Année 2021-2022 Session 1

Systèmes de Types et Programmation
IT234

Filière : Informatique, 2ème Année

Date de l'examen : 23/05/2022 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

Expliquer à quoi correspond la notion de *sûreté de typage*.
En quoi cette notion est importante pour un langage de programmation ?

✓ La sûreté de typage est la combinaison de deux propriétés des systèmes de types : la *préservation* (i.e le fait que l'évaluation d'une expression préserve son type apparent à chaque étape de réduction) et le *progrès* (i.e le fait que toute expression typable qui n'est pas une valeur puisse continuer à être évaluée correctement). Si ces propriétés sont vérifiées par le système de types d'un langage, elles assurent qu'un programme correctement typé va s'évaluer, tout en préservant les types de l'ensemble de ses composants. Au final, il s'agit d'une manière relativement simple de formaliser le genre d'invariants assurés par un système de type. En particulier, une variable d'un type donné (dnc son type apparent) ne change pas de type apparent au cours de l'exécution.

Entre parenthèses, la définition de la préservation est rendue complexe dans le cas de sous-typage. Les types concrets peuvent être des sous-types des types apparents, mais il faut se rappeler que les règles de typage (ce sur quoi le compilateur s'appuie pour vérifier) se font sur les types apparents.

Exercice 2: 2 points

1. Exhiber une fonction OCaml de type : `(int → 'a) → 'a`.
2. Exhiber une fonction OCaml de type : `'a → ('a → 'b) → 'b`.

✓
1. La fonction `let f g = g 42` répond à la question.
2. La fonction `let f a b = b a` répond à la question.

Exercice 3: 2 points

Inférer le prototype le plus générique possible pour la fonction suivante `type_me` en Java, et proposer un morceau de code réalisant une application correcte de cette fonction.

```
______ type_me( ______ f, ______ x, ______ y) {
    if (f.apply(x, y))
        return x;
    else
        return y;
}
```

Les symboles "_____" indiquent les endroits où des types ont été effacés.

✓ Le code impose que les types des deux branches du `if` soient les mêmes. Et donc Le prototype `static <A> A type_me(BiFunction<A,A,Boolean> f, A x, A y)` compile correctement en Java. Il devient alors possible de l'appliquer avec des lambdas, par exemple sur des chaînes de caractères : `type_me((u,v) → u.charAt(0) != v.charAt(0), "ab", "cd")` renvoie `"ab"`. Si l'on ne connaît pas l'interface fonctionnelle `BiFunction`, il est toujours possible de proposer une interface dans le style suivant :

```
interface Appliable<T> {
    boolean apply(T x, T y);
}
static <A> A type_me(Appliable<A> f, A x, A y)
```

Les propositions à base de `Object` sont forcément moins précises du point de vue des types (même si elles sont génériques).

Exercice 4: 3 points

Le code suivant [1], en C#, a pour finalité de gérer des fichiers de types différents avec des handlers différents. La fonction `GetFileManager` associe à chaque type de fichier le handler qui lui convient.

```
// A generic interface for files and a particular implementation
public interface IFile { }
public class NetworkFile : IFile { }

// A generic interface to handle files and a particular implementation
public interface IFileManager<T> where T : IFile {
    void SaveFile(T file);
}
public class NetworkFileManager : IFileManager<NetworkFile> {
    public void SaveFile(NetworkFile file) {
        Console.WriteLine("Saving file on the network");
    }
}

// A factory returning a handler for a give file type
public class FileManagerFactory {
    public static IFileManager<IFile> GetFileManager(IFile file) {
        if (file is NetworkFile)
            return (IFileManager<IFile>) new NetworkFileManager(); // Runtime error
        else
            throw new Exception("File type not handled");
    }

    public static void Main() {
      FileManagerFactory.GetFileManager(new NetworkFile());
    }
}
```

Ce code compile sans erreurs, mais engendre une erreur à l'exécution.

1. Quelle est plus spécifiquement l'erreur qui est générée ?
2. Pourquoi cette erreur est-elle générée ?
3. Est-ce qu'il est possible de la corriger en préservant l'idée générale du code ?

1. Inspiré d'une question https://stackoverflow.com/questions/27061361/downcasting-to-generic-type, dont le titre ne donne pas d'information intéressante pour résoudre cette question.

