Année **2023-2024** Session **1**

Systèmes de Types et Programmation
IT234

Filière : Informatique, 2ème Année

Date de l'examen : 07/05/2024 Durée de l'examen : 2h00

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 barême est purement indicatif.

Exercice 1: 2 points

En vous appuyant sur des exemples de programmation (et éventuellement de types), expliquer en quoi il est intéressant de vérifier des programmes statiquement et dynamiquement.

✓ Lorsqu'on s'intéresse à des programmes et à leur vérification, il est possible de considérer si on exécute ou pas un programme. Si l'on considère que l'on exécute le programme, les vérifications que l'on peut faire sont dites *dynamiques* (par exemple des tests, du monitoring ...). Il est connu que la plupart des propriétés concernant l'exécution des programmes sont indécidables. De plus, le nombre d'exécutions possibles fait souvent ressortir une explosion combinatoire. A contrario, si l'on considère le programme hors de toute exécution (par exemple à travers les types, du model checking ou de la preuve formelle), on se place dans un cadre *statique*. Il devient alors envisageable de vérifier des propriétés sur un objet (le code du programme) d'une complexité moindre. Par exemple, l'algorithme d'inférence de type de Hindley-Milner a une complexité en général quadratique en la taille du programme. Noter que les propriétés que l'on va vérifier dans les deux cas peuvent être différentes et se compléter, n'en rendant pas une meilleure que l'autre.

Exercice 2: 3 points

Considérons le code de la fonction `type_me` écrite ici en OCaml :

```
let rec type_me n acc =
  match n with
  | 0 → acc 0
  | _ → type_me (n-1) (fun r → acc (n+r));;
```

Inférer le type de cette fonction (aucun arbre de dérivation n'est demandé).
(*Bonus minimaliste* : que réalise cette fonction ?)

✓ Le compilateur OCaml propose le type suivant pour `type_me` :

```
val type_me : int → (int → 'a) → 'a = <fun>
```

Le piège ici était de laisser échapper le fait que le type de retour soit polymorphe (ce qui ne devrait pas arriver si on applique rigoureusement l'algorithme vu en cours). Il s'agit d'une fonction écrite avec *passage de continuation* (cf. `https://en.wikipedia.org/wiki/Continuation-passing_style`). Ici, `type_me n (fun x → x)` calcule la somme des entiers de 1 à n.

Exercice 3: 4 points
Considérons le type de la fonction `map` définie sur les flots (*streams*) en Java :

```
// A stream is a sequence of elements supporting sequential and parallel aggregate operations.
class Stream<T> {

  // The 'map' method in the class Stream<T>
  <R> Stream<R>   map(Function<? super T,? extends R> mapper)
  // Returns a stream consisting of the results of applying the
  // 'mapper' to the elements of this stream.
}
```

```
// An example of the usage of the map method
List<Integer> numbers = Arrays.asList(1, 2, 3, 4, 5);
// Using map to convert each integer to its square
List<Integer> squares = numbers.stream()                    // Transform the list into a stream
                             .map(n → n * n)
                             .collect(Collectors.toList()); // Transform the stream back to a list
// squares contains [ 1, 4, 9, 16, 25 ]
```

Rappelons ici que le type `Function<T,R>` représente une fonction qui accepte un argument de type `T` et produit un résultat de type `R`.

1. Que sont les indications de types contenant des "`?`" (points d'interrogations) dans le prototype de la méthode `map` ?
2. Expliquer chacune des deux indications de types en question (en particulier donner le nom associé à chaque indication).
3. Dans quel but donne t'on des indications de types aussi complexes ?

✓

1. Il s'agit de jokers ou de *wildcards* en Java. Ces indications servent à indiquer des relations de variance sur les types de données, ici le type `Function`. Ici, ils caractérisent les types utilisables comme paramètres de la méthode `map`.
2. La fonction `map` transforme chacun des éléments d'un flot à travers une fonction `mapper` passée en paramètre. Cette fonction `mapper` vérifie les deux propriétés suivantes :
   - Elle accepte des éléments qui sont des sur-types de `T`. En effet, elle va être appliquée aux éléments à l'intérieur d'un flot de type `Stream<T>`. L'indication s'applique ici à un type en position *contravariante*.
   - Elle renvoie des éléments qui sont des sous-types de `R`. En effet, ces éléments sont utilisés pour remplir un flot de type `Stream<R>`. L'indication s'applique ici à un type en position *covariante*.
3. Les types paramétriques en Java sont par défaut invariants. Si on donnait à la méthode `map` un type sans indications, de la forme suivante :

