Second demi-projet : interpréter, compiler

Rendu 3 : pour le 24 avril à 23h59

par mail à `henning.basold@ens-lyon.fr`,
`daniel.hirschkoff@ens-lyon.fr`,
`bertrand.simon@ens-lyon.fr`

NB : il y a aussi un *rendu intermédiaire* : cf. la fin de ce document
(un autre rendu suivra, dans le prolongement de celui-ci)

Préambule

Ce sujet est écrit en faisant l'hypothèse que tout a été traité (plus ou moins bien...) lors du rendu 2. Si tel n'est pas le cas, envoyez rapidement *un mail à* `daniel.hirschkoff@ens-lyon.fr` *pour indiquer ce qui manque dans votre rendu 2 par rapport au menu escompté.*
Le but étant de ne pas trop "charger la barque", et de maintenir une exigence raisonnable : si le rendu 2 est insuffisant, il n'y a pas de raison que le rendu 3 le soit automatiquement.

1 Débutants

Traitez le menu Intermédiaires du rendu 2 : fonctions, fonctions récursives.

Pour ce faire, commencez par étendre le type des expressions, et rendez-vous compte du fait que la fonction d'évaluation peut désormais renvoyer soit un entier, soit une fonction (ou plutôt une clôture : cf. les transparents du cours).

2 Intermédiaires

Traitez la partie "Avancés" du rendu 2 : traits impératifs, couples.

Traitez aussi l'extension suivante :

1. **Les exceptions.**

   Il y a une seule exception, qui est notée `E`, et prend un entier en argument (c'est "acquis", au sens où les programmes fouine ne contiendront pas de déclaration `exception E of int`).

   Il y a, comme en Caml, les constructions `raise` et `try .. with`.

   Exemples : `raise (E 17)` `let k = try f x with E x -> x+1`

   NB : on pourra omettre le "|" facultatif dans le `with E x -> ...`

   NB2 : il n'est pas demandé de traiter des cas de filtrage comme `with | E 2 -> ..`, on se contentera de `E x` (quitte à mettre après des `if x=2 then.. else..`).

   La règle du jeu est qu'il ne faut pas implémenter une exception en utilisant une exception de Caml ; il faut donc réfléchir à comment implémenter les exceptions.

3 Avancés

Traitez la partie **"Exceptions"** du rendu des Intermédiaires, ainsi que ce qui suit.

Le but est ici d'implémenter les transformations de programmes vues lors de la dernière séance en amphi. L'approche est la suivante :

(i) on récupère un programme fouine "étendu" grâce aux lexer&parser ;

(*ii*) on applique la transformation, de manière à obtenir un programme fouine sans extension ;

on est alors capable d'afficher le programme résultant (même si on ne sait pas si on a vraiment envie de le contempler) ;

(*iii*) on interprète le programme "fouine pur" ainsi obtenu avec l'interprète programmé au rendu 2.

1. **Transformations de programmes : aspects impératifs, passage d'état.**

   Les explications ont été données en cours. Le fichier (ou module Caml, si vous savez ce que cela signifie) qui fait cette transformation devra s'appuyer sur l'extension avec les couples que vous avez programmée.

   Il devra également s'appuyer sur une implémentation (en Caml, a priori, pas en fouine) de la mémoire, que vous donnerez dans un fichier à part, et qui sera utilisée au moment d'exécuter/interpréter le programme fouine résultant de la transformation (au même titre, si l'on veut, que pour `prInt`). Le programme résultant de la transformation sera ainsi un programme fouine utilisant les trois primitives suivantes (veillez à respecter les types) :[1]

   ```
   alloc :  mem -> (loc*mem)
   read :  mem -> loc -> valeur
   write :  mem -> loc -> valeur -> mem
   ```

   où `valeur` est le type de ce qu'on peut stocker dans la mémoire, `loc` est le type des "cases mémoires" (les références, si vous voulez), et `mem` est le type de la mémoire (l'état du monde)[2]

   Si vous voulez être dans l'esprit à tout prix, vous pouvez concevoir une implémentation purement fonctionnelle de ces trois primitives (à coup de paires, ou alors en étendant fouine avec des listes).

2. **Transformations de programmes : exceptions, continuations.**

   Référez-vous à ce qui a été décrit en cours, pour implémenter la transformation qui élimine les constructions propres aux exceptions afin d'obtenir un programme fouine. La transformation vue en cours s'appuie sur le style "par continuations"[3].

Remarques sur les transformations de programmes.

- *Toujours une fonction.* Comme l'indiquent les transparents du cours, la transformation d'un programme donne toujours une fonction (et le programme fouine obtenu a un type différent par rapport au programme de départ).

