Systèmes Synchrones
MPRI – cours 2.23-1 2018-2019

Examen

25 novembre 2018

L'énoncé est composé de 7 pages. Cette épreuve est prévue pour une durée de 2h. Les notes de cours sont autorisés.

Le sujet comporte volontairement peu de questions. Elles sont là pour vous aider à atteindre l'objectif annoncé. Vous avez toute liberté de suivre une autre voie. Vous veillerez à décrire vos solutions avec précision.

Ce sujet est tiré de l'article [1] qui présente une technique de compilation reprise dans le compilateur du langage Lucid Synchrone.

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 (p) est un ensemble de définitions de noeuds (d). Un noeud f est une fonction à un paramètre x et qui retourne la valeur d'une expression e. A des détails de syntaxe près, les expressions sont celles du langage Lustre: un registre initialisé `pre` v e avec une valeur immediate (v); une valeur immédiate (v), une variable (x), une paire et ses fonctions d'accès, l'application d'un opérateur point-à-point à son entrée d'arité un ($op^1(e)$) ou deux ($op^2(e)$), une conditionnelle stricte (`if` e_1 `then` e_2 `else` e_3), l'application d'un noeud f à une entrée ($f(e)$), une définition locale (`let` $x = e_1$ `in` e_2) et une équation de point-fixe (`rec` $x = e$).

$$
\begin{array}{lcl}
p & ::= & d; ...; d \\
d & ::= & \texttt{node}\, f(x) = e \\
e & ::= & \texttt{pre}\, v\, e \mid v \mid x \mid (e, e) \mid \texttt{fst}\, e \mid \texttt{snd}\, e \\
 & & \quad \mid op^1(e) \mid op^2(e) \mid \texttt{if}\, e\, \texttt{then}\, e\, \texttt{else}\, e \mid f(e) \\
 & & \quad \mid \texttt{let}\, x = e\, \texttt{in}\, e \mid \texttt{rec}\, x = e \\
v & ::= & 0 \mid 1 \mid ... \mid \texttt{true} \mid \texttt{false} \\
op & ::= & + \mid - \mid * \mid ...
\end{array}
$$

Ainsi, $x + y$ pour $(+)^2(x, y)$, le flot des entiers naturels s'écrit: `rec` nat = `pre` $0\ nat + 1$. Le flot alternant de valeurs booléenne s'écrira: `rec` h = `pre true` $not^1(h)$ (ou simplement `rec` h = `pre true` $not(h)$). Ce noyau correspond à la définition des types inductifs suivants.

```
type p = d list
and d = Node of string * string * e
and e = | Epre of v * e | Eimmediate of v | Evar of string
        | Epair of e * e | Efst of e | Esnd of e
        | Eop1 of string * e | Eop2 of string * e * e | Emux of e * e * e
        | Eapp of string * e | Elet of string * e * e | Erec of string * e
```

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 voir comme des structures de données potentiellement infinie. On définit ainsi le type *co-inductif* des suites infinies de valeurs de type A en Haskell:

```
data Stream A = Cons A (Stream A)
```

A la difference de OCaml, Haskell permet de manipuler des structures infinies telles que celles décrites par le type ci-dessus.

Les fonctions produisant des valeurs du type `Stream A` sont définies de manière co-récursive. Par exemple, le flot des entiers naturels commençant à n peut s'écrire:

```
nat(n) = Cons n (nat(n+1))
```

On voit ici que la définition d'une telle fonction dans un langage avec appel par valeur tel que OCaml ne produit pas de valeur de type *Stream A* : `nat 0` mais boucle infiniment. En revanche, en Haskell, la suite est produite progressivement et à la demande. Les suites paresseuses sont représentables en OCaml en utilisant les fonctions du module `Lazy`. Les opérations du mini-langage ont les signatures suivantes:

`pre . .`	:	$A \to \mathit{Stream}\ A \to \mathit{Stream}\ A$
`lift0 .`	:	$A \to \mathit{Stream}\ A$
`lift1 . .`	:	$(A \to B) \to \mathit{Stream}\ A \to \mathit{Stream}\ B$
`lift2 . . .`	:	$(A \to B \to C) \to \mathit{Stream}\ A \to \mathit{Stream}\ B \to \mathit{Stream}\ C$
`mux . . .`	:	$\mathit{Stream}\ \mathit{Bool} \to \mathit{Stream}\ A \to \mathit{Stream}\ A \to \mathit{Stream}\ A$
`pair . .`	:	$A \to B \to A \times B$
`fst .`	:	$A \times B \to A$
`snd .`	:	$A \times B \to B$

