ENS Paris-Saclay 2021-2022

Programmation avancée

Examen

Tous les documents sur papier sont autorisés, dans la limite du raisonnable (50 pages maximum). L'usage d'un dispositif électronique est interdit.

La durée de l'épreuve est de 3 heures. Il n'est pas nécessaire de traiter l'intégralité du sujet pour avoir une bonne note.

Les parties sont indépendantes. Au sein d'une même partie, il est préférable de traiter les questions dans l'ordre, mais chaque question est indépendante.

1 Closure conversion et types existentiels

On considère un petit langage, où les valeurs sont des entiers ou tuples d'entiers, et où les expressions sont de la forme suivante :

e	::=	n	constante entière
	|	x	variable
	|	$e \oplus e$	opération binaire (addition, soustraction)
	|	`ifzero` e `then` e `else` e	test à zéro
	|	`fun` x `->` e	fonction anonyme
	|	`funrec` x x `->` e	fonction anonyme récursive
	|	e e	application de fonction
	|	$(e, \ldots, e)$	tuple
	|	`let` $(x_1, \ldots, x_n) = e$ `in` e	variables locales
d	::=	`let` x $=$ e	
$\oplus$	$\in$	$\{+, -\}$	

La construction `funrec` f x `->` $\ldots$ introduit une fonction anonyme *récursive*, où f est le nom à utiliser pour un appel récursif, et x le nom de l'argument. La construction `let` $(x_1, \ldots, x_n) = e$ `in` $\ldots$ permet de nommer les éléments d'un tuple (lorsque $n > 1$). Un programme est constitué d'une ou plusieurs déclarations d.

Voici un exemple de programme :

```
let fibo = fun n ->
  let fibo_aux = funrec aux acc -> fun n ->
    let (x, y) = acc in
    ifzero n then x else aux (y, x+y) (n-1)
  in
  fibo_aux (0, 1) n
```

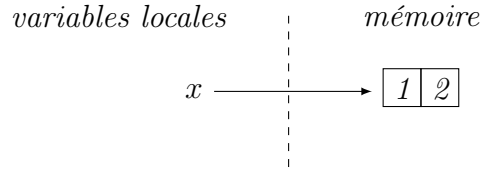
1.1 Échauffement

Question 1. *Écrire un programme définissant une fonction* `fact` *calculant la factorielle de son argument. On pourra définir une fonction auxiliaire calculant la multiplication (les opérations primitives du langage n'incluant que l'addition et la soustraction).*

Question 2. *Les valeurs de notre langage (entiers et tuples) sont représentés en mémoire de la même façon que les valeurs en OCaml. Par exemple, le programme :*

```
let x = (1, 2)
```

alloue un bloc en mémoire pointé par la variable `x` *et contenant les entiers* 1 *et* 2*, que l'on représente comme suit :*



Représenter l'état de la mémoire après l'exécution du programme suivant :

```
let x = (1, 2)
let y = (x, (3, 4), x)

let mktree1 = funrec mktree1 n ->
  ifzero n then 42 else (mktree1 (n-1), mktree1 (n-1))

let t1 = mktree1 3

let mktree2 = funrec mktree2 n ->
  ifzero n then 42 else
    let t = mktree2 (n-1) in (t, t)

let t2 = mktree2 3
```

1.2 Closure conversion

Dans cette partie, on s'intéresse à l'implémentation d'une transformation de *closure conversion* sur les programmes de notre langage. Cette transformation prend un programme du langage en entrée, et produit un programme équivalent dans lequel toutes les fonctions sont *closes*, autrement dit, n'ont pas de variables libres.

L'idée de la transformation est de traduire chaque fonction du programme source en une paire d'une fonction close et de son *environnement*. L'environnement d'une fonction est un tuple contenant les valeurs correspondant aux variables libres de la fonction d'origine.

Après traduction, chaque fonction (close) prend en argument une paire, indiquant à la fois l'argument de la fonction et l'environnement à utiliser.

Par exemple, le programme suivant :

```
let add = fun x -> fun y -> x + y
```

est transformé en :

```
let add =
  let code1 = fun xe ->
    let (x, env) = xe in
    let () = env in
    let code2 = fun ye ->
      let (y, env) = ye in
      let x = env in
      x + y
    in
    (code2, x)
  in
  (code1, ())
```

Question 3. *Comment doit-on traduire une application de fonction ? Traduire le programme suivant (où* `add` *dans le programme traduit réfèrera au* `add` *après transformation écrit ci-dessus) :*

```
let res = add 40 2
```

Question 4. *Définir (par induction sur e) une fonction* $\mathrm{fv}(e)$ *calculant l'ensemble des variables libres d'une expression e du langage.*

On implémente la transformation comme une fonction $\mathrm{closconv}(e)$, prenant en argument une expression e et renvoyant une expression traduite e', et définie par induction sur e.

