Systèmes Synchrones
MPRI – cours 2.23-1 2023-2024

Examen
21 novembre 2023

L'énoncé est composé de 6 pages. Cette épreuve est prévue pour une durée de 3h. Les notes de cours sont autorisés.

Le sujet comporte deux parties, chacune contenant une dernière question qui vous prendra du temps. Il n'est pas nécessaire de faire les deux parties pour avoir tous les points! Vous pouvez ne faire qu'une des deux parties (mais complètement). Vous veillerez à décrire vos solutions avec précision. Le code qui vous est demandé devra être écrit en OCaml de manière lisible. Vous avez toute liberté d'utiliser des fonctions de la librairie standard.

Flots en ML

L'objectif de ce problème est de vous montrer deux représentations des suites infinies en ML et l'application des techniques classiques du synchrone pour compiler efficacement des fonctions de suites. On considère le mini-langage suivant. Un programme (*prog*) est un ensemble de définitions de noeuds (*def*). Un noeud f est une fonction à n entrées et dont le corps est une expression. Un pattern p est une simple variable (x) ou une liste (de longueur au moins deux) de variables. A des détails de syntaxe près, les expressions e sont celles du langage Lustre: un registre initialisé `pre` v e avec une valeur immediate (v); une valeur immédiate (v); une variable (x); une tuple ($(e_1, \ldots, e_n)$); l'application d'un opérateur combinatoire unaire $uop(e)$ ou binaire $bop(e_1, e_2)$; l'application $f(e_1, ..., e_n)$ d'un noeud f à n entrées; une conditionnelle stricte (`if` e_1 `then` e_2 `else` e_3); une définition locale `letrec` E `in` e dans laquelle E est un ensemble d'équations E définissant des suites mutuellement récursives. On suppose que les définitions sont implicitement récursives, c'est-à-dire que dans la définition `letrec` $x = f(x)$ `in` $x + 1$, x désigne la même valeur à gauche et à droite du signe $=$.

$$
\begin{array}{lll}
prog & ::= & def; ...; def \\
def & ::= & \texttt{node}\, f(p) = e \\
p & ::= & x \mid (x, ..., x) \\
e & ::= & \texttt{pre}\, v\, e \mid v \mid x \mid (e, \ldots, e) \mid uop(e) \mid bop(e, e) \mid \texttt{if}\, e\, \texttt{then}\, e\, \texttt{else}\, e \mid f(e, ..., e) \\
 & & \mid \texttt{letrec}\, E\, \texttt{in}\, e \\
E & ::= & E\, \texttt{and}\, E \mid p = e \\
v & ::= & 0 \mid 1 \mid ... \mid \texttt{true} \mid \texttt{false} \\
uop & ::= & + \mid - \mid \texttt{not} \mid ... \\
bop & ::= & + \mid - \mid * \mid ...
\end{array}
$$

On écrira simplement $x + y$ pour $(+)(x, y)$, le flot des entiers naturels s'écrit:

$$\texttt{letrec}\ nat = \texttt{pre}\ 0\ nat + 1\ \texttt{in}\ nat$$

Les définitions formelles ci-dessus se traduisent directement dans le code OCaml suivant.

```
type var = string
type prog = d list
and d = Node of string * var list * e
and p = var list
```

```
and e = | Epre of v * e | Eimm of v | Evar of var | Etuple of e list
        | Eop1 of string * e | Eop2 of string * e * e | Emux of e * e * e
        | Eapp of string * e list | Eletrec of eq list * e

and eq = p * e
and v = | Int of int | Bool of bool
```

1 Une représentation co-inductive des flots

La solution classique pour représenter des flots dans un langage fonctionnel paresseux tel que Haskell ou un environnement de preuve tel que Coq est de les voir comme des structure de données potentiellement infinie. En OCaml, l'exécution paresseuse doit être gérée explicitement en utilisant le module `Lazy` qui a l'interface suivante.

```
module Lazy : sig
  type 'a t
  (* OCaml: expr = ... | lazy expr | ... *)
  val force : 'a t -> 'a
end
```

Si l'expression OCaml `expr` est de type `'a`, l'expression `lazy expr` est de type `'a Lazy.t` et elle n'est réduite qu'avec l'application de `force` (sa valeur est alors mémoïsée).

