Partiel du cours PROG1

6 novembre 2024

Durée : 1 heure 30 minutes.
Documents autorisés, machines interdites.

A. Listes en λ-calcul

On se donne l'encodage suivant des listes en λ-calcul, où $[]$ représente la liste vide et $h :: t$ représente la liste dont la tête est h et dont la queue est t :

$$([]) \stackrel{\text{def}}{=} \lambda n\, c.n$$
$$(h :: t) \stackrel{\text{def}}{=} \lambda n\, c.c\, h\, t$$

Pour inspecter le contenu d'une liste, on définit comme suit l'opérateur $\mathsf{match}\ l\ \mathsf{with}\ [] \mapsto bn \mid :: \mapsto bc$, qui exécute la branche bn si l est vide et qui exécute la branche bc avec les argument h et t si l est $h :: t$:

$$(\mathsf{match}\ l\ \mathsf{with}\ [] \mapsto bn \mid :: \mapsto bc) \stackrel{\text{def}}{=} l\, bn\, bc$$

On veut que l'encodage satisfasse les conditions suivantes [1], pour une certaine stratégie de réduction $\longrightarrow$:

$$\frac{t \longrightarrow t'}{(\mathsf{match}\ t\ \mathsf{with}\ [] \mapsto bn \mid :: \mapsto bc) \longrightarrow (\mathsf{match}\ t'\ \mathsf{with}\ [] \mapsto bn \mid :: \mapsto bc)}$$

$$\frac{}{(\mathsf{match}\ []\ \mathsf{with}\ [] \mapsto bn \mid :: \mapsto bc) \longrightarrow bn} \qquad \frac{}{(\mathsf{match}\ h :: t\ \mathsf{with}\ [] \mapsto bn \mid :: \mapsto bc) \longrightarrow bc\, h\, t}$$

Question 1: Montrer que ces conditions sont vérifiées si $\longrightarrow$ est la stratégie de réduction en appel par nom.

Question 2: Les règles ci-dessus sont en petits pas. Donner des règles équivalentes en grands pas.

Question 3: Réduire $(\mathsf{match}\ h :: t\ \mathsf{with}\ [] \mapsto \Omega \mid :: \mapsto (\lambda h\, c.[]))$ en appel par valeur. Expliquer le problème par rapport aux conditions attendues sur les réductions de notre encodage.

Pour la fin de l'exercice, on considère que $\longrightarrow$ est en appel par nom. On considère l'opérateur de point fixe $\Theta \stackrel{\text{def}}{=} (\lambda x\, y.y\, (x\, x\, y))\, (\lambda x\, y.y\, (x\, x\, y))$.

Question 4: Montrer que l'on a $\Theta\, f \longrightarrow^* f\, (\Theta\, f)$.

Question 5: Donner une définition d'un opérateur $\texttt{map}(f, t)$ en λ-calcul, basée sur les encodages précédents,

1. Autrement dit, pour une relation $\longrightarrow$ encodant une stratégie de réduction possible du λ-calcul, on souhaite que, pour chaque règle, la conclusion soit vérifiée quand les prémisses le sont.

et satisfaisant les règles suivantes :

$$\frac{t \longrightarrow t'}{\texttt{map}(f,t) \longrightarrow \texttt{map}(f,t')}$$

$$\frac{}{\texttt{map}(f,[]) \longrightarrow []}$$

$$\frac{}{\texttt{map}(f,h :: t) \longrightarrow (f\ h) :: \texttt{map}(f,t)}$$

B. Cachoteries

On considère le code OCaml suivant, vu comme [2] une expression Mini-ML e :

```
let f =
  let r = ref (fun x -> x) in
  let f = fun x -> if x = 0 then 1 else x * !r (x-1) in
  r := f;
  !r
in
f 3
```

Question 1: On cherche à évaluer e dans l'environnement et la mémoire vides. Indiquer pour quelles valeurs de E, σ, σ' et n on verra apparaître $E \vdash \sigma, (f\ 3) \Downarrow \sigma', n$ dans la dérivation de $\emptyset \vdash \emptyset, e \Downarrow \sigma', n$.

