Examen du cours PROG1

18 décembre 2023

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

1 Habitation

Dans tout cet exercice on considère le λ-calcul simplement typé, et des extensions de ce système. On dit qu'un type τ est habité quand il existe un terme (clos) M tel que $\emptyset \vdash M : \tau$. Par exemple, pour tous types τ et τ', le type $\tau \to \tau' \to \tau$ est habité : le terme $\lambda x.\lambda y.x$ en est le *témoin*.

Question 0 Montrer que le type $(\tau \to \tau') \to (\tau \to \tau' \to \tau'') \to (\tau \to \tau'')$ est habité quels que soient τ, τ' et τ''.

Question 1 On considère une extension du λ-calcul simplement typé avec des constantes : pour chaque type de base τ on se donne un nouveau symbole c_τ représentant une constante arbitraire de ce type. La syntaxe des termes est alors étendue en

$$M ::= M\ M' \mid \lambda x.M \mid x \mid c$$

où x est une variable quelconque et c une des constantes c_τ pour un type de base τ. Par exemple, si τ_1 et τ_2 sont des types de base, $\lambda f.\ f\ c_{\tau_1}\ c_{\tau_2}$ est un terme. On étend enfin les règles de typage du λ-calcul simplement typé en ajoutant, pour chaque type de base τ, la règle suivante :

$$\frac{}{\Gamma \vdash c_\tau : \tau}$$

Montrer que tout type est habité dans ce λ-calcul enrichi.

Question 2 On retourne maintenant au λ-calcul simplement typé du cours, sans constantes ajoutées, où $M ::= M\ M' \mid \lambda x.M \mid x$. Montrer qu'aucun type de base n'est habité. *Indice : on pourra utiliser le théorème de normalisation des termes typés.*

Question 3 On considère maintenant une autre extension du λ-calcul simplement typé. On étend les termes avec le **fixfun** de Mini-ML, noté ici avec la lettre grecque μ, puis on étend la réduction et le typage comme suit :

$$M ::= M\ M' \mid \lambda x.M \mid x \mid \mu f x.M \qquad (\mu f x.M)\ M' \to_\mu M[f := \mu f x.M][x := M']$$

$$\frac{\Gamma, f : \tau \to \tau', x : \tau \vdash M : \tau'}{\Gamma \vdash \mu f x. M : \tau \to \tau'}$$

Montrer que tout type est habité dans cette extension du λ-calcul.

Question 4 On retourne finalement au λ-calcul simplement typé standard. On définit inductivement le jugement "$\Gamma \vdash \tau$ habité", pour un type τ et un ensemble de types Γ, par les deux règles d'inférence suivantes :

$$\frac{}{\Gamma, \tau \vdash \tau \text{ habité}} \qquad \frac{\Gamma, \tau \vdash \tau' \text{ habité}}{\Gamma \vdash \tau \to \tau' \text{ habité}}$$

Démontrer que si $\emptyset \vdash \tau$ habité est dérivable, alors τ est habité. Montrer que la réciproque n'est pas vraie, et proposer une façon de corriger cela.

2 Rust

On considère un fragment du Mini-Rust vu en cours, sans branchement conditionnel, et où la seule fonction disponible est **drop**, de type $(\mathsf{String}) \to ()$. On considère de plus uniquement deux types possibles dans ce langage : String et $()$. On donne en figure 1 un sous-ensemble des règles de typage vues en cours. Dans cet exercice, nous appellerons ce système de type *strict*, ce qu'on indique en décorant les $\vdash$ en $\vdash_S$. On considère en figure 2 une variante *laxiste* de ce système de type, pour laquelle on utilise $\vdash_L$. Seules les trois dernières règles diffèrent entre les deux systèmes. Dans les deux figures, s dénote une chaîne de caractère constante (e.g. `"bonjour"`) et les variables τ, τ' prennent leurs valeurs dans $\{\mathsf{String}, ()\}$. Dans les deux dernières règles laxistes, on écrit $\tau'_\perp$ pour signifier que la valeur spéciale $\perp$ est autorisée.

Question 1 Donner une expression Micro-Rust close qui type dans le système laxiste mais pas dans le système strict.

$$\frac{}{\Gamma, x : \tau \vdash_S x : \tau \Rrightarrow \Gamma, x : \perp} \qquad \frac{}{\Gamma \vdash_S s : \mathsf{String} \Rrightarrow \Gamma} \qquad \frac{\Gamma \vdash_S e : \mathsf{String} \Rrightarrow \Gamma'}{\Gamma \vdash_S \mathbf{drop}(e) : () \Rrightarrow \Gamma'}$$

$$\frac{\Gamma \vdash_S e : () \Rrightarrow \Gamma' \quad \Gamma' \vdash_S e' : \tau \Rrightarrow \Gamma''}{\Gamma \vdash_S (e; e') : \tau \Rrightarrow \Gamma''}$$

$$\frac{\Gamma \vdash_S e : \tau \Rrightarrow \Gamma' \quad \Gamma', x : \tau \vdash_S e' : \tau' \Rrightarrow \Gamma'', x : \perp}{\Gamma \vdash_S (\mathbf{let}\ x = e\ \mathbf{in}\ e') : \tau' \Rrightarrow \Gamma''}$$

$$\frac{\Gamma \vdash_S e : \tau \Rrightarrow \Gamma', x : \perp}{\Gamma \vdash_S (x = e) : () \Rrightarrow \Gamma', x : \tau}$$

