Devoir sur table de PROG1

16 décembre 2024

Durée : 2 heures. Documents autorisés, machines interdites.

A. Style "tagless-final" avec les modules

Dans cet exercice, on explore une façon inhabituelle d'écrire des interpréteurs. Nous considérons un langage simple d'expressions arithmétiques construites à partir des constantes 0 et 1 au moyen des opérations d'addition et de multiplication. Ce langage est représenté par la signature suivante :

```
module type ARITH = sig
  type t
  val zero : t
  val one : t
  val add : t -> t -> t
  val mul : t -> t -> t
end
```

On définit ci-dessous un foncteur qui calcule l'expression "$1+(0\times1)$" au moyen d'une implémentation/interprétation arbitraire de `ARITH` :

```
module Ex1(A:ARITH) = struct
  let v = A.add A.one (A.mul A.zero A.one)
end
```

On dira qu'un module `M:ARITH` est un anneau quand :

- les opérations `M.add` et `M.mul` sont associatives et commutatives ;
- `M.zero` et `M.one` sont respectivement des unités pour `M.add` et `M.mul`, et `M.zero` est absorbant pour `M.mul` ;
- `M.mul` se distribue sur `M.add`.

L'égalité considérée pour ces propriétés est l'égalité structurelle OCaml. Ainsi, la distributivité signifie que pour toutes valeurs `x`, `y` et `z` on a, en OCaml, `M.mul x (M.add y z)= M.add (M.mul x y)(M.mul x z)`.

Question 1: Définir un module `IntArith` qui implémente la signature `ARITH` avec `type t = int` et les opérations naturelles sur les entiers. Définir ensuite `BoolArith` sur le type `bool` implémentant l'anneau $\mathbb{Z}/2\mathbb{Z}$. Ces deux modules doivent définir des anneaux. La valeur `v` de `Ex1(IntArith)` doit être égale à `1`.

On se donne maintenant le type algébrique suivant décrivant les expressions arithmétiques non-interprétées :

```
type aexpr = Zero | One | Add of aexpr*aexpr | Mul of aexpr*aexpr
```

Question 2: Définir un module `FreeArith` implémentant `ARITH` avec `type t = aexpr`. Ce module n'est pas un anneau, mais chaque opération de la signature est implémentée par la construction associée dans le type `aexpr`. La valeur `v` de `Ex1(FreeArith)` vaut ainsi `Add (One, Mul (Zero, One))`.

Question 3: Définir un foncteur `Eval` prenant en argument un module `A:ARITH` et renvoyant une structure contenant une fonction `eval : aexpr -> A.t` qui interprète une aexpr dans `A`. Le test suivant doit ainsi passer pour tout `A:ARITH` :

```
module M = Eval(A)
module E = Ex1(A)
let () = assert (E.v = M.eval (Add (One, Mul (Zero, One))))
```

Ajout de l'exponentiation

On considère maintenant l'extension de notre langage avec une opération "puissance", pour laquelle on définit un nouveau foncteur d'exemple :

```
module type POLY = sig
  include ARITH
  val exp : t -> int -> t
end
```

```
module Ex2(P:POLY) = struct
  open P
  let v = exp (add one one) 3
end
```

Question 4: Définir un foncteur `Poly` qui prend en argument un module `A:ARITH` et renvoie une implémentation de `POLY` qui étend `A` en définissant `exp` récursivement en termes des opérations de `A`. On suivra précisément l'équation $x^{n+1} = x \times x^n$ et pas l'une de ses variantes, ce qui a son importance si `A` n'est pas un anneau.

Question 5: De façon analogue à `FreeArith`, définir `FreePoly` qui implémente `POLY` sur le type `pexpr` suivant :

```
type pexpr = Zero | One | Add of pexpr*pexpr | Mul of pexpr*pexpr | Exp of pexpr*int
```

Question 6: On considère le test suivant :

```
module Test2(P:POLY) = struct
  module Ex2 = Ex2(P)
  let one = P.one
  let two = P.add one one
  let four = P.add two two
  let eight = P.add four four
  let () =
    Format.printf "%b␣%b␣%b\n"
      (Ex2.v = four)
      (Ex2.v = eight)
      (Ex2.v = P.mul two (P.mul two (P.mul two one)))
end
```

Quels sont les trois booléens affichés lorsqu'on instancie `Test2(Poly(IntArith))`, `Test2(Poly(BoolArith))`, `Test2(Poly(FreeArith))` et `Test2(FreePoly)` ?