✓

1. L'erreur est provoquée sur une ligne réalisant un transtypage / cast. La seule manière de provoquer une telle erreur tient dans l'impossibilité de ladite conversion de type, une erreur appelée `InvalidCastException` en C#. Il est impossible de convertir un `NetworkFileManager` en `IFileManager<IFile>`.
2. Pour que la conversion soit acceptable, il faut en particulier la relation `IFileManager<NetworkFile>` <: `IFileManager<IFile>` (sachant que `NetworkFile` <: `IFile`). Sans modification du code, cela demanderait à ce que `IFileManager<T>` soit covariant en son paramètre `T`. Or en C#, les generics sont par défaut invariants.
3. Pour corriger la chose, il faudrait donc rendre le type `IFileManager<T>` covariant en `T`. C'est néanmoins impossible, car le type `T` est en position *contravariante* dans l'interface de `IFileManager<T>` : il sert en tant que paramètre à la fonction `SaveFile`. La seule possibilité pour ce paramètre est donc d'être invariant.

   Autre explication en reprenant les principes du get-put principle : les instances de `T` sont consommées (*supplied*) par les instances de `IFileManager<T>`, et ne peuvent pas varier de manière covariante.

   Néanmoins, il est possible de corriger ce code si on allège les contraintes de type. Un de nos problèmes ici est que le sous-typage autorise `GetFileManager` à prendre en paramètre n'importe quel sous-type de `IFile` (d'où le problème de variance). Si on enlève cette contrainte, le code suivant est possible :

```
public class GenericFile<T> where T : IFile
public static IFileManager<T> GetFileManager<T> (GenericFile<T> file) where T : IFile {
    if (file is GenericFile<NetworkFile>)
        return (IFileManager<T>) new NetworkFileManager();
    else
        throw new Exception("File type not handled");
}
```

   (ici, le type `GenericFile<T>` est invariant en `T`).

Exercice 5: 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 plusieurs types abstraits de données distincts. L'exemple doit permettre de gérer des bibliothèques [2] accueillant des usagers pour emprunter et remettre des livres.

2. L'établissement, pas le meuble.

✓ 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.

```
type date (* Public type probably existing in the standard library *)

module L : sig (* Interface *)
  type library
  val available_books : library → book list
  val overdue_books_for_user : library → user → book list
  val rent_book : user → book → library → date
  val return_book : user → book → library → unit
end = struct   (* Implementation example *)
  type library = { contents : book list, due_dates : (user * book * date) list }
  (* ... *)
end

module U : sig (* Interface *)
  type user
  val get_name : user → string
end = struct   (* Implementation example *)
  type user = { name : string }
  (* ... *)
end

module B : sig (* Interface *)
  type book
  val get_title : book → string
end = struct   (* Implementation example *)
  type book = { title : string }
  (* ... *)
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 6: 4 points

Le code suivant[3] a été écrit à la fois en C++ (à gauche) et en Julia (à droite), et met en oeuvre divers mécanismes de polymorphisme. En dessous de chacun des codes, il y a le résultat de leur exécution.

```cpp
// C++ code
struct Pet {
  string name;
  Pet(string n) : name(n) {};
};

struct Dog: public Pet { Dog(string n) : Pet(n) {}; };
struct Cat: public Pet { Cat(string n) : Pet(n) {}; };

string meets(Pet a, Pet b) { return "UNDEF"; }
string meets(Dog a, Dog b) { return "sniffs"; }
string meets(Dog a, Cat b) { return "chases"; }
string meets(Cat a, Dog b) { return "hisses"; }
string meets(Cat a, Cat b) { return "slinks"; }

void encounter(Pet a, Pet b) {
  string verb = meets(a, b);
  cout << a.name << "␣meets␣" << b.name <<
    "␣and␣" << verb << endl;
}

int main(void) {
  Dog fido("Fido");         Dog rex("Rex");
  Cat whiskers("Whiskers"); Cat spots("Spots");

  encounter(fido, rex);
  encounter(fido, whiskers);
  encounter(spots, rex);
  encounter(whiskers, spots);
}
```

```
Fido meets Rex and UNDEF
Fido meets Whiskers and UNDEF
Spots meets Rex and UNDEF
Whiskers meets Spots and UNDEF
```

