Année 2020-2021 Session 1

Systèmes de Types et Programmation
IT234

Filière : Informatique, 2ème Année

Date de l'examen : 25/05/2021 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 barème est purement indicatif.

Exercice 1: 2 points

Expliquer la phrase suivante tirée du cours : "un type représente une approximation de l'évaluation d'une expression".

✓ L'ensemble des exécutions (ou des évaluations) d'un programme est un ensemble complexe, sur lequel il n'est pas possible de dire grand chose du fait de l'indécidable (cf. le théorème de Rice). Les vérifications dynamiques comme les tests s'attachent à vérifier des propriétés des sous-ensembles *finis* de ces exécutions. A contrario, les systèmes de types proposent de vérifier des propriétés sur ces exécutions d'une manière statique, *sans* les exécuter.
Pour réaliser cela, ils introduisent la notion de type d'une expression qui, sans évaluer l'expression, représentent une approximation de ce qui peut être le résultat de cette évaluation (sous la forme d'un ensemble de valeurs possibles). Les vérifications faites par les systèmes de types consistent principalement à assurer la cohérence de la composition de ces approximations (i.e qu'on passe bien un entier à une fonction qui attend un entier).
En considérant uniquement des approximations, on peut au final résoudre des problèmes décidables sur les programmes.

Exercice 2: 1 point

Inférer le type de l'expression suivante : $\lambda f.\lambda x.\lambda y.\big($`f x - f y`$\big)$ (aucun arbre de dérivation n'est demandé, la fonction "-" est l'opérateur infixe de soustraction sur les entiers)

✓ En OCaml, `fun f x y → (f x - f y)` renvoie `('a → int) → 'a → 'a → int`.

Exercice 3: 2 points

La nouvelle norme C++ 20 introduit dans le langage une nouvelle construction appelée *concepts*, qui permet de faire une vérification particulière sur les paramètres de type passés aux templates :

```cpp
// "Measurable" concept
template<typename T>
concept Measurable = requires(const T& t) {
  { t.area() } -> std::convertible_to<double>;
};

struct Square { double area() const { return 1; } };
struct Banana { void eat()    const { std::cout << "Yum" << std::endl; } };

// Constrained C++20 function template:
template<typename T>
requires Measurable<T>
void display_area(const T& t) { std::cout << t.area() << std::endl; }

int main() {
  Square r;
  display_area(r); // Outputs ''1''
  // Banana b;
  // display_area(b); // Unsatisfied constraints
}
```

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.

✓ `Measurable<T>` permet ici d'associer une contrainte (**`requires`**) sur la variable de type `T` paramétrant le template englobant 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 4: 4 points

Lors de la programmation d'un jeu de cartes, un programmeur Python a construit une fonction `choose_game_card` permettant de choisir une carte (appartenant à la classe `Card`) aléatoirement parmi un ensemble de cartes possibles, et autorisant le joueur à ne pas jouer en renvoyant `None`. Le pseudo-code de la fonction est le suivant :

```python
def choose_game_card(self, cards, maybeEmpty):
    allowedCards = cards
    if maybeEmpty:
        allowedCards.append(None)       # append concatenates its parameter at the end of the list
    card = random.choice(allowedCards) # choice returns a random element of its parameter
    return card
```

Un autre programmeur passant par là et un peu zélé propose d'ajouter des annotations de type à ce code. En particulier, il propose les annotations suivantes :

```python
def choose_game_card(self, cards : List[Card], maybeEmpty : bool) -> Optional[Card]
    allowedCards : List[Optional[Card]] = cards
    # ... rest of the code unchanged ...
```

Le type-checker de Python se rebelle contre cette annotation, renvoyant le message suivant :

```
Incompatible types in assignment (expression has type "List[Card]", variable has type "List[Optional[Card]]")
note: "List" is invariant -- see http://mypy.readthedocs.io/en/latest/common_issues.html#variance
note: Consider using "Sequence" instead, which is covariant
```

1. Expliquer ce que représente selon vous le type `Optional[T]`.
2. Expliquer le contenu du message d'erreur et justifier pourquoi le compilateur s'oppose à un tel typage.
3. Pour que le conseil proposé par le compilateur consistant à remplacer `List` par `Sequence` soit valide, il est nécessaire d'avoir une relation type/sous-type impliquant `Optional[T]`. Laquelle ?