Ajout d'une variable

On étend le langage de départ avec une variable x, on définit un foncteur interprétant l'expression $x \times (x+1)$, et l'on se donne aussi une signature pour les modules déclarant juste une variable x :

```
module type ARITHX = sig
  include ARITH
  val x : t
end
```

```
module Ex3(A:ARITHX) = struct
  let v = A.mul A.x (A.add A.x A.one)
end
```

```
module type X = sig
  type t
  val x : t
end
```

Question 7: Définir un foncteur `MakeX` prenant une implémentation de `X` et une implémentation de `ARITH` sur le même type, et renvoie une implémentation de `ARITHX` étendant l'implémentation de `ARITH` avec la variable fournie en premier argument. La signature de votre foncteur doit, par exemple, permettre de faire fonctionner le test suivant (mais il doit aussi, bien sûr, permettre des exemples similaires sur `BoolArith`) :

```
module M = MakeX(struct type t = int let x = 42 end)(IntArith)
module E = Ex3(M)
let () = assert (42*43 = E.v)
```

Optimisation

Pour finir, nous allons définir un foncteur `Optim` permettant d'optimiser une implémentation de `ARITH` en forçant les égalités correspondant aux axiomes $0 + x = x$, $0 \times x = 0$, $1 \times x = x$ et leurs variantes par

commutativité. Par exemple, dans `Optim(FreeArith)`, on aura égalité entre $1 + (0 \times 1)$ et 1, ce qui n'est pas le cas dans `FreeArith`. Le foncteur ne devra forcer aucune autre égalité que celles demandées. On exige par ailleurs que le surcoût lié à l'optimisation soit constant pour chaque opération, ce qui exclut d'utiliser l'égalité structurelle dont le coût peut être arbitraire.

Pour réaliser cela, l'interprétation dans `Optim(A)` va porter sur des valeurs dans un type enrichi par rapport à `A.t`, permettant de déterminer si une valeur vaut zéro ou un. Afin de pouvoir faire le lien entre le type des expressions dans `A` et `Optim(A)`, on introduit une extension de `ARITH` avec des fonctions de traduction entre le type `t` et un type "source" `s` :

```
module type ARITHS = sig
  include ARITH
  type s
  val from_s : s -> t
  val to_s : t -> s
end
```

Question 8: Définir un foncteur `Optim` répondant à ces exigences, prenant une implémentation de `ARITH` et renvoyant une implémentation de `ARITHS` sur les types pertinents. Le test suivant devrait ainsi fonctionner :

```
module M = Optim(FreeArith)
module E = Ex1(M)
module F = Ex1(FreeArith)
let () = assert (M.to_s E.v <> F.v)
let () = assert (M.to_s E.v = FreeArith.one)
```

B. Typage linéaire

On considère les λ-termes usuels, notés avec les lettres M, $N\ldots$ et des types simples linéaires, notés avec la lettre τ, qui ne diffèrent des types simples du cours que par la notation de la flèche :

$$\begin{array}{lll} M & ::= & x \mid (\lambda x.M) \mid (M\ M') \qquad x \text{ une variable} \\ \tau & ::= & \tau_b \mid \tau \multimap \tau' \qquad \tau_b \text{ un type de base} \end{array}$$

Un environnement de typage est un ensemble d'associations de la forme $x : \tau$. On ne s'autorisera ici à former l'union (Γ, Γ') de deux environnements de typage que si ceux-ci portent sur des variables distinctes. Ainsi on pourra former l'union de $x : \tau$ et $y : \tau, z : \tau'$, mais pas celle de $x : \tau$ et $x : \tau, y : \tau'$.