  Pour faire tourner le programme résultant de la transformation, vous devez lui passer un argument supplémentaire (l'état initial de la mémoire, la continuation "banale" qu'est la fonction identité).
- *Étendre les transformations.* En cours nous n'avons parlé que des transformations éliminant un trait de programmation à partir du langage (fouine + ce trait). Il vous faudra étendre les définitions vues en cours pour pouvoir prendre en compte (fouine + références + exceptions).
- *Factoriser.* Autant que faire se peut, privilégiez l'élégance et la modularité dans le code : il serait bien d'éviter d'avoir 3 fonctions ne partageant rien pour implémenter les quatre transformations demandées dans cette partie 4.
- *Amélioration possible.* Vos transformations pourraient traiter "astucieusement" la curryfication, en évitant d'appliquer systématiquement la transformation de base lorsqu'on a affaire à une définition du style `let f x y z = ....`

[1] La version précédente du sujet contenait les types absurdes suivants : `alloc :  unit -> loc`, `read :  loc -> valeur`, `write :  loc -> valeur -> unit`.

[2] Vous n'avez bien sûr pas à implémenter les types — on veut pouvoir tester vos programmes, et que ceux-ci contiennent des `(write s r v1)` plutôt que des `(write s v1 r)`.

[3] On rappelle les cas de l'application et du let:

```
⟦e1 e2⟧ = fun (k,kE) -> ⟦e2⟧((fun v -> ⟦e1⟧((fun f -> f v (k,kE)), kE)) , kE)
⟦let x = e1 in e2⟧ = fun (k,kE) -> ⟦e1⟧(fun x -> ⟦e2⟧(k,kE) )
```

- *Les bonus.* Les transformations de programmes portent sur le langage fouine avec références et/ou exceptions, mais sans les bonus décrits ci-dessous. Si vous traitez des bonus, pas besoin d'étendre les transformations de programmes.

3. **Bonus, facultatif.**

   Si tout ce qui est demandé marche, a été testé, et est tout propre, vous pouvez traiter les extensions suivantes :

   - Listes et/ou types somme : cf. rendu 2.
   - Au-delà des exceptions.

     Ajoutez une instruction `call/cc` dans le langage source (en regardant sur internet comment ça marche — `call/cc` signifie *"call with current continuation"*).

     Cette extension peut faire sens à la fois pour l'interprète et pour la transformation de programmes.

     Si le cœur vous en dit, vous pouvez même voir si vous pouvez traiter les *continuations délimitées* (`shift/reset`).
   - Tableaux.

     Ajoutez des tableaux d'entiers, pour pouvoir écrire des programmes qui "font quelque chose", à part additionner et multiplier des entiers. Les tableaux seront manipulés à l'aide des primitives suivantes :

     – `aMake` prend un argument un entier k et renvoie un tableau de taille k, initialisé à 0 partout ;
     – comme en Caml, on utilise `t.(1)` pour accéder à la deuxième case du tableau `t`, et on utilise `<-` pour modifier une case du tableau.

     On peut en Caml définir `amake` de la façon suivante :

```
# let aMake = fun k -> Array.make k 0;;
val aMake : int -> int array = <fun>
```

4 Spécification : exécutable, options, etc.

Options.

Sans options, le programme est exécuté avec l'interprète : il affiche ce qu'on lui dit d'afficher en utilisant `prInt`.

`-debug` Comme avant.

`-E` Applique la transformation pour éliminer les exceptions, puis exécute le programme ainsi obtenu à l'aide de l'interprète (rendu 2).

`-R` Même chose, pour la transformation pour éliminer les aspects impératifs.

`-ER` La combinaison des deux transformations (d'abord `R`, puis `E`).

`-RE` L'autre combinaison des deux transformations (`E` puis `R`).

`-outcode` Cette option, combinée avec l'une des 4 options ci-dessus, aura pour effet d'afficher à l'écran le programme résultant de la transformation, sans l'exécuter (par exemple : `./fouine -E -outcode pgm.ml`).

Des tests. Fournissez un répertoire avec des programmes de test que vous avez soumis à `fouine`.

Veillez à ce que les tests couvrent l'ensemble des aspects que vous traitez (lex-yacc, exécution des divers composants du langage fouine).

Indiquez s'il y a des tests qui échouent, en raison de bugs pas encore résolus.

Le rendu. Vous pouvez vous reporter aux rendus 1 et 2 : l'esprit est le même, en ce qui concerne la propreté du rendu, les explications, etc. N'oubliez pas d'expliquer dans le README comment vous avez procédé, notamment si vous traitez la partie sur les transformations de programmes.

** Important : rendu intermédiaire **

Vous avez du temps pour traiter ce rendu 3, mais la séance prévue le 6 avril ne pourra être assurée. Avec les vacances, cela fera juste une séance, le 13 avril, pour discuter de l'avancement du rendu.
Il vous est donc demandé de nous envoyer un **rendu intermédiaire, pour le 12 avril à 23h59**, consistant en une archive contenant :

- l'état courant de votre code (tant pis si tout ne marche pas, tant pis si tout n'est pas très propre)
- un README, éventuellement succinct, indiquant
    1. ce qui marche
    2. ce qui n'a pas encore traité
    3. les points qui causent des difficultés

Comme toujours, votre mail est à envoyer à
`henning.basold@ens-lyon.fr`, `daniel.hirschkoff@ens-lyon.fr`, `bertrand.simon@ens-lyon.fr`.