Question 5. *Définir* closconv *pour le cas des fonctions anonymes.*

$$\mathrm{closconv}(\texttt{fun}\ x\ \texttt{->}\ e) = ...$$

Question 6. *Définir* closconv *pour le cas des appels de fonction.*

$$\mathrm{closconv}(e_1\ e_2) = ...$$

Les cas restants sont faciles à définir. (Pour le cas `funrec`, on le traduit vers un `funrec` de manière similaire au cas `fun`.)

On se pose maintenant la question du typage du code avant et après transformation.

Question 7. *On suppose notre langage muni d'un système de types similaire à celui d'OCaml.*

Donner alors un exemple de programme (bien typé), tel que, après transformation, le résultat serait mal typé. (Sans écrire le programme transformé, expliquer pourquoi le résultat de la transformation sur l'exemple donné serait mal typé.)

Indication : considérer une fonction d'ordre supérieur.

Question 8. *Dans une extension hypothétique d'OCaml avec types existentiels, on pourrait définir le type suivant :*

```
type ('a, 'b) clos = exists 'e. (('a * 'e) -> 'b) * 'e
```

Définir, en OCaml, un type `('a, 'b) clos` *équivalent à l'aide d'un type algébrique généralisé (GADT). L'utiliser pour implémenter deux fonctions OCaml* `mk` *et* `app` *avec les types suivants :*

```
val mk : (('a * 'e) -> 'b) -> 'e -> ('a, 'b) clos
val app : ('a, 'b) clos -> 'a -> 'b
```

(NB : `mk` *et* `app` *doivent être implémentées comme des fonctions prenant deux arguments et n'utilisant pas de fonctions anonymes supplémentaires.)*

Question 9. *Comment peut-on utiliser les fonctions* `mk` *et* `app` *pour définir une variante de* closconv *produisant du code bien typé dans tous les cas ? (Décrire cette variante informellement.)*

Appliquer la variante proposée au programme ci-dessous :

```
let add = fun x -> fun y -> x + y
let res = add 40 2
```

2 Noms frais

Lors de l'implémentation de transformations de programmes, il est souvent utile d'utiliser un mécanisme permettant de générer des noms frais, garantis différents d'un ensemble de noms déjà rencontrés.

Dans cette section, on s'intéresse à différentes abstractions exposant un tel mécanisme.

Question 10. *Un nom est ici représenté par une chaîne de caractères. Écrire une fonction OCaml* `gensym` *ayant le type* `unit -> string`*. À chaque appel,* `gensym` *doit renvoyer un nom différent de tous les noms renvoyés par les appels précédents à* `gensym`*.*

Question 11. *Une approche alternative est de voir la génération d'un nom frais comme un* effet *à part entière, et de définir une interface monadique des « calculs s'appuyant sur la génération de noms frais ».*

On propose ci-dessous l'interface pour une telle monade, munie d'une opération monadique `fresh` *permettant d'obtenir un nouveau nom frais, et une opération* `run` *permettant d'exécuter un calcul monadique.*

```
module type FreshMonad = sig
  type 'a t
  val return : 'a -> 'a t
  val bind : 'a t -> ('a -> 'b t) -> 'b t
  val fresh : string t
  val run : 'a t -> 'a
end
```

Écrire un module OCaml implémentant la signature `FreshMonad`*.*

3 Fonction `rayon::join` en Rust

En Rust, `rayon` est une bibliothèque populaire pour écrire des programmes multicœurs avec des threads légers. Entre autres, elle expose une fonction `join` permettant d'exécuter deux clôtures de manière concurrente (et potentiellement en parallèle sur deux cœurs différents). Cette fonction a le type suivant :

```
fn join<A, B, RA, RB>(oper_a: A, oper_b: B) -> (RA, RB) where
    A: FnOnce() -> RA,
    B: FnOnce() -> RB,
    A: Send,
    B: Send,
    RA: Send,
    RB: Send
```

Elle ne se termine que lorsque les deux clôtures ont fini leurs exécutions.

Question 12. *Expliquer le type de* `rayon::join`*.*

En particulier :

1. *À quoi servent les types* `A`*,* `B`*,* `RA` *et* `RB` *?*
2. *Expliquer à quoi sert la contrainte* `Send` *associée à* `A` *? Dans quel cas l'omettre poserait un problème de sûreté ?*
3. *Même question avec la contrainte* `Send` *associée à* `RA`*.*

Pour la question suivante, on pourra consulter l'annexe à la fin du sujet pour y lire le prototype de la fonction `spawn` :

Question 13. *Pourquoi la contrainte « +* `'static` *» utilisée pour* `spawn` *n'est-elle pas nécessaire pour* `rayon::join` *? Est-il possible d'implémenter* `rayon::join` *sans utiliser* `unsafe` *et en utilisant* `spawn` *comme seul moyen de créer un nouveau thread ? Inversement, est-il possible d'implémenter* `spawn` *en utilisant uniquement* `rayon::join` *pour créer un nouveau thread, et sans utiliser* `unsafe` *? Justifier vos réponses.*

4 Évaluation paresseuse en Rust

On définit en Rust les types suivants :

```
enum Thunk<E, V> {
    Unevaluated(E),
    Evaluating,
    Evaluated(V),
}
```

À chaque fois qu'une fonction utilisera le type `Thunk<E, V>`, elle aura la contrainte `E: FnOnce() -> V`. C'est-à-dire que `E` sera toujours le type d'une clôture ne prenant pas de paramètre et renvoyant une valeur de type `V`.