On définit un système de types linéaire pour notre λ-calcul par les règles suivantes :

$$\frac{}{x : \tau \vdash x : \tau} \qquad \frac{\Gamma, x : \tau \vdash M : \tau'}{\Gamma \vdash \lambda x.M : \tau \multimap \tau'} \qquad \frac{\Gamma \vdash M : \tau \multimap \tau' \quad \Gamma' \vdash N : \tau}{\Gamma, \Gamma' \vdash M\ N : \tau'}$$

On remarquera que la règle de typage pour les variables impose que l'environnement soit un singleton.

Question 1: Pour chacun des termes suivants, donner une dérivation de type ou justifier que le terme n'est pas typable :

(a) $\lambda a.\lambda b.\lambda f.\ f\ a\ b$
(b) $\lambda a.\lambda f.\ f\ a\ a$
(c) $\lambda a.\lambda f.\ f\ (f a)$
(d) $\lambda a.\lambda b.\lambda c.\ b\ (a\ c)$

Question 2: Le lemme de substitution du λ-calcul simplement typé énonce que si $\Gamma, x : \tau \vdash M : \tau'$ et $\Gamma \vdash N : \tau$, alors $\Gamma \vdash M[x := N] : \tau'$.

(a) Expliquer pourquoi cette propriété n'est pas correcte dans le cadre linéaire.

(b) Proposer un énoncé correct, et le démontrer.

Question 3: Démontrer que la propriété de préservation du typage par β-réduction reste vraie pour le système linéaire.

Question 4: Comment la preuve de normalisation faible du λ-calcul simplement typé s'adapte-t-elle dans le cadre linéaire ? *Bonus* : proposer une borne sur la longueur des réductions d'un terme bien typé, en fonction des poids de ses redexes.

C. Garbage collection

On s'intéresse à deux mécanismes de gestion mémoire, les *références faibles* et les *éphémérons*, en adoptant deux points de vue : d'une part, on va étendre l'algorithme *mark & sweep* basique pour supporter ces mécanismes avancés ; d'autre part, on va voir l'utilité de ces mécanismes sur le cas d'usage commun de la mémoïsation. Dans les deux cas on écrira du code OCaml, mais il ne faut pas confondre les deux niveaux : dans le premier cas le code OCaml implémente l'environnement d'exécution de nos programmes, et ce code est d'habitude écrit en C ; dans le second cas, il s'agit de code OCaml usuel, destiné à être compilé et exécuté dans l'environnement d'exécution qui précède. Quand on veut insister sur le fait qu'on est dans de la seconde situation, on parle de programmes OCaml "utilisateur".

Pour commencer nous mettons en place une vision idéalisée des valeurs OCaml :

```
type value = Immediate of int | Boxed of int
type box = {
  values : value array ;
  (* ... à compléter ... *)
}
let memory : box option array = Array.make 1024 None
```

Une valeur est soit représentée de façon immédiate par un entier (dans la réalité, on aurait ici 63 bits) soit représentée par l'index dans `memory` d'une valeur de type `box` (dans la réalité, on aurait ici un pointeur vers le tas). Une paire `(x,y)` sera par exemple représentée par un pointeur `Boxed i` avec [1] `(get_some memory.(i)).values` un tableau `[|vx;vy|]` contenant les représentations des valeurs `x` et `y`.

Le type `box` ci-dessus n'est pas complet. Vous pourrez le compléter à votre convenance dans les questions suivantes. Mais avant cela, nous y ajoutons un champ `finalise : value -> unit` : la fonction ainsi attachée à une `box` devra être appelée avant le nettoyage de la valeur correspondante par le garbage collector (GC). Ce champ permettrait d'implémenter la fonctionnalité suivante accessible depuis les programmes OCaml "utilisateur" :

```
Gc.finalise : ('a -> unit) -> 'a -> unit
(** [finalise f v] registers [f] as a finalisation function for [v]. *)
```