Question 2: Donner une expression OCaml de type `int -> bool` qui renvoie un booléen aléatoire [3] pour chaque nouvelle entrée, mais renvoie toujours le même booléen pour une entrée donnée. Autrement dit, cette fonction serait indistinguable d'une fonction pure choisie aléatoirement dans `int -> bool`.

- **Attention :** on demande une *expression*, il n'est donc pas permis d'écrire des *déclarations* toplevel OCaml. En particulier, il n'est pas possible d'utiliser une variable globale mutable (qui pourrait de toute façon être utilisée de façon maladroite par d'autres fonctions de notre application fictive) : tout utilisation d'un état mutable doit être cachée.
- Vous pouvez utiliser des fonctions de la librairie standard comme `Hashtbl.create` ou `List.assoc`.

C. Sémantique non-déterministe

On considère les expressions arithmétiques suivantes, où n dénote une constante entière et x une variable :

$$e ::= x \mid e_1 + e_2$$

On munit ces expressions d'une sémantique à grands pas non-déterministe, basée sur des environnements (notés E) qui associent à chaque variable un *ensemble* de valeurs entières possibles. Une première sémantique à grands pas est donnée pour ce langage, via la relation $E \vdash e \Downarrow_1 n$ exprimant qu'une expression e peut s'évaluer en $n \in \mathbb{N}$ dans l'environnement E :

$$\frac{}{E \vdash x \Downarrow_1 n}\ n \in E(x) \qquad \frac{E \vdash e_1 \Downarrow_1 n_1 \quad E \vdash e_2 \Downarrow_1 n_2}{E \vdash e_1 + e_2 \Downarrow_1 n_1 + n_2}$$

Question 1: Donner toutes les valeurs de n telles que $E \vdash x + x \Downarrow_1 n$ quand $E(x) = \{1, 2\}$.

2. Pour cela, on remplace `if x = 0 then ...` par `ifz x then ...` et éventuellement `fun x -> ...` par `fixfun f x -> ...`.
3. On utilisera ici `Random.bool : unit -> bool`.

Une seconde sémantique à grands pas, donnée par la relation $E \vdash e \Downarrow_2 S$ définie ci-dessous, donne d'un coup l'ensemble $S \subseteq \mathbb{N}$ de toutes les valeurs possibles pour e dans l'environnement E :

$$\frac{}{E \vdash x \Downarrow_2 E(x)} \qquad \frac{E \vdash e_1 \Downarrow_2 S_1 \quad E \vdash e_2 \Downarrow_2 S_2}{E \vdash e_1 + e_2 \Downarrow_2 \{n_1 + n_2 \mid n_1 \in S_1, n_2 \in S_2\}}$$

On admet que cette seconde relation est totale et fonctionnelle :

$$\text{pour tous } e \text{ et } E \text{ tels que } \mathsf{fv}(e) \subseteq \mathsf{dom}(E), \text{ il existe } S \text{ tel que } E \vdash e \Downarrow_2 S \tag{1}$$

$$\text{pour tous } e, E, S \text{ et } S' \text{ tels que } E \vdash e \Downarrow_2 S \text{ et } E \vdash e \Downarrow_2 S', \text{ on a } S = S' \tag{2}$$

Ceci nous permet de définir $[\![e]\!]_E$ comme l'unique $S \subseteq \mathbb{N}$ tel que $E \vdash e \Downarrow_2 S$.