Avec ce module, on définit les flots par le type suivant:

```
type 'a lazy_stream = 'a stream Lazy.t
and 'a stream = Cons of 'a * 'a lazy_stream

let hd_tl : 'a lazy_stream -> 'a * 'a lazy_stream =
  fun xs -> match Lazy.force xs with Cons(x, xs) -> x, xs

(* retourne les [n] premiers elements *)
let rec list_of : int -> 'a lazy_stream -> 'a list =
  fun n l -> if n = 0 then [] else let x, xs = hd_tl l in x :: (list_of (n-1) xs)
```

Les fonctions produisant des valeurs de ce type sont définies de manière récursive. Par exemple, le flot des entiers naturels commençant à n peut s'écrire:

```
let rec nat n = lazy (Cons(n, nat (n+1)))
```

Les opérations du mini-langage ont les signatures suivantes:

`pre . .`	:	$\forall\alpha.\alpha \to \alpha\ \mathit{lazy\_stream} \to \alpha\ \mathit{lazy\_stream}$
`lift0 .`	:	$\forall\alpha.\alpha \to \alpha\ \mathit{lazy\_stream}$
`lift1 . .`	:	$\forall\alpha,\beta.(\alpha \to \beta) \to \alpha\ \mathit{lazy\_stream} \to \beta\ \mathit{lazy\_stream}$
`lift2 . . .`	:	$\forall\alpha,\beta,\gamma.(\alpha \to \beta \to \gamma) \to \alpha\ \mathit{lazy\_stream} \to \beta\ \mathit{lazy\_stream} \to \gamma\ \mathit{lazy\_stream}$
`mux . . .`	:	$\forall\alpha.\mathit{bool}\ \mathit{lazy\_stream} \to \alpha\ \mathit{lazy\_stream} \to \alpha\ \mathit{lazy\_stream} \to \alpha\ \mathit{lazy\_stream}$

`bool` est le type des booléens et dont les deux constructeurs sont `false` et `true`. On souhaite que les autres opérations du langage, c'est-à-dire l'application de noeud $f(e_1, ..., e_n)$, les définitions locales de flots mutuellement récursives (`letrec` E `in` e) soient celles d'un langage fonctionnel existant, ici OCaml. En choisissant le type co-inductif précédent pour les flots, on peut donner l'implémentation suivante de quelques uns des opérateurs ci-dessus:

```
let pre v e = lazy (Cons(v, e))
let rec lift0 v = lazy (Cons(v, lift0 v))
```

```
let rec lift1 f xs =
  lazy (let x, xs = hd_tl xs in
        Cons(f x, lift1 f xs))
let plus1 xs = lift1 (fun x -> x+1) xs
```

Pour implémenter une définition récursive de flots, il faut utiliser la construction `lazy`. Pour accéder à son contenu, il faut utiliser l'opération `force` (du module `Lazy` de OCaml). Par exemple, si `plus1` désigne la fonction qui ajoute la valeur un à son entrée, le flot des entiers naturels s'écrit en OCaml:

```
let rec nat = lazy (Lazy.force (pre 0 (plus1 nat)))
```

Rappelez-vous qu'en OCaml dans une définition mutuellement récursive comme la suivante,

```
let rec x1 = e1 ... and xn = en in
```

Les expressions `ei` doivent être soit de la forme `fun x -> ...`, soit de la forme `lazy e`, soit de la forme `C e` où `C` est un constructeur. [1]. Ainsi:

```
 # let rec x = y and y = 42;;
Error: This kind of expression is not allowed as right-hand side of 'let rec'
```

Question 1 Compléter les définitions manquantes pour la conditionnelle et l'application point-à-point pour les arités deux et trois (`lift2` et `lift3`). (Mettez `assert false` si un cas est mal formé).

L'objectif est maintenant de pouvoir exécuter les programmes de ce mini-langage. Vous avez le choix de traduire les programmes du mini-langage vers des programmes fonctionnels OCaml; ou bien d'écrire un interprète en OCaml pour ce mini-langage.

Question 2 Commencez en donnant la traduction du programme suivante en OCaml.

$$\texttt{letrec}\ \mathit{fibo} = \texttt{pre}\ 0\ \mathit{pfibo}\ \texttt{and}\ \mathit{pfibo} = \texttt{pre}\ 1\ (\mathit{fibo} + \mathit{pfibo})\ \texttt{in}\ \mathit{fibo}$$

Question 3 Dans un premier temps, vous considèrerez seulement le cas de noeuds n'ayant qu'une entrée et qu'une sortie. De même, pour les équations récursives, traitez d'abord le cas d'une récursion simple de flot, de la forme `letrec` $x = e$ `in` e.