   ```
   <R> Stream<R> map(Function<T,R> mapper)
   ```

   alors cela obligerait les utilisateurs de `map` de passer des fonctions avec des types contraints par le flot d'entrée et de sortie. Dans l'exemple, on ne pourrait passer *que* des fonctions `Integer` → `Integer`. Cela limite par exemple la réutilisation de fonctions existantes n'ayant pas exactement le type demandé. Les jokers permettent ici d'autoriser de passer des fonctions dont le type est un *sous-type* de `Integer` → `Integer`, et donc d'utiliser le polymorphisme de manière plus appuyée.

Exercice 4: 4 points

La bibliothèque Pydantic en Python se présente comme une bibliothèque de "validation de données". Le code fourni ci-après donne un exemple d'utilisation de cette bibliothèque. Il y apparaît le définition d'une classe `User` très simple, ainsi que quelques tests de validation. Le code annoté avec des types, qui peut être vérifié par un type-checker comme `mypy`. Pour rappel, en Python, les types sont indiqués après le caractère ":", comme en OCaml.

Dans cet exemple, les noms `str`, `Login` et `Password` sont des noms de types. `str` représente le type des chaînes de caractères en Python. À droite sont fournis d'une part le résultat de exécution du type-checker `mypy`, mais aussi le résultat de l'exécution du fichier :

```python
Login = Annotated[
    str,                # The type of Python strings
    StringConstraints(
        min_length = 3,
        max_length = 10)
        ]

Password = Annotated[
    str,
    StringConstraints(
        min_length = 3,
        pattern = "^[a-z0-9]+$")
        ]

class User(BaseModel):
    login: Login
    password: Password

def test_user(log: Login, pwd: Password):
    try:
        u = User(login=log, password=pwd)
        print(f"User '{login}' is OK")
    except ValidationError as err:
        print(f"User '{login}' is KO")
        print(err)

# test_user(True,    7)   # Incompatible type error
test_user("georgy", "azerty0") # Should be OK
test_user("g",      "azerty0") # Should be KO
test_user("georgy", "^!$=@")   # Should be KO
```

```
% mypy user.py
    Success: no issues found in 1 source file
```

```
% python user.py
    User 'georgy' is OK

    User 'a' is KO
1 validation error for User
login
  String should have at least 3 characters
  [type=string_too_short,
   input_value='g',
   input_type=str]

    User 'georgy' is KO
1 validation error for User
password
  String should match pattern '^[a-z0-9]+$'
  [type=string_pattern_mismatch,
   input_value='^!$=@',
   input_type=str]
```

Le résultat de la commande `mypy` doit être considéré comme une vérification fiable des types fournis dans le fichier. Noter que la ligne 27 a été commentée parce qu'elle engendrait une erreur avec le type-checker.

1. Les types `Login` et `Password` mélangent deux informations. Lesquelles ?
2. Qu'est-ce qui est ici selon vous vérifié à la compilation et qu'est-ce qui est vérifié à l'exécution ?
3. Quel nom a t-on utilisé en cours pour désigner les types paramétrés `Login` et `Password` ?
4. À quel autre système de validation par les types cet exemple vous fait-il penser ?

✓

1. Les types `Login` et `Password` sont construits à partir de `Annotated`, un type paramétré contenant deux informations : le type de la valeur qu'il contient (dans les deux cas `str`), et une sorte d'annotation sous la forme d'un `StringConstraints`. Cette annotation est manifestement utilisée pour définir des contraintes sur les valeurs de ces types (ici de longueur, et de format grâce à une expression régulière).
2. Au vu des commandes exécutées à droite, la commande de compilation se termine sans erreur. Donc les types utilisés dans le fichier sont valides. On peut voir aux lignes 28–30 que des valeurs de type `str` peuvent être utilisées comme `Login` et `Password`, ce qui sous-entend que `str` est un sous-type de ces deux types.