✓

1. Le type `Optional[T]` est ici utilisé pour représenter des valeurs étant possiblement soit de type `T`, soit égales à `None`. La documentation le décrit explicitement comme `Union[T, None]`. Il est donc adapté pour gérer une liste contenant soit des instances de `Card`, soit la valeur `None`.
2. Dans ce contexte, la variable `allowedCards` a le type `List[Optional[Card]]`, et on lui affecte une valeur de type `List[Card]`. L'affectation en question serait bien typée si (et seulement si) le second type était sous-type du premier.
   Le message indique que le type `List[T]` en Python est invariant, et donc il n'y a pas de relation de type/sous-type entre ces deux types. L'affectation est donc mal typée. Selon le message du compilateur, l'utilisation de `Sequence[T]` à la place de `List[T]` peut autoriser des relations type/sous-type parce qu'il est covariant en `T`.
3. Pour que la relation évoquée précédemment soit valide en utilisant `Sequence` et sa covariance, il faut supposer que $\mathsf{T} <: \mathsf{Optional}[\mathsf{T}]$ (ce qui est valide en Python typé avec le compilateur `mypy`).

Exercice 5: 4 points

Une construction classique dans les systèmes de types consiste à permettre de construire des *types somme*, capable de représenter des valeurs de plusieurs types différents. Par exemple, en OCaml, un tel type pourrait s'utiliser de la manière suivante afin à la fois d'effectuer des calculs sur des flottants, et de transmettre des messages d'erreur :

```
type ('a,'b) sum = Left of 'a | Right of 'b;;

let safe_sqrt x = match x with
| Left n  → if (n >= 0.)
            then Left (sqrt n)
            else Right "Cannot␣take␣the␣square␣root␣of␣a␣negative␣number"
| Right s → Right s;;
(* val safe_sqrt : (float, string) sum → (float, string) sum = <fun> *)

safe_sqrt (Left 4.);;               (* → Left 2. *)
safe_sqrt (Left (-2.));;            (* → Right "Cannot take .." *)
safe_sqrt (Right "Not␣a␣number");;  (* → Right "Not a number" *)
```

En partant du λ-calcul simplement typé décrit en cours, on propose les ajouts suivants à la grammaire des expressions :

Syntax and Types

`t` ::=	...	*expressions*
	`Left(t)`	*left expression*
	`Right(t)`	*right expression*
	`match t with` { `Left(x)` $\to$ `t`; `Right(y)` $\to$ `t` }	*pattern-matching*
`v` ::=	...	*values*
	`Left(v)`	*left value*
	`Right(v)`	*right value*
T ::=	...	*types*
	Sum[T, T]	*sum type*

Décrire les ajouts nécessaires au langage pour ajouter de tels *types somme* et permettre d'évaluer et de typer le calcul présenté ci-dessus.

✓ Un ensemble de modifications pour gérer les types somme, aussi décrite dans le chapitre 11.9 du Pierce, pourrait prendre la forme :

Evaluation rules

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

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

$$\frac{\texttt{t} \to_\beta \texttt{t'}}{\texttt{match t with} \begin{cases} \texttt{Left(x)} \to \texttt{u} \\ \texttt{Right(y)} \to \texttt{v} \end{cases} \to_\beta \texttt{match t' with} \begin{cases} \texttt{Left(x)} \to \texttt{u} \\ \texttt{Right(y)} \to \texttt{v} \end{cases}}$$

$$\texttt{match Left(z) with} \begin{cases} \texttt{Left(x)} \to \texttt{u} \\ \texttt{Right(y)} \to \texttt{v} \end{cases} \to_\beta [\texttt{x} \mapsto \texttt{z}]\texttt{u}$$

$$\texttt{match Right(z) with} \begin{cases} \texttt{Left(x)} \to \texttt{u} \\ \texttt{Right(y)} \to \texttt{v} \end{cases} \to_\beta [\texttt{y} \mapsto \texttt{z}]\texttt{v}$$