`Bool` est le type des booléens et dont les deux constructeurs sont `False` et `True`. Toutes les autres opérations (application $f(e)$, définition locale (`let` $x = e_1$ `in` e_2 et récursion `rec` $x = e$) sont celles d'un langage fonctionnel paresseux. En choisissant le type co-inductif précédent pour les flots, on peut donner l'implémentation suivante des opérateurs ci-dessus:

```
pre v e = Cons v e
lift0 v = Cons v (lift0 v)
lift1 f (Cons v e) = Cons (f v) (lift1 f e)
lift2 f (Cons v1 e1) (Cons v2 e2) = Cons (f v1 v2) (lift2 f e1 e2)
mux (Cons x xs) (Cons t ts) (Cons f fs) =
    Cons (if x then t else f) (mux xs ts fs)
```

En notant `un` pour `lift0 1`, le flot des entiers naturels s'écrit:

```
nat = pre 0 (plus nat un)
```

Question 1 On souhaite ajouter un opérateur `spair` qui prend deux flots et produit un flot de couples, un opérateur `sfst` qui prend un flot de couples et rend le flot formé de la première composante. `ssnd` agit de même avec la seconde composante.[1] A partir des opérateurs donnés précédemment, proposer un encodage des opérateurs `spair` . ., `sfst` . et `ssnd` . dont les signatures sont:

$$\begin{array}{lcl} \texttt{spair}\,.\,. & : & \mathit{Stream}\ A \to \mathit{Stream}\ B \to \mathit{Stream}\ A \times B \\ \texttt{sfst}\,. & : & \mathit{Stream}\ A \times B \to \mathit{Stream}\ A \\ \texttt{ssnd}\,. & : & \mathit{Stream}\ A \times B \to \mathit{Stream}\ B \end{array}$$

L'objectif est maintenant de traduire les programmes du mini-langage vers des programmes fonctionnels paresseux écrits en Haskell. Les seules constructions dont vous avez besoin ici pour le langage cible sont celles d'abstraction, de définition locale et d'application. Vous pourrez écrire `fun x -> e` (en lieu et place de `\ x -> e`) et `let rec x = e1 in e2` (en lieu et place de `let x = e1 in e2` puisque les définitions sont implicitement récursives en Haskell).

```
node f(x) =
  let reset = fst x in
  let top = snd x in
  rec count = if reset then 0
              else (if top then ((pre 0 count) + 1)
              else pre 0 count)
```

sera traduit en:

```
let f(x) =
  let reset = fst x in
  let top = snd x in
  let count = mux reset (lift0 0)
                  (mux top (lift2 (+) (pre 0 count) (lift0 1))
                  (pre 0 count)
  in count
```

[1] Ce sont les opérateurs de bus en Simulink (dit de "signal routing").

Question 2 Ecrire une fonction *trad-vers-haskell*, en OCaml, qui traduit les programmes du mini-langage vers Haskell. Vous pourrez, soit directement produire une chaîne de caractère, soit (mieux) définir un type inductif auxiliaire pour reprenter les expressions de Haskell dont vous avez besoin (et vous épargner l'écriture d'un *printer*).

2 Une représentation co-itérative des flots

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

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 [2]. C'est un grand classique de l'optimisation de programmes fonctionnels (avec des publications régulières sur le sujet depuis 20 ans à la conférence ICFP). 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:

```
data CoStream S A = Co (S -> A * S) S
```

Un flot *Co f s* : *CoStream S A* est une paire formée d'une fonction de transition f de type $S \to A \times S$ 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 *CoStream S A* à *Stream A* par la fonction `run`:

```
run (Co f s) = let v, s' = f s in
               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:

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

L'operation produisant un flot constant s'écrit:

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

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

```
plus (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)

plus :: CoStream Int S1 -> CoStream Int S2 -> CoStream Int (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 3 Sur le mode précédent, proposer une implémentation des opérateurs `lift1`, `lift2` et `mux`.