   L'exécution du fichier, elle, mène à plusieurs erreurs, qui montrent que les tests ne passent pas. Ces erreurs sont des erreurs de validation associées aux contraintes ajoutées dans les types. Manifestement, la bibliothèque `pydantic` ajoute automatiquement des vérifications à l'exécution, vérifications qui sont attachées aux types définis dans le fichier.
3. Les annotations de type associées à `Login` et `Password` ressemblent fortement à des *types fantômes*. Elles correspondent aux exemples vus en cours, par exemple en LiquidHaskell, où on a annoté les types avec des propriétés logiques. Formellement, on pourrait argumenter sur le fait qu'ils n'ont pas d'influence à l'exécution, dans la mesure où ils sont utilisés pour générer du code de validation pour la classe `User`. Une réponse à cette affirmation est que les types eux-mêmes sont bien des fantômes, parce qu'en tant qu'ensemble de valeurs ils représentent l'ensemble vide. Seule l'annotation contenue dans le type est utilisée pour engendrer les vérifications.
4. Les types de la famille *refinement types* de LiquidHaskell ou Dafny sont similaires à cette approche. Mais on peut aussi évoquer les types fantômes vus en OCaml et en Haskell. La différence ici avec les approches vues en cours tient dans le choix (raisonnable) d'utiliser ces annotations pour engendre une vérification dynamique en `pydantic`, à l'opposé de la vérification statique des exemples vus en cours.

Exercice 5: 4 points

Le code suivant en Javascript est un exemple de ce que l'on peut faire dans le langage pour écrire des fonctions prenant un nombre variable d'arguments, aussi appelées *variadiques* :

```
function foldLeft(aFun, aStart, ...someLists) {
    let anAcc = aStart;
    if (someLists.length > 0) {
        for (let i=0; i<someLists[0].length; i++) {
            let values = someLists.map((l) ⇒ l[i]);  // Get the i-th element of each list
            anAcc = aFun(anAcc, ...values);          // Transform the accumulator
        }
    }
    return anAcc;
}

function times(acc, ...args) {
    return acc + args.reduce((a,b) ⇒ a*b, 1);        // Return acc + args[0]*args[1]*...*args[n-1]
}

times(1, 10, 11, 12);                          // → 1321 = 1+10*11*12
foldLeft(times, 0);                            // → 0
foldLeft(times, 0, [1,2,3]);                   // → 6    = 1+2+3
foldLeft(times, 0, [1,2,3], [4,5,6]);          // → 32   = 1*4+2*5+3*6
foldLeft(times, 0, [1,2,3], [4,5,6], [7,8,9]); // → 270  = 1*4*7+2*5*8+3*6*9
```

Dans un premier temps, on propose que tous les éléments des listes présentes dans `someLists` aient le même type (mais qui ne soit pas forcément `number`). Les types des arguments variadiques peuvent être décrits comme des tableaux. Par exemple, `f(...args: number[])` pour une fonction `f` prenant un nombre variable de paramètres de type `number`.

1. Proposer un type aussi précis que possible pour les fonctions `times` et `foldLeft`.
2. Quelle forme de polymorphisme est en jeu dans la fonction `foldLeft` ?

Supposons maintenant que les types des valeurs présentes dans `someLists` soient différents, tout en restant les mêmes liste par liste. Par exemple :

```
function stringProduct(acc, i, s) {
    return acc + i * s.length;
}

foldLeft(stringProduct, 0, [1,0,2], [ "abc", "x", "d" ]); // → 5 = 1*3+0*1+2*1
```

3. Quel problème se pose maintenant pour typer la fonction `foldLeft` ?
   Pouvez-vous y proposer une solution ? Que perd-on (si l'on perd quelque chose) ?

✓

1. Les types suivants sont valides en Typescript pour le code proposé :

```
function foldLeft<T>(aFun : (acc: T, ...elems:T[]) ⇒ T,
                     aStart : T,
                     ...someLists : T[][]) : T { ... }
function times(acc : number,
               ...args: number[]) : number { ... }
```

2. La fonction `foldLeft` utilise le polymorphisme paramétrique.
3. La fonction `foldLeft` prend une liste variable de paramètres à partir de son 3ème argument. Chacun de ces paramètres sont censés être des tableaux. Dans la question précédente, on considérait que tous ces tableaux contenaient le même type d'éléments. Donc le type `T[][]` était adapté, représentant une liste de listes d'éléments de type `T`.