Question 1: On suppose que l'on dispose de fonctions `make_1 : value -> box` et `make_2 : value -> value -> box` permettant de créer des `box` contenant respectivement une et deux valeurs, avec un finaliseur par défaut. On suppose que `memory` est initialement remplie de `None`, puis modifiée ainsi :

```
memory.(12) <- Some (make_1 (Boxed 3)) ;
memory.(3)  <- Some (make_2 (Immediate 3) (Boxed 12)) ;
memory.(20) <- Some (make_2 (Boxed 20) (Boxed 3)) ;
memory.(24) <- Some (make_2 (Immediate 7) (Immediate 8))
```

Quelles valeurs sont vivantes, si l'on considère une unique racine `Boxed 3` ?

1. On utilise ici l'utilitaire `let get_some = function Some x -> x | None -> assert false` pour indiquer que la mémoire est allouée et accéder à la `box` contenue.

Question 2: Définir une fonction `collect : value list -> unit` qui prend une liste de racines et implémente l'algorithme *mark & sweep*, en prenant en compte les fonctions de finalisation. Pour la phase de nettoyage, la libération d'une case de `memory` est simplement représentée par l'écriture de `None` dans cette case. Vous pouvez étendre le type `box` si besoin, en précisant comment les nouveaux champs doivent être initialisés. *Attention*, cette fonction va être modifiée deux fois dans la suite : soignez-la, et prévoyez de la place pour pouvoir identifier les sous-sections qui auront été adaptées.

Références faibles

Une *référence faible* est une référence vers une valeur, mais qui ne maintient pas cette valeur en vie. Ainsi la valeur contenue dans une référence peut être nettoyée par le GC même si la référence reste en vie. Dans ce cas, pour éviter d'accéder à une valeur inexistante, la référence devra avoir été "vidée" par le GC.

Les règles de vivacité sont modifiées en présence de références faibles :
- Les racines sont vivantes.
- Si une valeur *autre qu'une référence faibe* est vivante alors les valeurs qu'elle contient sont aussi vivantes.

Ainsi, si l'on a uniquement en mémoire une valeur v et un pointeur faible p contenant v, et que ce pointeur est l'unique racine, alors v n'est pas vivante et pourra être nettoyée par le GC.

Pour ajouter les références faibles dans notre modèle de valeurs, on ajoute simplement un champ `weak : bool` aux enregistrements `box` et l'on supposera que quand ce champ est vrai, le tableau `values` est toujours de taille 1. Par convention, on considère qu'une référence est vide si elle contient `Immediate 0` — en effet, une référence pleine n'est utile que si elle contient une valeur `Boxed`.

Question 3: Implémenter une nouvelle fonction `collect` qui prend en compte les références faibles lors de chacune des phases. Le temps d'exécution de chaque phase doit rester globalement inchangé : linéaire en le nombre de blocs vivants pour le marquage, et en la taille du tableau mémoire pour le balayage.

On considère maintenant l'utilisation des références faibles dans du code OCaml "utilisateur", dans le contexte de la mémoïsation. On part d'un dispositif simple, en évitant les tables de hachage pour simplifier :

```
let memo f =
  let cache = ref [] in
  fun x ->
    try List.assoc x cache with
    | Not_found ->
      let res = f x in
      cache := (x,res) :: !cache ;
      res
```

On peut alors définir, pour une fonction donnée `f`, sa version mémoïsée `f' = memo f`. Tant que `f'` est vivante, le cache et toutes ses entrées vont le rester aussi. Il serait pourtant légitime de pouvoir nettoyer les entrées correspondant aux clés `x` qui ne sont plus utilisées ailleurs dans le programme !

En premier lieu, il faut que `x` ne soit pas maintenu vivant juste par sa présence en tant que clé dans le cache. Pour cela, au lieu de stocker `(x,res)` dans `cache`, on va y stocker `(w,res)` où `w` est une référence faible sur `x`. Cela va permettre à `x` d'être supprimé par le GC. On utilisera enfin un finaliseur pour que, quand `x` est supprimé, l'entrée `(w,res)` soit à son tour supprimée du cache.