```julia
# Julia code
abstract type Pet end



struct Dog <: Pet; name :: String end
struct Cat <: Pet; name :: String end

meets(a::Pet, b::Pet) = "UNDEF"
meets(a::Dog, b::Dog) = "sniffs"
meets(a::Dog, b::Cat) = "chases"
meets(a::Cat, b::Dog) = "hisses"
meets(a::Cat, b::Cat) = "slinks"

function encounter(a::Pet, b::Pet)
    verb = meets(a, b)
    println("$(a.name)␣meets␣$(b.name)␣and␣$verb")
end


function main()
    fido     = Dog("Fido");     rex   = Dog("Rex")
    whiskers = Cat("Whiskers"); spots = Cat("Spots")

    encounter(fido, rex)
    encounter(fido, whiskers)
    encounter(spots, rex)
    encounter(whiskers, spots)
end
```

```
Fido meets Rex and sniffs
Fido meets Whiskers and chases
Spots meets Rex and hisses
Whiskers meets Spots and slinks
```

1. Quel est le polymorphisme en jeu dans le code C++ ?
   Expliquer le comportement obtenu.
2. Expliquer le comportement obtenu dans le code Julia (comparé au code C++).
   Quel est le polymorphisme en jeu ?

3. Tiré d'une vidéo de S. Karpinski à la conférence JuliaCon 2019 (https://www.youtube.com/watch?v=kc9HwsxE1OY), vidéo dont le titre donne vraiment trop d'informations pour être raisonnablement recopié ici.

✓

1. Le code en C++ illustre le *polymorphisme de surcharge* (dans la fonction `meets`) et (théoriquement) le *polymorphisme de redéfinition* (dans la fonction `encounter`). La fonction `meets` est surchargée, avec des codes différents pour tous les couples de types de paramètres possibles. Dans ce cas, le polymorphisme de surcharge est résolu à la compilation, en fonction des *types apparents* de ses paramètres. Ainsi, l'utilisation de la fonction `meets` au sein de la fonction `encounter` s'applique à des valeurs de type `Pet`. Le résultat de cette fonction renvoie donc systématiquement `"UNDEF"`, et le polymorphisme de redéfinition n'est en fait pas utilisé ici.
2. Le code en Julia ne souffre pas du travers du code en C++. Cela signifie que la fonction `meets` est sélectionnée parmi les implémentations possibles, et ici manifestement en fonction des *types concrets* de ses paramètres. Il y a donc un mécanisme de dispatch (dynamique) mis en place. Manifestement, le runtime est capable de sélectionner la méthode en fonction des deux paramètres, il s'agit donc de *multiple dispatch*.

Exercice 7: 2 points
Le code suivant écrit en Rust permet de gérer les surfaces de différentes formes géométriques :

```rust
trait Measurable {
    fn area(&self) → f64;
}

struct Rectangle { length: f64, width: f64 }
struct Circle    { radius: f64 }

impl Measurable for Rectangle { fn area(&self) → f64 { self.length * self.width } }
impl Measurable for Circle {    fn area(&self) → f64 { self.radius * self.radius * 3.14 } }

fn display_area<T: Measurable>(t: &T) → () { println!("It␣measures␣:␣{}", t.area()); }

fn main() {
    let rectangle = Rectangle { length: 3.0, width: 4.0 };
    let circle    = Circle    { radius: 3.0 };

    display_area(&rectangle); // It measures : 12
    display_area(&circle);    // It measures : 28.26
}
```

Donner le nom du polymorphisme à l'oeuvre dans le type de la fonction `display_area`.
Relier ce polymorphisme à d'autres constructions du même style vues en cours.

✓ Le trait `Measurable` définit une interface pour un type générique `T`. La notation `T: Measurable` permet d'associer une contrainte sur la variable de type `T` utilisée dans la fonction `display_area`. Il s'agit donc de *polymorphisme paramétrique contraint*. Le cours donne des exemples en Sml (*equality types*), en Haskell (*type classes*) et en Java (*interfaces génériques*).

Exercice 8: 4 points

La fonction `sprintf` est une fonction appartenant à la bibliothèque standard C, permettant de formater des chaînes de caractères [4] comme le montrent (s'il en est besoin) les exemples suivants compilés avec gcc :

```
char output[80];
sprintf(output, "%s", "This ");                 // → "This "
sprintf(output, "%d", 3);                       // → "3"
sprintf(output, "This %d is %c", 3, '3');       // → "This 3 is 3"
sprintf(output, "This %d is %s", 3, '3');       // Warning : format '%s' expects argument of type 'char *',
                                                // but argument has type 'int'
```

Le dernier exemple produit un avertissement à la compilation et une erreur de segmentation à l'exécution. Pour les besoins de cet exercice, on considère que cet avertissement est une erreur de type.

1. A votre avis, est-ce qu'il s'agit ici d'un polymorphisme de surcharge ?

Afin de générer des erreurs de types statiques, on s'intéresse à la technique suivante en OCaml, utilisant certains mécanismes du langage n'existant pas en C. Le code suivant montre la nouvelle définition de la fonction `sprintf`, ainsi que quelques exemples d'utilisation [5] :