   Si les types des éléments dans les tableaux diffèrent, il n'est plus possible d'écrire un type précis pour décrire l'entièreté de ces paramètres. En Typescript, il est possible d'utiliser le type `any` pour typer tous ces éléments :

```
function foldLeft(aFun : (_: any, ..._:any[]) ⇒ any,
                  aStart : any,
                  ...someLists : any[][]) : any  {
```

   Cette solution fonctionne dans les langages possédant un type "maximal" comme `Object` en Java. Elle n'est pas entièrement satisfaisante, parce qu'elle est moins précise (du point de vue des types) que la version de la première question. En particulier, elle autorise de mixer des valeurs de types différents à l'intérieur des listes, ce qui, selon l'utilisation que l'on veut faire de ces fonctions, peut être ou ne pas être le comportement attendu.

Exercice 6: 4 points

Le code suivant en Java considère un module spatial explorant différentes planètes, en utilisant un motif de programmation assez classique nommé le *visiteur*. Ci-après sont fournies une classe abstraite `Planet` implémentée par trois classes, une classe `Explorer`, et un code mettant en scène ces différentes classes.

```
abstract class Planet {
  public abstract void accept(Explorer ex);
}

class Mercury extends Planet {
  @Override public void accept(Explorer ex) {
    ex.visit(this);
  }
}

class Mars extends Planet {
  @Override public void accept(Explorer ex) {
    ex.visit(this);
  }
}

class Saturn extends Planet {
  @Override public void accept(Explorer ex) {
    ex.visit(this);
  }
}
```

```
class Explorer {
  public void visit(Mercury mercury) {
    System.out.println(
       "This␣is␣Mercury...␣it's␣hot");
  }

  public void visit(Mars mars) {
    System.out.println(
       "Here␣on␣Mars...␣life␣is␣red");
  }

  public void visit(Saturn saturn) {
    System.out.println(
       "Come␣to␣Saturn...␣check␣the␣ring");
  }

  public void visit(Planet planet) {
    System.out.println(
       "Cannot␣explore␣life␣on␣an␣unknown␣planet");
  }
}
```

```
Planet mars = new Mars();
Planet saturn = new Saturn();
Planet mercury = new Mercury();

Explorer explorer = new Explorer();

List<Planet> planetsToBeVisited =
  Arrays.asList(mars, saturn, mercury);

for (Planet planet : planetsToBeVisited) {
  planet.accept(explorer);
}
```

Le code compile sans erreurs. L'exécution produit la sortie suivante :

```
Here on Mars... life is red
Come to Saturn... check the ring
This is Mercury... it's hot
```

1. Quelle forme de polymorphisme est mis en jeu pour la méthode `accept` de `Planet` ?
2. Quelle forme de polymorphisme est mis en jeu pour la méthode `visit` d'`Explorer` ?
3. Expliquer comment les méthodes sont sélectionnées lors d'un appel comme `mars.accept(explorer)`.
4. Que se passerait-il si on décidait de factoriser le code de la méthode `accept` directement dans `Planet` ?

✓

1. Comme on peut le voir avec les mot-clés `@Override`, il s'agit d'un *polymorphisme d'inclusion* tout à fait classique, à base de redéfinition de méthode.
2. La méthode `visit` a plusieurs implémentations prenant des paramètres différents. Il s'agit donc d'un *polymorphisme de surcharge*.
3. L'appel `mars.accept(explorer)` appelle une méthode `accept` sur un objet de type apparent `Planet`. Comme cette méthode est redéfinie par les sous-classes, c'est la méthode du type réel qui est appelée, dans ce cas celle dans la classe `Mars`.

   Dans cette méthode, le code exécuté est `ex.visit(this)`. La méthode `visit` est surchargée, et donc la sélection se fait selon le type apparent. La subtilité de ce code est que le type apparent de `this` est ici `Mars`. Donc la méthode qui est appelée est celle de la classe `Explorer` qui prend un paramètre de type `Mars`.

   En pratique, on parle de *double dispatch*, même si ici l'un des deux est dynamique et l'autre est statique, parce qu'il y a une double sélection de méthode.
4. Si l'on factorisait la méthode `accept` dans l'interface `Planet`, alors le type apparent de `this` dans la classe serait `Planet`. Et donc la méthode sélectionnée serait celle prenant en paramètre un `Planet`, et donc serait incorrecte. Il s'agit d'un exemple notable de code apparemment dupliqué, mais pour lequel la duplication est nécessaire.

Noter que pour les besoins de cet exercice, on a simplifié le principe du motif *visiteur*. Dans un cas d'utilisation concret, on aurait voulu pouvoir faire plusieurs classes "visiteuses" différentes comme `Explorer`, et donc avoir une interface permettant de typer ces visiteuses.