Définissez une fonction de traduction du mini-langage vers du code OCaml. Si vous le souhaitez (mais ce n'est pas indispensable), vous pouvez définir les types inductifs nécessaires pour représenter les termes du langage source et du langage cible.

Question 4 Proposer une traduction d'un ensemble de définitions mutuellement récursives de la forme `letrec` $x_1 = e_1$ `and` ... `and` $x_n = e_n$ `in` e?

Question 5 Traitez maintenant le cas de la définition d'un noeud ayant plusieurs entrées et plusieurs sorties.

2 Une représentation co-itérative des flots

Avec la représentation précédente, les flots sont construits progressivement et paresseusement. Cet encodage est inefficace pour trois raisons: (1) chaque opération va allouer un constructeur à chaque étape, constructeur qui devient inutile à l'étape suivante; (2) il nécessite l'usage d'un ramasse miettes pour désallouer les constructeurs devenant inutiles; (3) il s'appuie sur un mode d'exécution paresseuse couteux. Enfin, le fait que la quantité de mémoire nécessaire au cours du temps soit bornée est difficile à démontrer.

[1] En réalité, OCaml est un tout petit peu plus permissif.

On peut faire beaucoup mieux: l'élimination des structures intermédiaires dans les programmes ML est connue sous le nom de *deforestation* et les premières techniques sont dues à Philip Wadler [1]. Nous allons voir maintenant la solution adoptée dans les compilateur synchrones pour traiter les cas (1) et (2). Le cas (3) fait l'objet d'une dernière question.

Les flots infinis peuvent être définis de manière co-itérative:

```
type ('s, 'a) coStream = Co of ('s -> 'a * 's) * 's
```

Un flot $Co(f, s) : ('s, 'a)$ *coStream* est une paire formée d'une fonction de transition f et d'un état initial. La suite des valeurs est produite itérativement par application de la fonction de transition du flot. On passe d'un élément de $('s, 'a)$ *coStream* à $'a$ *lazy_stream* par la fonction `run`:

```
let rec run : ('a, 'b) coStream -> 'b lazy_stream =
  fun (Co(f, s)) ->
   let v, s' = f s in (lazy(Cons(v, run (Co(f, s')))))
```

Avec cette définition des flots, la suite des entiers naturels peut être construite, sans récursion, de la manière suivante:

```
let nat = Co((fun s -> s, (s+1)), 0)
```

L'opération produisant un flot constant s'écrit:

```
let lift0 v = Co((fun () -> v, ()), ())
```

L'opération d'addition de deux flot peut se définir ainsi:

```
let plus: ('a, int) coStream -> ('b, int) coStream -> ('a * 'b, int) coStream =
  fun (Co(f1, s1)) (Co(f2, s2)) ->
    Co((fun (s1, s2) ->
        let v1, s'1 = f1 s1 in
        let v2, s'2 = f2 s2 in
        (v1 + v2, (s'1, s'2))),
     (s1, s2))
```

Autrement dit, pour produire la valeur courante de la sortie, il suffit de calculer la valeur courante de chacune des entrées. L'état initial du système est le couple des états de l'entrée et de la sortie.

Question 6 Sur le mode précédent, proposer une implémentation des opérateurs `lift1`, `lift2` et `mux`.

Question 7 Quelle pourrait être l'implémentation de:

```
(lift1 (+) (lift1 (*) (lift0 1) (lift0 2)) (lift0 3))
```

Pour traiter une expression contenant un registre unitaire, il faut introduire une notion d'état. Une représentation de l'expression `(pre 0 (lift0 1)` pourra être:

```
Co (fun (s_pre, s) ->
      let v = 1 in
      s_pre, (v, s))
   (0, ())
```

On stocke dans la première composante de l'état `(s_pre, s)`, la valeur précédente du délai unitaire. Une implémentation pour `plus (pre 0 (lift0 1)) (pre 2 (lift0 3))` pourra être:

```
Co((fun ((s_pre1, s1), (s_pre2, s2)) ->
      let v1 = 1 in
      let v2 = 3 in
      (s_pre1 + s_pre2), ((v1, s1), (v2, s2))),
   ((0, ()), (2, ())))
```

Peut-on la simplifier (et comment) pour obtenir une autre implémentation équivalente? Précisément, $Co(f_1, s_1)$ et $Co(f_2, s_2)$ sont équivalents s'ils définissent le même flot, c'est-à-dire, si et seulement si *run* $Co(f_1, s_1) = run\ Co(f_2, s_2)$. A quelle équivalence classique correspond elle?