FIGURE 1 – Règles de typage strict de Micro-Rust

$$\frac{}{\Gamma, x : \tau \vdash_L x : \tau \Rrightarrow \Gamma, x : \bot} \qquad \frac{}{\Gamma \vdash_L s : \mathsf{String} \Rrightarrow \Gamma} \qquad \frac{\Gamma \vdash_L e : \mathsf{String} \Rrightarrow \Gamma'}{\Gamma \vdash_L \mathbf{drop}(e) : () \Rrightarrow \Gamma'}$$

$$\frac{\Gamma \vdash_L e : \tau' \Rrightarrow \Gamma' \quad \Gamma' \vdash_L e' : \tau \Rrightarrow \Gamma''}{\Gamma \vdash_L (e; e') : \tau \Rrightarrow \Gamma''}$$

$$\frac{\Gamma \vdash_L e : \tau \Rrightarrow \Gamma' \quad \Gamma', x : \tau \vdash_L e' : \tau' \Rrightarrow \Gamma'', x : \tau'_\bot}{\Gamma \vdash_L (\mathbf{let}\ x = e\ \mathbf{in}\ e') : \tau' \Rrightarrow \Gamma''}$$

$$\frac{\Gamma \vdash_L e : \tau \Rrightarrow \Gamma', x : \tau'_\bot}{\Gamma \vdash_L (x = e) : () \Rrightarrow \Gamma', x : \tau}$$

FIGURE 2 – Règles de typage laxiste de Micro-Rust

Question 2 Montrer que pour toute expression Micro-Rust e, et pour tous Γ, Γ', τ tels que $\Gamma \vdash_L e : \tau \Rrightarrow \Gamma'$, il existe e' tel que $\Gamma \vdash_S e' : \tau \Rrightarrow \Gamma'$. Une preuve formelle, par induction, est exigée. Elle devra induire une construction de e' à partir de e, de sorte que la nouvelle expression s'exécute de façon "similaire" à l'originale. *Indice : il s'agit de corriger dans e des problèmes de gestion mémoire.*

3 OCaml : GADTs, foncteurs et CPS

Question 1 Considérons l'exemple suivant de GADT (type algébrique généralisé) :

```
type _ t =
  | C1 : int t
  | C2 : string t -> string t
```

Pour quels types `x` existe-t-il des *valeurs* de type `x t` ?

Question 2 Définir un type `'a bla` tel que les valeurs de type `x bla`, pour un type `x` quelconque, soient exactement les valeurs

— `X`,
— `A(X)`,
— `L(A(X))`,
— `B(L(A(X)))`,
— `A(B(L(A(X))))`,
— `L(A(B(L(A(X)))))`,
— `B(L(A(B(L(A(X))))))`,
— etc.

On ne demande pas que toutes ces valeurs aient le type `x t` pour un même `x` : le paramètre de type peut (et doit) varier d'une valeur à l'autre.

```ocaml
module type SET = sig
  type t
  type elt
  val empty : t
  val add : elt -> t -> t
  val remove : elt -> t -> t
  val member : elt -> t -> bool
  val cardinal : t -> int
end
```

FIGURE 3 – Signature du type de données “ensemble”.

Question 3 On considère la signature de la figure 3. Implémenter un foncteur qui transforme une implémentation de `SET` en une nouvelle implémentation dans laquelle la fonction `cardinal` est en $\mathcal{O}(1)$. Le foncteur prendra en argument un module de type `SET` quelconque, pour lequel on supposera seulement (sans les expliciter) des invariants naturels, et l'on construira un nouveau module qui devra satisfaire ces mêmes invariants. Un exemple d'invariant naturel est :

```ocaml
if not (member x t) then cardinal (add x t) = 1 + cardinal t
else cardinal (add x t) = cardinal t
```

Question 4 Les deux fonctions suivantes ne sont *pas* en CPS. Donner le type de chaque fonction, expliquer pourquoi elle n'est pas en CPS, et reformuler pour obtenir une fonction en CPS.

```ocaml
let rec fold f a l k =
  match l with
  | [] -> k a
  | hd::tl -> fold f a tl (fun a' -> f a' hd)

let rec sum_assoc keys l k =
  match keys with
  | [] -> k 0
  | hd::tl ->
    sum_assoc tl l
      (fun s ->
         try k (s + List.assoc hd l) with
         | Not_found -> k s)
```

On rappelle que `List.assoc x l` cherche le premier élément de `l` de la forme `(x,y)` et renvoie `y`. Si aucun élément n'a `x` en première composante, l'exception `Not_found` est levée.