Question 4: Améliorer la fonction `memo` précédente pour implémenter ce dispositif, en utilisant l'API (fictive) suivante :

```
val WRef.make : 'a -> 'a WRef.t
  (** [make v] crée une référence faible contenant [v]. *)
val WRef.get : 'a WRef.t -> 'a option
  (** [get v] renvoie [Some v] si la référence contient [v], [None] si elle a été vidée. *)
```

Éphémérons

Il reste un problème avec le dispositif précédent. Supposons par exemple que notre fonction `f` calcule, pour un arbre `x`, l'ensemble de ses sous-arbres satisfaisant une certaine condition, et que cet ensemble puisse contenir `x` lui-même. Après avoir calculé le résultat `res` et stocké `(w,res)` dans le cache, la valeur `x` va alors rester vivante (via `res`) tant que la fonction mémoïsée reste vivante. On voudrait cependant, de nouveau, pouvoir supprimer cette entrée du cache si `x` est inutilisé ailleurs dans le programme. Pour cela, les éphémérons vont nous être utiles.

Un éphéméron est une paire spéciale dont la première composante est appélée clé et la seconde valeur. Comme une référence faible, un éphéméron vivant ne maintient pas en vie sa clé et, si la clé d'un éphéméron est libérée par le GC, les deux champs de l'éphéméron vont être vidés. La nouveauté est qu'un éphéméron vivant ne maintient en vie sa valeur que si sa clé est vivante — si la clé est vivante, c'est forcément pour d'autres raisons que sa présence en tant que clé dans l'éphéméron. Comme une référence faible correspond à un éphéméron avec une valeur arbitraire, on ne considère plus de références faibles dans la suite.

Plus précisément, les règles de vivacité sont modifiées ainsi en présence d'éphémérons :
- Une racine est vivante.
- Si une valeur *autre qu'un éphéméron* est vivante alors les valeurs qu'elle contient sont aussi vivantes.
- *Si un éphéméron est vivant, non vidé, et a une clé vivante, alors sa valeur est vivante.*

Question 5: Pour cette question on se donne trois paires d'entiers v_1, v_2, v_3, distinctes deux à deux.
(a) On suppose qu'on a comme unique racine une paire (v_1, e) et e est un éphéméron avec la clé v_2 et la valeur v_3, ce qu'on note $e = (v_2, v_3)$. Quelles valeurs sont vivantes ? Après garbage collection, e est-il vide ?
(b) Comme en (a) mais en rajoutant v_3 comme racine.
(c) Comme en (a) mais en rajoutant cette fois v_2 comme racine.
(d) On suppose qu'on a pour racines v_1 et un éphéméron $e = (v_1, e')$ où $e' = (v_2, v_3)$ est un second éphéméron. Quelles valeurs sont vivantes ? Après garbage collection, e et e' sont-il vides ?
(e) On suppose qu'on a pour racines v_2 et un éphéméron $e = (v_1, e')$ où $e' = (v_2, v_3)$ est un second éphéméron. Quelles valeurs sont vivantes ? Après garbage collection, e et e' sont-il vides ?

Question 6: Proposer une modification du type `box` et de la fonction `collect` qui prenne en compte les éphémérons, en gardant si possible des complexités linéaires respectivement en le nombre de boxes allouées et en la taille de la mémoire. Justifier la correction de votre fonction vis-à-vis de la nouvelle notion de vivacité.

Question 7: Donner une variante de la fonction `memo` qui utilise les éphémérons pour gérer au mieux la mémoire, en utilisant l'API (fictive) suivante :

```
val Eph.create : 'a -> 'b -> ('a,'b) Eph.t
  (** [create k v] crée un nouvel éphéméron avec clé [k] et valeur [v]. *)
val Eph.query : ('a,'b) Eph.t -> 'a -> 'b option
  (** [query e k] renvoie [Some v] si l'éphéméron [e] n'a pas été vidé,
      et si sa clé est [k] et sa valeur [v];
      sinon [None] est renvoyé. *)
```