Typing rules

$$\frac{\Gamma \vdash \texttt{t} : \mathsf{T}}{\Gamma \vdash \texttt{Left(t)} : \mathsf{Sum}[\mathsf{T}, \mathsf{U}]}$$

$$\frac{\Gamma \vdash \texttt{t} : \mathsf{U}}{\Gamma \vdash \texttt{Right(t)} : \mathsf{Sum}[\mathsf{T}, \mathsf{U}]}$$

$$\frac{\Gamma \vdash \texttt{t}_1 : \mathsf{Sum}[\mathsf{T}, \mathsf{U}] \qquad \Gamma, \texttt{x} : \mathsf{T} \vdash \texttt{t}_2 : \mathsf{V} \qquad \Gamma, \texttt{y} : \mathsf{U} \vdash \texttt{t}_3 : \mathsf{V}}{\Gamma \vdash \texttt{match t}_1 \texttt{ with} \begin{cases} \texttt{Left(x)} \to \texttt{t}_2 \\ \texttt{Right(y)} \to \texttt{t}_3 \end{cases} : \mathsf{V}}$$

Exercice 6: 3 points

La version 15 du langage Java introduit dans la JEP 360 la notion de *sealed classes* (que l'on pourrait traduire en français par "scellées"). Le principe de telles classes consiste à pouvoir contrôler dans une classe donnée quelles sont les classes qui pourraient en dériver. La documentation fournit les exemples suivants :

```java
sealed class Shape
    permits Circle, Square, Rectangle { }

final class Circle extends Shape {
    public float radius; }

non-sealed class Square extends Shape {
   public double side; }

sealed class Rectangle extends Shape permits FilledRectangle {
    public double length, width; }

final class FilledRectangle extends Rectangle {
    public int red, green, blue; }
```

Les classes dérivant d'une classe scellée doivent à la fois faire partie des classes autorisées par `permits` mais aussi être étiquetées par l'un des mots-clés `final`, `sealed` ou `non-sealed`.

Plaçons nous dans un contexte où le mot-clé `non-sealed` n'est pas utilisé.

1. A quelle autre construction de type vue en cours une telle hiérarchie de classes complètement fermée ressemble t'elle ?
2. Quels avantages le compilateur pourrait-il tirer de cette construction en terme de sélection de méthode sur les instances de `Shape` ?

✓

1. Une hiérarchie de classe complètement fermée, c'est un type somme, à savoir une simplification d'un type algébrique ou variant (sans la récursivité donc). La construction précédente se traduirait en OCaml par :

   ```ocaml
   type shape =
     | Circle of float
     | Square of float
     | Rectangle of float * float
   ```

   Une telle construction permet d'envisager de faire de la reconnaissance de motifs sur les valeurs de type `Shape` (d'ailleur un des buts annoncés de la JEP 360). Noter que la construction Java construit une famille de types différents les uns des autres mais partageant des relations de sous-typage, alors qu'en OCaml, il n'y a qu'un seul type.
2. Si l'arborescence de classe est complètement verrouillée, alors il est envisageable de laisser les algorithmes de sélection de méthode compiler du code purement statique, plutôt que de faire du *dynamic dispatch*.

Exercice 7: 5 points

Le procédé de *curryfication* est une opération consistant à transformer une fonction non-curryfiée (en anglais *uncurried*) prenant plusieurs paramètres en une fonction curryfiée (en anglais *curried*) réalisant le même calcul, mais prenant ses paramètres un par un et renvoyant une nouvelle fonction tant que le calcul n'est pas terminé. L'exemple suivant est écrit en Typescript :

```typescript
function uncurried (name: string, age: number) : boolean {
    console.log(name, age);
    return true;
};

uncurried('Jane', 26);  // → true, logs "Jane 26"

function curried(name: string) {
    return function (age: number) : boolean {
        console.log(name, age);
        return true;
    };
}

curried('Jane');        // → [Function (anonymous)]
curried('Jane')(26);    // → true, logs "Jane 26"
```

Pour les deux questions suivantes, on ne considère que les fonctions à 2 paramètres.