Le type `Thunk<E, V>` représente des *suspensions*. Les suspensions peuvent être utilisées pour reporter le calcul d'une valeur qui peut être coûteux au moment où on en a vraiment besoin. Si pendant l'exécution d'un programme on ne se sert jamais de cette valeur, on ne la calculera pas, et on ne paiera donc pas le coût de son calcul [1]. Par contre, une fois que le calcul est effectué, on stocke son résultat afin de s'assurer qu'il est effectué qu'une fois. Une suspension peut donc être dans trois états : `Unevaluated` lorsque le calcul n'a pas été effectué, `Evaluating` si celui-ci est en cours, et `Evaluated` lorsque celui-ci est en cours.

Pour les questions de programmation, on pourra utiliser l'annexe du sujet comme aide-mémoire des fonctions de la bibliothèque standard.

Question 14. *Écrire une fonction dont le prototype est*

```
fn new<E, V>(e: E) -> Thunk<E, V> where
    E: FnOnce() -> V
```

et permettant de créer une suspension à partir d'une clôture.

Question 15.

1. *Écrire une fonction dont le prototype est*

   ```
   fn force_mut<E, V>(t: &mut Thunk<E, V>) -> &V where
       E: FnOnce() -> V
   ```

   et permettant de forcer *une suspension, c'est-à-dire d'effectuer le calcul si nécessaire puis de renvoyer une référence sur la valeur contenue dans la suspension.*

2. *Est-il possible que le paramètre de `force_mut` soit dans l'état `Evaluating` ? Pourquoi ?*

3. *En pratique, il est souvent nécessaire de forcer les suspensions alors qu'elles sont* partagées. *Nous avons donc plutôt besoin d'une fonction de prototype :*

   ```
   fn force<E, V>(t: &Thunk<E, V>) -> &V where
       E: FnOnce() -> V
   ```

1. En plus d'être utilisées systématiquement pour toutes les expressions en Haskell, les suspensions sont le point de départ d'un célèbre livre d'algorithmique, *Purely Functional Data Structures*, de Chris Okasaki.

Expliquer pourquoi il n'est pas possible, avec la définition ci-dessus du type `Thunk`*, d'écrire une telle fonction correctement.*

4. *Discuter de modifications du type* `Thunk` *permettant d'écrire une telle fonction. On ne demande ni d'écrire le code correspondant, ni même de trouver une solution : on pourra simplement discuter des raisons pour lesquelles certaines idées ne fonctionnent pas.*

Question 16. *Écrire une fonction dont le prototype est*

```
fn unwrap<E, V>(t: Thunk<E, V>) -> V where
    E: FnOnce() -> V
```

et qui force la suspension donnée en paramètre, puis renvoie la valeur en propriété complète.

On définit le type suivant :

```
type Lazy<T> = Thunk<Box<dyn FnOnce() -> T>, T>;
```

Question 17. *Que signifie* `dyn FnOnce() -> T` *? Dans quel cas le type* `Lazy` *est-il utile ?*

On définit le type des « listes chaînées paresseuses » d'entiers non signés de 32 bits :

```
type LazyList = Lazy<Box<LazyListCell>>;
enum LazyListCell {
    Nil,
    Cons(u32, LazyList)
}
```

Question 18. *Écrire deux fonctions Rust de prototypes :*

```
fn infinite_list(x: u32) -> LazyList<u32>
fn append(a: LazyList<u32>, b: LazyList<u32>) -> LazyList<u32>
```

La première renvoie une liste paresseuse infinie ne contenant comme élément que l'entier passé en paramètre, et la deuxième renvoie la concaténation des deux listes passées en paramètre.

Attention, ces fonctions doivent être paresseuses au maximum : une cellule de liste ne doit être créée qu'à la demande, lorsque cela est strictement nécessaire.

Sommes-nous sûrs que ces fonctions terminent toujours ?

Annexe : aide mémoire Rust

Nous rappelons ici quelques éléments de la bibliothèque standard de Rust :

```
/* La fonction `spawn` crée un nouveau thread qui exécute la clôture passée
   en paramètre. */
fn spawn<F, T>(f: F) -> JoinHandle<T> where
    F: FnOnce() -> T,
    F: Send + 'static,
    T: Send + 'static

/* La fonction `std::mem::replace` remplace le contenu d'un emprunt unique
   par une valeur donnée, et renvoie l'ancien contenu de l'emprunt. */
fn replace<T>(dest: &mut T, src: T) -> T

/* La macro `panic!()` permet d'arrêter immédiatement le thread en cours
   d'exécution. À utiliser par exemple si un point de programme n'est pas
   sensé être atteint. Exemple d'utilisation : */
panic!("Message d'erreur optionnel");
```