Question 8 Proposer une implémentation (non récursive) pour l'expression:

```
pre 0 (pre 1 (lift0 2))
```

Question 9 Proposer une implémentation (non récursive) pour l'équation:

```
fib = pre 1 (plus fib (pre 0 fib))
```

Question 10 Définir une implémentation (non récursive) pour l'opérateur `pre`.

```
let pre v (Co f s) = ...
```

Vous remarquerez que tous les opérateurs utilisés *préservent les longueurs*, c'est-à-dire qu'ils lisent une et une seule valeur de leur entrée pour produire une et une seule valeur de leur sortie. On dit aussi qu'ils sont synchrones. Plutôt que de voir une fonction comme un objet de type $\alpha\ lazy\_stream \rightarrow \beta\ lazy\_stream$ (ou, de manière équivalente $(s_1, \alpha)\ coStream\ \rightarrow (s_2, \beta)\ coStream$), nous allons définir le type des fonctions synchrones préservant les longueurs:

```
type ('a, 'b, 's) process = CoP of ('s -> 'a -> 'b * 's) * 's
```

Autrement dit, étant donné un état initial s, une fonction synchrone lit la valeur courante de son entrée et produit, la valeur courante de sa sortie ainsi qu'un nouvel état.

Question 11 Ecrire un opérateur `co_apply` qui permet d'appliquer une fonction synchrone (de type $Process\ 'a\ 'b\ 's$) à un flot d'entrées (de type $('a, 's)\ coStream$).

Etant donnée une fonction synchrone f et un flot v de valeur de type $'a$, `co_apply` $f\ v$ produit un flot de valeurs de type $'b$. Vous noterez que l'état (mémoire) de l'application doit contenir la mémoire interne de la fonction f et du flot v.

Question 12 Ecrire un opérateur `pipe` qui étant données deux fonctions synchrones, les composent, c'est-à-dire que `pipe f g x = f(g(x))`. Sa signature aura la forme suivante:

```
pipe : ('b, 'c, 's1) process -> ('a, 'b, 's2) process -> ('a, 'b, ...) process
```

Quel est le type correspondant à `...`?

Question 13 On veut maintenant définir l'opération nécessaire au calcul d'un point-fixe sur les flots. Dans un premier temps, vous considèrerez la situation d'une seule équation de flot de la forme `letrec` $x = C[$`pre` $v\ x]$ `in` e où $C[$`pre` $v\ x]$ désigne une expression dans laquelle la variable x apparaît à droite d'un opérateur de délai unitaire. On dit que cette récursion est "gardée".

Quel serait la manière d'implémenter cette définition récursive sous la forme co-itérative et sans utiliser de récursivité)?

Question 14 Proposer une fonction de traduction du mini-langage vers du OCaml sans récursivité, dans le cas où les noeuds n'ont qu'une entrée et qu'une sortie et ne contiennent qu'une seule équation récursive "gardée".

Question 15 Proposer une fonction de traduction pour une définition mutuellement récursive de la forme

$$\texttt{letrec}\ x_1 = e_1\ \texttt{and} \ldots \texttt{and}\ x_n = e_n\ \texttt{in}\ e$$

Quelles contraintes suffisantes doit-on imposer pour pouvoir produire du code OCaml non récursif?

Question 16 (plus long) Proposer une solution générale dans le cas d'équations mutuellement récursives, sans restriction syntaxique particulière; puis pour l'ensemble du minilangage (définitions de fonctions, application et tuples).

Pour cela, commencer par le cas d'une définition simple `letrec` $x = e_1$ `in` e_2. Vous pouvez utiliser les constructions du module `Lazy` de OCaml.

References

[1] P. Wadler. Deforestation: transforming programs to eliminate trees. *Theoretical Computer Science*, 73:231–248, 1990.