Question 2: Soit E un environnement et e une expression. Montrer la propriété suivante, i.e. la correction de $\Downarrow_2$ par rapport à $\Downarrow_1$:

$$[\![e]\!]_E \subseteq \{n \in \mathbb{N} \mid E \vdash e \Downarrow_1 n\} \tag{3}$$

On admet la propriété symétrique, i.e. la complétude de $\Downarrow_2$ par rapport à $\Downarrow_1$:

$$\{n \in \mathbb{N} \mid E \vdash e \Downarrow_1 n\} \subseteq [\![e]\!]_E \tag{4}$$

Définitions locales

On étend notre langage d'expressions avec des définitions locales :

$$e ::= x \mid e_1 + e_2 \mid \mathsf{let}\ x = e_1\ \mathsf{in}\ e_2$$

On ajoute les règles suivantes aux définitions de $\Downarrow_1$ et $\Downarrow_2$:

$$\frac{E \vdash e_1 \Downarrow_1 n_1 \quad E + \{x \mapsto \{n_1\}\} \vdash e_2 \Downarrow_1 n_2}{E \vdash \mathsf{let}\ x = e_1\ \mathsf{in}\ e_2 \Downarrow_1 n_2} \qquad \frac{E \vdash e_1 \Downarrow_1 S_1 \quad E + \{x \mapsto S_1\} \vdash e_2 \Downarrow_1 S_2}{E \vdash \mathsf{let}\ x = e_1\ \mathsf{in}\ e_2 \Downarrow_2 S_2}$$

Question 3: Parmi les propriétés (1) à (4) ci-dessus, lesquelles restent vraies dans cette extension ? Justifier brièvement sans détailler aucune preuve.

Conditionnelles

On enrichit notre langage de départ avec un ifz comme en Mini-ML :

$$e ::= x \mid e_1 + e_2 \mid \mathsf{ifz}\ e_1\ \mathsf{then}\ e_2\ \mathsf{else}\ e_3$$

La définition initiale de la sémantique $\Downarrow_1$ est étendue par les règles suivantes :

$$\frac{E \vdash e_1 \Downarrow_1 0 \quad E \vdash e_2 \Downarrow_1 n}{E \vdash \mathsf{ifz}\ e_1\ \mathsf{then}\ e_2\ \mathsf{else}\ e_3 \Downarrow_1 n} \qquad \frac{E \vdash e_1 \Downarrow_1 m \quad E \vdash e_3 \Downarrow_1 n}{E \vdash \mathsf{ifz}\ e_1\ \mathsf{then}\ e_2\ \mathsf{else}\ e_3 \Downarrow_1 n}\ m \neq 0$$

Question 4: Étendre la définition de $\Downarrow_2$ pour que les propriétés (1) à (4) restent vraies. Aucune preuve n'est demandée, mais vérifiez que vous sauriez les faire pour être sûrs de ne rien oublier !

Implémentation

On revient pour cette dernière partie au langage original. On définit les types suivants :

```
type expr = Var of string | Add of expr*expr
type env = (string * int list) list
```

Question 5: Implémenter un interprète pour la sémantique $\Downarrow_1$, qui itère une fonction donnée en argument sur tous les résultats possibles :

```
val eval : env -> expr -> (int -> unit) -> unit
```

Ainsi, `eval ["x",[1;2];"y",[1]] (Add (Var "x", Var "y"))` devrait être équivalent à `fun f -> List.iter f [2;3]`.

Question 6: Implémenter une fonction `first` de type `env -> expr -> int` qui renvoie un entier pair vers lequel l'expression peut s'évaluer, si un tel entier existe, et lève l'exception `Not_found` sinon. Votre implémentation ne devra pas ré-implémenter une variante de la fonction d'évaluation mais utiliser `eval env expr` "en boîte noire". De plus, l'évaluation devra être interrompue à la première valeur paire rencontrée : dans ce cas on souhaite éviter d'itérer sur les valeurs suivantes.

Question 7: Passer votre interprète en CPS :

```
val eval_cps :
  env -> expr -> (int -> (unit -> 'a) -> 'a) -> (unit -> 'a) -> 'a
```

Question 8: Pouvez-vous faire la même chose qu'en question 6 avec `eval_cps`, sans utiliser aucun effet (état, exceptions, entrées-sorties, etc.) ? On renverra un `int option` plutôt que de lever une exception en cas d'échec.