Année **2024-2025** Session **1**

Systèmes de Types et Programmation
IT234

Filière : Informatique, 2ème Année

Date de l'examen : 14/05/2025 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: 4 points

En OCaml, il est possible de définir des *extensible variant types*. Comme leur nom l'indique, ce sont des types variants (aussi décrits dans le cours comme des *algebraic datatypes* ou types algébriques) que l'on peut étendre de la manière suivante :

```
type attr = ..                  (* Define an (empty) extensible variant type with .. *)
type attr += Str of string      (* Add a new constructor, possibly in a different file *)
type attr += | Int of int       (* Add two other constructors at once, possibly again in another file *)
             | Float of float
```

Il est alors possible de faire du pattern-matching sur ces types, mais la documentation demande à avoir un cas par défaut pour gérer les constructeurs inconnus :

```
let attr_to_string = function
  | Str s   → s
  | Int i   → Int.to_string i
  | Float f → string_of_float f
  | _       → "?"             (* The code does not compile if this statement is missing *)
```

1. Quelle construction trouvée dans d'autres langages de programmation connaissez-vous qui permette de créer des types et d'étendre leur définition dans des unités de code (par exemple des fichiers) différents ?
2. Quelle propriété importante du pattern-matching perd t'on ici ?

Supposons vouloir étendre le type `attr` dans sa propre unité de code :

```
type attr += Bool of bool
```

3. Quel problème cela pose t'il pour les définitions existantes comme `attr_to_string` ?
4. Quel mécanisme connaissez-vous dans d'autres langages de programmation qui permettraient de résoudre le même problème en étendant une telle fonction ?

✓

1. Un exemple très classiques de types que l'on peut étendre dans différents fichiers serait ceux fournis par les interfaces en Java (par exemple `Comparable` que l'on peut étendre en fournissant des classes implémentant cette interface. Noter que cela fonctionne aussi en étendant des classes, mais les classes contiennent en plus de type une implémentation, alors qu'ici on parle uniquement de type.
2. La propriété du pattern-matching vue en OCaml qui se perd ici est celle du *test d'exhaustivité* : au moment où le compilateur analyse a fonction `to_string`, il ne lui est pas possible de connaître l'entièreté du type `attr`, et donc il ne peut pas assurer que tous les cas sont bien gérés. Noter que la fonction est forcément exhaustive du fait du cas par défaut, simplement le compilateur n'est plus capable de produire une liste des cas manquants.
3. Les fonctions déjà existantes, telles que `attr_to_string`, ne peuvent tenir compte des nouveaux types, dans la mesure où elles ont été écrites sans leur connaissance. Il faut donc soit accepter leur comportement par défaut, soit en écrire de nouvelles (risque de duplication de code), soit les encapsuler dans des fonctions étendues.
4. La question semble étrange, mais elle contient l'expression "étendre une fonction". Le mécanisme adapté en programmation pour ce genre de choses est celui de l'*héritage*, ce qui est cohérent avec la réponse à la première question.

Exercice 2: 5 points

Le langage de programmation Koka propose dans son système de types des *effect types*, à savoir des annotations de types censées représenter les *effets* réalisés par un calcul. Un *effet* est un comportement particulier d'une fonction, et contient en particulier les fameux effets de bords. La documentation du langage [1] fournit les exemples suivants :

```
fun square1( x : int ) : total int   { x*x }
fun square2( x : int ) : console int { println( "a_not_so_secret_side-effect" ); x*x }
fun square3( x : int ) : div int     { x * square3( x ) }
fun square4( x : int ) : exn int     { throw( "oops" ); x*x }
```

Les 4 fonctions présentées ici prennent en paramètre un entier et renvoient un entier. En plus de cela, leurs types de retour possèdent une annotation décrite de la manière suivante :

— `total` représente l'absence d'effets ;

— `exn` représente la possibilité de lever une exception ;

— `div` représente la possibilite pour le calcul de ne pas terminer (`div`-erger) ;

— `console` représente l'action pour une fonction de produire des entrées/sorties.