1. Écrire le type d'une fonction `currify` telle que `currify(uncurried) === curried`.
2. Proposer une généralisation de ce type utilisant le polymorphisme paramétrique.

Un programmeur nommé P.A Mills a proposé de générer automatiquement les types des fonctions précédentes ayant un nombre quelconque de paramètres en Typescript [1]. Il propose les constructions suivantes :

```typescript
// Returns a type telling if an array is empty or not
type HasTail<T extends any[]> =
    T extends ([] | [any]) ? false : true

// Returns the type of the first element of an array
type Head<T extends any[]> =
    T extends [any, ...any[]]
    ? T[0] : never

// Returns the type of the tail on an array
type Tail<T extends any[]> =
    ((...t: T) => any) extends ((_: any, ...tail: infer TT) => any)
    ? TT : never

// Type of a function transforming an uncurried one into a curried one
type Curry<P extends any[], R> =
    (arg0: Head<P>) =>
        (HasTail<P> extends true
             ? Curry<Tail<P>, R> : R)

// The currify function has type Curry<P, R>
function currify<P extends any[], R>(f: (...args: P) => R): Curry<P, R> { /* ... */ }

// Examples
function uncurried (name: string, age: number) { return true; }
const curried = currify(uncurried);    // Curry<[string, number], boolean>
const test1   = curried('Jane')(26)    // Type-checks OK
const test2   = curried('Jane', 26)    // Type error, expected 1 arguments, but got 2.
```

1. L'article en question est accessible sur son blog à l'adresse `https://medium.com/free-code-camp/typescript-curry-ramda-types-f747e99744ab` et gère même des formes plus générales de curryfication.

Le code précédent est considéré valide par le compilateur Typescript si l'on omet la toute dernière ligne engendrant une erreur de type.

3. Quelle est la nature de `Head`, `Tail` et `HasTail` ?

Le calcul de `Curry<P, R>` suit un algorithme particulier apparaissant aux lignes 16 à 19.

4. Quel est à votre avis le rôle du mot-clé `infer` (cf. l.12) dans cet algorithme ?
5. A quel phase de la vie du programme l'algorithme précédent est-il exécuté ? Quels types de risques un tel calcul induit-il ?

✓

1. Le type de la fonction `currify` est entièrement spécifié par on entrée et sa sortie, et ceux-ci sont fournis dans le code :

$$\texttt{currify} : \texttt{((string, number) → boolean) → (string → number → boolean)}$$

2. Généraliser le type précédent est très simple, surtout dans un cadre limité aux fonctions à deux paramètres :

$$\texttt{currify} : \forall\ \texttt{T U V, ((T, U) → V) → (T → U → V)}$$

3. Le code Typescript fourni peut sembler effrayant, mais le formalisme utilisé est classique. Les noms des fonctions `HasTail`, `Head` et `Tail` sont des noms fonctions de manipulation de listes. Au vu de le syntaxe avec chevrons `<>` et du mot-clé `type`, ce sont des *fonctions sur les types*. L'algorithme de calcul de $\mathsf{Curry} < \mathsf{P}, \mathsf{V} >$ est alors une simple application d'une fonction récursive sur des listes (de types).
4. Le mot-clé `infer` doit faire penser à la notion d'*inférence de type*. Ce mot-clé est utilisé dans la fonction `Tail`, qui calcule la queue d'une liste de types. Ce type est indiqué comme s'appelant `TT`. Le mot-clé `infer` doit donc indiquer au compilateur qu'il faut *inférer* ce type. La ligne 12 offre de faire cette inférence en faisant du pattern-matching entre `(...t: T) ⇒ any` d'une part et `(_: any, ...tail: TT) ⇒ any` d'autre part.
5. Le code fourni fait manifestement des calculs sur les types. Il est donc voué à être vérifié statiquement à la *phase de compilation*. Un tel exemple montre qu'il est possible en Typescript de faire exécuter des algorithmes sur les listes au compilateur. Un risque classique de tels algorithmes est leur non-terminaison, et donc la possibilité de faire créer une boucle infinie au compilateur. A titre de complément, le compilateur Typescript est capable de lever une erreur ("Type instantiation is excessively deep and possibly infinite") dans ce genre de cas.