Question 4 Quelle pourrait être l'implémenation 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 5 Proposer une implémentation (non récursive) pour l'expression:

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

Question 6 Proposer une implémentation (non récursive) pour l'équation:

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

Question 7 Définir une implémentation (non récursive) pour l'opérateur `pre`.

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

Vous remarquerez que tous les opérateurs utilisés *préservent les longueurs*, autrement dit, lisent une et une seule valeur de leur entrée pour produire une et une seule valeur de leur sortie. Ils sont synchrones, en un sens strict donc. Nous allons exploiter cette caractéristique pour le traitement des définitions de fonctions. Plutôt que de voir une fonction comme un objet de type *Stream* $A \rightarrow$ *Stream* B (ou, de manière équivalente *CoStream* A $S_1 \rightarrow$ *CoStream* B S_2), nous allons définir le type des fonctions synchrones préservant les longueurs par:

```
data Process A B S = CoP (S -> A -> B * S) S
```

Autrement dit, étant donné un état initial $s : S$, une fonction synchrone lit la valeur courante de son entrée (de type A) et produit, la valeur courante de sa sortie (de type B) ainsi qu'un nouvel état.

Question 8 Ecrire un opérateur `co_apply` qui permet d'appliquer une fonction synchrone (de type *Process* A B S_1) à un flot d'entrées (de type *CoStream* A S_2). La signature de cet opérateur pourra être:

```
co_apply: Process A B S1 -> (Stream A S2 -> Stream B (S2, S1))
```

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 9 Ecrire un opérateur `pipe` qui étant données deux fonctions synchrones, les compose. Sa signature est:

```
pipe : Process A B S1 -> Process B C S2 -> CoStream A S3
                      -> CoStream C (S1, S2, S3)
```

Question 10 Il reste à définir l'opération nécessaire au calcul d'un point-fixe sur les flots. Ecrire une définition pour l'opérateur `co_fix` de signature suivante:

```
co_fix : Process (A * B) B S1 -> CoStream A S2 -> CoStream B (S1, S2)
```

Cet opérateur prend en argument une fonction f, un flot x et calcule le point-fixe de l'équation $y = f(x, y)$.

Question 11 A partir des questions précédentes, écrire une nouvelle implémentation de la fonction *trad-vers-haskell* qui traduit les programmes du mini-langage vers du code Haskell où les flots sont construits co-itérativement.

Question 12 Pouvez-vous justifier que le code généré nécessite seulement une mémoire bornée dont la taille maximale peut être calculée statiquement?

Question 13 (optionnel) Sous quelles conditions peut-on produire un code co-itératif séquentiel ne nécessitant aucun mécanisme de paresse? Autrement dit, sous quelles conditions on peut donner une implémentation sans aucune récursion à l'opérateur `co_fix`?

Question 14 (optionnel)

Les structures paresseuses peuvent être représentées en OCaml en utilisant les fonctions du module `Lazy`. Ainsi, le type des listes infinies s'écrit:

```
open Lazy

(* flots paresseux *)
type 'a stream = Cons of 'a * 'a stream lazy_t
```

La documentation OCaml indique:

> A value of type `'a Lazy.t` is a deferred computation, called a suspension, that has a result of type 'a. The special expression syntax lazy (expr) makes a suspension of the computation of expr, without computing expr itself yet. "Forcing" the suspension will then compute expr and return its result.

```
val force : 'a t -> 'a
```

> force x forces the suspension x and returns its result. If x has already been forced, Lazy.force x returns the same value again without recomputing it. If it raised an exception, the same exception is raised again. Raise Lazy.Undefined if the forcing of x tries to force x itself recursively.

Reprendre les questions précédentes en générant du OCaml plutôt que du Haskell.

References

[1] Paul Caspi and Marc Pouzet. A Co-iterative Characterization of Synchronous Stream Functions. In *Coalgebraic Methods in Computer Science (CMCS'98)*, Electronic Notes in Theoretical Computer Science, March 1998. Extended version available as a VER-IMAG tech. report no. 97–07 at `www.di.ens.fr/~pouzet/bib/bib.html`.

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