Il est même possible de composer les effets de la manière suivante :

```
fun combine-effects() : <exn,div,ndet> int {
  val i = srandom-int() // non-deterministic
  throw("oops")         // exception raising
  combine-effects()     // non-terminating
  1                     // and returning an int
}
```

1. Cf. https://koka-lang.github.io/koka/doc/book.html#sec-effect-types

1. Quel polymorphisme à l'œuvre ici ?
   A quelle forme de type vue en cours ce style d'annotations fait-il penser ?

Considérons la fonction `whilef` en Koka définie de la manière suivante :

```
fun whilef(pred, func) {
  if (pred()) then {
    func();
    whilef(pred, func)
  }
}
```

2. Proposer un type pour (l'équivalent de) cette fonction en OCaml (*i.e* sans les effets).
3. Proposer un type pour cette fonction en Koka (*i.e* avec les effets).
   Remarque : les paramètres de la fonction `whilef` peuvent eux aussi produire des effets qui leurs sont propres. La question demande une petite réflexion sur le fait que le type doive en tenir compte. Si vous proposez un type, l'explication comptera autant que le type lui-même. On ne sera pas regardant sur la syntaxe utilisée si elle est claire.
4. En quoi ce style d'annotations ressemble t'il à une partie du système de types de Java ?
5. Quel(s) intérêt(s) voyez-vous à un tel système de type ?

✓

1. Les effets de Koka sont des annotations de types, vues en cours sous le nom de *types fantômes*. Ce sont des annotations qui n'ont de sens qu'au moment de la compilation, et les types associés n'influent pas l'exécution des programmes. Un exemple de tels types a été donné en cours en utilisant les types paramétriques d'OCaml et de Java.
2. La fonction `whilef` (ou en tout cas son équivalent) a le type suivant inferré par le compilateur OCaml :

   ```
   val whilef : (unit → bool) * (unit → 'a) → unit = <fun>
   ```

   Il était tout à fait acceptable de considérer que le type de retour du paramètre `func` était `unit`.
3. L'idée d'une fonction nommée `while` est certainement sa capacité à produire des calculs qui divergent. Ensuite, il faut imaginer que les deux paramètres de cette fonction peuvent eux aussi produire des effets qui leur sont propres. Dans ce cas, il est nécessaire d'inventer la notion d'effet *polymorphe*, qui est propagé des paramètres au retour de la fonction. En Koka, le type de la fonction `whilef` est le suivant :

   ```
   fun whilef(pred : () → <div|e> bool, func : () → <div|e> ()): <div|e> ()
   ```

   Le point le plus important ici était de se rendre compte que la fonction produisait un effet `div`. Le second point, motivé par la remarque était que les effets pouvaient se propager des paramètres aux résultats. On pouvait même proposer que les effets des deux paramètres soient différents.

   Le choix de Koka d'unifier les variables d'effets est expliqué dans la documentation (cf. `https://koka-lang.github.io/koka/doc/book.html#sec-polymorphic-effects`). Il s'agit d'un choix du langage de systématiquement prendre l'union des effets dans les paramètres. A minima, cela permet de simplifier les types des fonctions engendrés.
4. Le langage Java permet d'annoter ses types avec les exceptions renvoyées au sein des fonctions, ce qui se rapproche fortement de l'effet `exn`. Cela permet au compilateur de pouvoir vérifier les exceptions rattrapées et celles qui sont propagées.
5. Avoir des annotations de types précises, et inférées par le compilateur permet d'avoir de grandes qualités de vérification, que l'on peut imaginer à partir de celles de Java.

   Dans le cas de Koka, le système d'annotations est généralisé à une famille d'effets plus large (non-déterminisme, entrées/sorties...) et qui est en plus extensible. Cela permet plus de finesse dans l'écriture des types de fonctions, et un meilleur contrôle de la propagation desdits effets.

Exercice 3: 5 points

Le langage Typescript permet de construire des types conditionnels (*conditional types*) en les écrivant de la manière suivante [2] :

```
SomeType extends OtherType ? TrueType : FalseType;
```

2. Cf. la documentation à `https://www.typescriptlang.org/docs/handbook/2/conditional-types.html`.

```typescript
interface Animal             { live(): void; }
interface Dog extends Animal { woof(): void; }

type Example1 = Dog extends Animal ? number : string;      // Example1 is equal to number (as types)
type Example2 = RegExp extends Animal ? true : false;      // Example2 is equal to false (as types)
```

Cette construction est au coeur de nombreux programmes Typescript effectuant des calculs sur les types, permettant par exemple d'exécuter n'importe quel code WebAssembly [3].

