectation `a_ref = b1` copie un `A` à l'intérieur de `b1` du fait du type apparent de `a_ref`. Au final, seul le champ `x` est copié, amenant `a_ref` dans un état potentiellement incohérent. Pour éviter ce genre de comportement, il faut rendre les opérateurs d'affectation `virtual` pour activer la sélection dynamique (`dynamic dispatch`).

**Exercice 6: 2 points**
Une API propose une interface Transport[T] est utilisée pour transmettre de l'information entre deux processus distants à la manière d'une FIFO.

```
interface Transport<T> {
  void send(T t);
  T receive();
  // ... other boilerplate methods
}

Transport<Cat> tr = initTransport();
Thread th         = new Thread();
registerTransport(th, tr);  // register the transport as usable in the thread

tr.send(new Cat("Mungojerrie")); // inside the main loop
Cat c = tr.receive()             // inside the thread
```

L'API fournit une méthode permettant de chaîner deux transports l'un après l'autre, nommée `chainTransports` , qui prend deux transports `t1` et `t2`, et produit un transport `res` (tout neuf) qui connecte `t1` et `t2`. La connection entre `t1` et `t2` se fait à la manière d'un pipe : un élément produit par `t1` (`receive`) est envoyé à `t2` (`send`). L'appel à la méthode `send` sur `res` déclenche l'appel à la méthode `send` de `t1` et l'appel à la méthode `receive` sur `res` déclenche l'appel à la méthode `receive` de `t2`.

```
static <T> void chainTransports(Transport<T> t1, Transport<T> t2, // inputs
                                Transport<T> res);                // output
```

Néanmoins, un client de cette API estime qu'elle est trop contraignante, parce qu'elle impose de chaîner des transports d'éléments du même type. Il propose de remplacer le prototype de la méthode par :

```
static <T> void chainTransports(Transport<? extends T> t1, Transport<? extends T> t2, //inputs
                                Transport<T> res);                                    //output
```

En quoi est-ce une mauvaise idée ?

✓ La classe Transport[T] est typiquement une classe utilisant la variable de type T en écriture (`send`, position contravariante) et en lecture (`receive`, position covariante). Les règles de variance comme le *get-and-put principle* affirment que dans ce cas, il faut que la classe soit invariante selon T.
Le prototype proposé pour la méthode `chainTransports` rend le paramètre de type *covariant*, ce qui est dangereux. Si T = Animal, cette méthode devrait permettre de chaîner un transport de Cat avec un transport de Dog, ce qui posera problème lors de la conversion de type entre les deux transports.

**Exercice 7: 5 points**
Le langage Swift a une gestion relativement avancée des *optionals*, des types de valeurs pouvant soit être initialisées, soit être égales à `nil`. En particulier, il est possible d'écrire du code qui "traverse" les *optionals* de la manière suivante :

```
class Person    {
  var residence: Residence?          // may be nil or Optional(Residence)
}
class Residence {
  var numberOfRooms : Int = 1
  init(rooms: Int) { numberOfRooms = rooms }
}

let john = Person()
print(john.residence)                // nil
print(john.residence?.numberOfRooms) // nil, no error
john.residence?.numberOfChimneys     // type error, no attribute 'numberOfChimneys'
john.residence = Residence(rooms: 2)
print(john.residence)                // Optional(Swift.Residence)
print(john.residence?.numberOfRooms) // Optional(2)
```

En partant du $\lambda$-calcul simplement typé décrit en cours (étendu avec les *record types*), décrire les modifications du langage nécessaires pour ajouter les *optionals* avec un opérateur `(?.)` qui permette d'écrire des expressions s'évaluant même sur des champs non définis des enregistrements.

✓ Un ensemble de modifications pour gérer les types options :

**Syntax and Types**

| | | |
|---|---|---|
| t | ::= ... | *expressions* |
| | Nil | *nil expression* |
| | Optional(t) | *optional expression* |
| | t-▷?t | *optional projection* |
| v | ::= ... | *values* |
| | Nil | *nil value* |
| | Optional(v) | *optional value* |
| T | ::= ... | *types* |
| | Optional[T] | *option type* (meaning T?) |

**Evaluation rules**

$$\frac{\texttt{t} \to_\beta \texttt{t'}}{\texttt{Optional(t)} \to_\beta \texttt{Optional(t')}}$$

$$\texttt{Nil-▷?t} \to_\beta \texttt{Nil}$$

$$\texttt{Optional}(\{l_i = \texttt{v}_i\})\texttt{-▷?}l_j \to_\beta \texttt{Optional}(\texttt{v}_j)$$

$$\frac{u \text{ is not a field in } \{l_i\}_i}{\texttt{Optional}(\{l_i = \texttt{v}_i\})\texttt{-▷?}u \to_\beta \texttt{Nil}}$$

**Typing rules**

$$\vdash \texttt{Nil} : \texttt{Optional[T]}$$

$$\frac{\Gamma \vdash \texttt{t} : \texttt{T}}{\Gamma \vdash \texttt{Optional(t)} : \texttt{Optional[T]}}$$

$$\frac{\Gamma \vdash \texttt{t} : \texttt{Optional}[\{l_i : \texttt{T}_i\}]}{\Gamma \vdash \texttt{t-▷?}l_i : \texttt{Optional}[\texttt{T}_i]}$$

Accessoirement, la possibilité de chaîner les projections de manière optionnelle est une des possibilités offerte pas la monade option (aussi appelée monade Maybe:), décrite par exemple dans `http://learnyouahaskell.com/a-fistful-of-monads`.