```
type (_,_) print =
  | Fstr : string → ('a, 'a) print
  | Fint : ('a, (int → 'a)) print
  | Fchr : ('a, (char → 'a)) print
  | Comp : (('b, 'c) print * ('a, 'b) print) → ('a, 'c) print

let rec interprete : type a b. (a, b) print → (string → a) → b =
  fun p k → match p with
  | Fstr str   → k str
  | Fint       → (fun x → k (string_of_int x))
  | Fchr       → (fun x → k (Char.escaped x))
  | Comp (a,b) → interprete a (fun sa → interprete b (fun sb → k (sa^sb)));;

let sprintf : type a. (string, a) print → a = fun fmt → interprete fmt (fun x → x);;
```

```
let (^^) x y = Comp (x,y);;   (* infix shorthand for 'Comp' *)

sprintf  (Fstr "This ");;                                  (* string = "This " *)
sprintf ((Fstr "This ") ^^ Fint);;                         (* int → string = <fun> *)
sprintf ((Fstr "This ") ^^ Fint ^^ (Fstr " is ") ^^ Fchr);;       (* int → char → string = <fun> *)
sprintf ((Fstr "This ") ^^ Fint ^^ (Fstr " is ") ^^ Fchr) 3 '3';; (* string = "This 3 is 3" *)
```

2. Quelle est la construction reconnaissable du langage OCaml utilisée ici ?
   A quoi la reconnaît-on ?
3. Expliquer en quelques mots la relation entre le premier paramètre de la fonction `sprintf` et le reste de ses paramètres (il n'est pas demandé d'expliquer le type `('a,'b) print`)

4. Avec une histoire étendue, comme on peut le lire sur la page https://en.wikipedia.org/wiki/Printf.
5. Code simplifié à partir d'une version accessible sur la page https://okmij.org/ftp/ML/GADT.ml.

4. Comment appele t'on un type (de fonction) dans lequel les types de certains paramètres sont contraints par les types des paramètres précédents ?

✓

1. Au vu des exemples, les deux dernières utilisations de `sprintf` sont indistinguables du point de vue des types, et l'un deux provoque un avertissement / erreur de type et pas l'autre. Il *ne peut donc pas* s'agir d'un simple polymorphisme de surcharge.
   Pour référence, le type de la fonction `sprintf` en C++ est le suivant :

   ```
   int sprintf(char *str, const char *format, ...);
   ```

   Il n'était pas demandé de connaître ce type par coeur pour répondre à la question.
2. Le type `('a, 'b) print` est un exemple de type abstrait de données généralisé, aussi appelé GADT. Il est reconnaissable à la fois à l'utilisation de types fantômes et au fait que ces types fantômes sont contraints selon les constructeurs utilisés (`Fstr`, `Comp` ...).
   Noter que le code utilise un style de passage par continuation pour l'implémentation de la fonction `interprete` (le paramètre `k`). Mais il ne s'agit pas d'une construction du langage, simplement un style de programmation.
3. Le premier paramètre de la fonction `sprintf` est une chaîne de formatage. En C++, cette chaîne est de type `char*`, mais dans cette implémentation en OCaml, on lui donne un type particulier appelé `(string, 'a) print`. Si on applique la fonction à ce premier paramètre, les exemples montrent que le type résultant est différent selon la forme du premier paramètre.
   - soit ce premier paramètre est simplement un `Fstr`, auquel cas la fonction renvoie directement une `string` ;
   - soit ce premier paramètre contient des occurrences de `Fint` (resp. `Fchr`), auquel cas la fonction prend autant de paramètres supplémentaires que d'occurrences, de types `int` (resp. `char`), avant de renvoyer une `string`.

   Grosso modo, le résultat est une fonction demandant à prendre tous les paramètres nécessaires pour construire la chaîne de caractères finale.
4. Un tel type est appelé un *type dépendant*. Le cours fournit un exemple avec des listes dont le type contient la longueur de la liste. Les types associés montrent de nombreux exemples où les types des paramètres et résultats sont contraints par cet entier.