1. Proposer un type `Equal<X,Y>` censé être égal à `true` si et seulement si les deux types `X` et `Y` sont égaux.

Nous nous proposons de construire, à partir d'une chaîne de caractères représentant une route pour un serveur HTTP, un type de fonction qui permette de réaliser une action sur cette route. La route en question est fournie par une chaîne de caractères, que l'on peut découper en blocs séparés par le caractère `'/'`. Chacun de ces blocs peut être fixe ou variable, et dans le cas où il est variable, il commence par le caractère `':'`. L'expression régulière pour les routes est la suivante : `^(\/:?[a-z]*)+$`. Par exemple :

```typescript
// A route for an HTTP server :
let route:string = "/:posts/for/some/url/:id/where/:user"
                          // where the fixed blocks are "for", "some", "url", and "where"
                          // and the variable blocks are ":posts", ":id", and ":user"

// This route is supposed to match the following urls :
let urls:[string] = [
  "/prog/for/some/url/689/where/renault",          // :user is "renault", :id is "689", :posts is "prog"
  "/cuisine/for/some/url/754/where/dacreguierce", // :user is "dacreguierce", :id is "754", :posts is "cuisine"
]
```

L'idée consiste alors à associer à la route précédente le type de fonction suivant :

```typescript
(_: { posts: string, id: string, user: string }) => void
```

Concrètement, il s'agit d'associer à une route donnée une fonction capable de s'appliquer à n'importe quelle URL adressée au serveur et matchant la route en question (i.e dans laquelle les blocs fixes restent les mêmes, mais les blocs variables peuvent prendre d'autres valeurs).

Le code qui suit contient deux sortes d'expressions :

— des définitions de types (`RouteId`, `RouteParams` ...) ;

— des tests donnant des exemples de leur application (`__1`, `__2` ...).

Les questions suivantes portent principalement sur les deux premières fonctions, mais le code est fourni dans son entièreté.

3. Le code est disponible à `https://github.com/MichiganTypeScript/typescript-types-only-wasm-runtime`.

```ts
type RouteId<Route extends string> =
    Route extends `:${infer Label}`
    ? [ Label ]
    : [ ]

type __1 = [
    Assert<Equal<        RouteId<"posts">,         [ ]            >>,
    Assert<Equal<        RouteId<":posts">,        [ "posts" ]    >>,
]

type RouteParams<Route extends string> =
    Route extends `/${infer Block}/${infer SubRoute}`
    ? [ ...RouteId<Block>, ...RouteParams<`/${SubRoute}`>]
    : Route extends `/${infer Block}`
    ? [ ...RouteId<Block> ]
    : []

type __2 = [
    Assert<Equal<  RouteParams<"/posts/:id/:love">,                    ["id",  "love"]          >>,
    Assert<Equal<  RouteParams<"/posts">,                              []                       >>,
    Assert<Equal<  RouteParams<"/:id">,                                [ "id" ]                 >>,
    Assert<Equal<  RouteParams<"/posts/:id">,                          [ "id" ]                 >>,
    Assert<Equal<  RouteParams<"/posts/:id/:user">,                    ["id", "user"]           >>,
    Assert<Equal<  RouteParams<"/:posts/for/some/url/:id/where/:user">, ["posts", "id", "user"]  >>,
]

type TupleToIndex<Tuple extends any[]> =
    Tuple extends [ infer First,  ...infer Others ]
    ? TupleToIndex<Others> | Others['length']
    : never

type __3 = [
    Assert<Equal<TupleToIndex<[ ]>,                                          never            >>,
    Assert<Equal<TupleToIndex<[ "id", "love" ]>,                             0 | 1            >>,
    Assert<Equal<TupleToIndex<[ "id", "love", "first", "last", "email" ]>,   0 | 1 | 2 | 3 | 4 >>,
]

type RouteFun<Route extends string> = (_: {
    [ K in TupleToIndex<RouteParams<Route>> as RouteParams<Route>[K]]: string }) => void;

type __4 = [
    Assert<Equal<    RouteFun<"/posts/:id/:user">,
                     (_: { id: string, user: string }) => void                  >>,
    Assert<Equal<    RouteFun<"/:posts/for/some/url/:id/where/:user">,
                     (_: { posts: string, id: string, user: string }) => void   >>,
]
```

2. Expliquer l'algorithme sous-jacent au type `RouteParams`.
   Remarque : on ne demande pas d'expliquer le système de type de Typescript, simplement l'algorithme associé à ces 6 lignes de code.
3. Quel style de programmation est-il appliqué ici pour écrire de telles fonctions ?
   A votre avis, pourquoi ?

Les types ainsi construits permettent de faire de la vérification du code.

4. A quel moment de la vie du code cette vérification est-elle faite ?

5. En quoi ce style de programmation est-il intéressant par rapport à des tests classiques ?

✓

1. La bibliothèque `asserttt` (cf. `https://github.com/rauschma/asserttt`) propose l'approximation suivante :

```
type Equal<X, Y> = [X, Y] extends [Y, X] ? true : false ;
```

   Il se trouve en fait que cette approximation est fausse pour le type `any`, mais elle (ou ses variations) était tout à fait acceptable comme réponse à cette question.

2. Le type `RouteParams` est en fait une fonction de types, prenant un paramètre `Route` qui soit sous-type de `string`, et renvoyant un type qui est un tableau de `string`. En Typescript, il faut toujours faire attention à séparer ce qui est type de ce qui est valeur, mais la syntaxe du langage floute cette distinction. Une fois cette précaution prise, la suite de cette réponse se permet un peu d'abus de langage.

   Cette fonction utilise une sous-fonction `RouteId`, qui prend une chaîne de caractères, et renvoie un tableau de chaînes de caractères, qui est non vide si et seulement si la chaîne représente un bloc variable.

   Ensuite, la fonction `RouteParams` applique un algorithme récursif, décomposant son paramètre `Route` en blocs un par un, et appliquant la fonction `RouteId` à chaque bloc, pour construire le tableau final.

3. Le style de programmation utilisé dans ce code est un style de programmation fonctionnel. Cela n'aurait pas de sens d'utiliser un style impératif ici, avec des "variables de type" qui évolueraient au cours du calcul. Le style fonctionnel, avec sa manière de composer les calculs, se prête bien à l'écriture de tels programmes. Techniquement, ce style se retrouve lorsque l'on fait de la méta-programmation en C++.

4. Les vérifications de types faites en Typescript sont faites à la compilation, et sont dont statiques.

5. Comparés aux tests, qui eux sont dynamiques, ces constructions permettent de faire de la vérification sans exécuter le code. C'est une propriété générale de la programmation typée, mais ici poussée assez loin. Les types ainsi calculés sont par exemple utilisables pendant le développement pour vérifier que les paramètres des URLs sont bien utilisés par les fonctions qui gèrent les routes.

Exercice 4: 4 points

Considérons le code suivant [4] écrit en Python pour appliquer des filtres à des bases de données. Dans ce code, le mot-clé `@dataclass` est une facilité pour construire des classes Python en fournissant simplement une liste d'attributs. Le code est annoté avec des types compatibles avec la PEP 484, en particulier le type `Callable[[U1,...UN],R]` correspond au type de fonction classique $(U_1, \ldots, U_N) \to R$.

4. Inspiré d'une question `https://stackoverflow.com/questions/79571051/how-can-i-type-a-method-that-accepts-any-subclass-of-a-base-class` dont le titre ne donne pas d'indice pour résoudre cette question.

```python
@dataclass
class Data:
    pass

@dataclass
class Duck(Data):
    name: str
    age: int

@dataclass
class Mouse(Data):
    name: str
    size: float
```

```python
@dataclass
class Database:
    records: Mapping[str, Sequence[Data]]

    def filter(self,
               table: str,
               filter_fun: Callable[[Data], bool]
               ) -> Sequence[Data]:
        return [ i for i in self.records[table] if filter_fun(i) ]

db = Database({
    "ducks": [
        Duck(name="Patinhas", age=104),
        Duck(name="Donald",   age=30),
        Duck(name="Huguinho", age=12),
        Duck(name="Zezinho",  age=12),
        Duck(name="Luizinho", age=12),
    ],
    "mice": [
        Mouse(name="Mickeyo",   size=7.5),
        Mouse(name="Minniehas", size=8.3),
    ]})

ffun: Callable[[Duck], bool] = lambda t: t.age < 100  # Type error
print(*db.filter("ducks", ffun), sep="\n")
```

Analysé avec le type-checker mypy, le code engendre une unique erreur à la compilation sur l'avant-dernière ligne (à la définition de la variable `ffun`) :

```
error: Argument 2 to "filter" of "Database" has incompatible type "Callable[[Ducks],␣bool]";
       expected "Callable[[Data],␣bool]"  [arg-type]
```

Néanmoins son exécution avec Python n'engendre aucune erreur et renvoie le résultat suivant :

```
Duck(name='Donald', age=30)
Duck(name='Huguinho', age=12)
Duck(name='Zezinho', age=12)
Duck(name='Luizinho', age=12)
```

En Python, pour transtyper (*cast*) une valeur `v` dans le type `T`, il suffit d'écrire `cast(T, v)`.

1. Rappeler les propriétés de variance du type `Callable[[U1, ..., UN], R]` sur chacun de ses paramètres.
2. Expliquer d'où vient l'erreur de type.
3. Proposer une manière de la corriger.
   Note : la lambda-fonction stockée dans `ffun` utilise l'attribut `age` de la classe `Duck`.

Le type `Mapping` est défini à travers une classe abstraite de la bibliothèque `collections.abc` (qui signifie *Abstract Base Classes for Containers*). Dans la page de documentation [5] apparaissent les quelques lignes suivantes :

ABC	Inherits from	Abstract Methods	Mixin Methods
Mapping	Collection	`__getitem__`, `__iter__`, `__len__`	`__contains__`, `keys`, `items`, `values`, `get`, `__eq__`, `__ne__`
MutableMapping	Mapping	`__getitem__`, `__setitem__`, `__delitem__`, `__iter__`, `__len__`	Inherited Mapping methods and `pop`, `popitem`, `clear`, `update`, and `setdefault`

4. A partir de ces informations, quelles indications de variance estimez-vous que l'on peut associer au type `Mapping[Key, Value]` ? Justifier.

5. Que l'on peut retrouver à l'adresse `https://docs.python.org/3/library/collections.abc.html`.

✓

1. Le type `Callable[[U1,..., UN], R]` est le type classique de fonction, et il est donc *contravariant* sur chacun de ses types de paramètres d'entrée `Ui` et *covariant* sur son type de retour `R`.
2. Manifestement, il y a une incompatibilité entre le type de la variable `ffun` et le type du 2nd paramètre de la méthode `filter` de la classe `Database`. Le type de `ffun` n'est pas un sous-type du type attendu par la méthode. Le problème est dû à la contravariance de `Callable` en son premier paramètre. En effet, Duck <: Data, ce qui fait que Data→ Bool <: Duck→ Bool, l'inverse de ce qui est attendu ici.
3. La solution la plus simple consiste à modifier le type de `ffun`, par exemple comme `Callable[[Data],bool]`. Mais ce n'est pas complètement immédiat. En effet, le code de `ffun` utilise l'attribut `age` qui n'apparaît pas dans le type `Data`. Il faut donc transtyper le paramètre `t` de la manière suivante :

```
ffun: Callable[[TableData], bool] = lambda t: (cast(Duck, t)).age < 100
```

4. Manifestement, il existe deux types de `Mapping` dans la bibliothèque `collections.abc`, l'un d'entre eux étant mutable et pas l'autre. La classe `Mapping` est immutable (pas de méthode `set` ou `del`). Les clés et les valeurs sont en position d'accesseurs pour cette classe. Les règles de variance permettent donc qu'elle soit *covariante* pour chacun de ses deux paramètres.

   Noter qu'on pourrait être troublé par le fait que `MutableMapping` hérite de `Mapping`, et le fait que `MutableMapping` doive, lui, être invariant pour ses paramètres. Mais en dessinant les relations de types/sous-types, on se rend compte que cela ne pose pas de problèmes.

Exercice 5: 3 points

Expliquer en quoi, en Haskell, les classes de types participent d'au moins deux formes de polymorphisme différents (parmi celles vues en cours).

Pour rappel, voici un extrait de la définition de la classe de type `Eq a` :

```
class Eq a where
    (==), (/=)          :: a → a → Bool
    x /= y              = not (x == y)
    x == y              = not (x /= y)
```

```
instance (Eq a) ⇒ Eq [a] where
    []     == []     = True
    (x:xs) == (y:ys) = x == y && xs == ys
    _xs    == _ys    = False
```

✓ Les classes de type Haskell, comme `Eq a`, sont une forme de *polymorphisme paramétrique*, comme l'indique la variable de type `a` dans la définition du type. Dans cet exemple, les fonctions `==` et `/=` ont un type paramétrique.
Mais ces constructions représentent aussi une forme de *polymorphisme de surcharge*, puisqu'il est possible de fournir plusieurs implémentations des fonctions de la classe de type, en général une pour chaque type dans la classe.
Enfin, comme toutes les instances partagent la même interface, il s'agit aussi d'une forme de *polymorphisme d'inclusion*. Mais la justification devait parler de la notion d'interface (ou équivalent), parce que les types d'une classe de type ne partagent pas forcément des relations de sous-typage entre eux.