Projet programmation L3IF, hiver-printemps 2021

notes de cours

1 Intermédiaires et avancés : exceptions

1.1 Comment ça marche en Caml

On étend fouine avec une seule exception, appelée `E`, et qui prend un `int` en argument.

```
exception E of int
```

(la ligne ci-dessus n'est pas à écrire dans vos programmes fouine ; il faut l'inclure en revanche si on veut exécuter un programme fouine à l'aide du compilateur Caml).

Les exceptions servent à . . . gérer les cas exceptionnels.

Soit la fonction suivante, qui multiplie tous les entiers d'une liste.

```
let rec l_mul_naive = fun l ->
  match l with
  | [] -> 1
  | x::xs -> x*l_mul_naive xs
```

En faisant

```
let l = [ 2;4;1;0;3;5 ]

let k1 = l_mul_naive l
```

ça renverra `0` pour `k1`. Lors du calcul de `k1`, Caml considère la forme de la liste `l`, puis
s'appelle récursivement sur `[4;1;0;3;5]` en se souvenant qu'il faudra multiplier le résultat par 2, puis
s'appelle récursivement sur `[1;0;3;5]` en se souvenant qu'il faudra multiplier le résultat par 4, puis
s'appelle récursivement sur `[0;3;5]` en se souvenant qu'il faudra multiplier le résultat par 1, puis
s'appelle récursivement sur `[3;5]` en se souvenant qu'il faudra multiplier le résultat par 0, puis
s'appelle récursivement sur `[5]` en se souvenant qu'il faudra multiplier le résultat par 3, puis
s'appelle récursivement sur `[]` en se souvenant qu'il faudra multiplier le résultat par 5, puis
renvoie 1, puis
renvoie 5, puis
renvoie 15, puis
renvoie 0, puis
renvoie 0, puis
renvoie 0.

Vous le savez sans doute, les appels aux fonctions sont gérés à l'aide d'une pile, ce qui correspond au fonctionnement ci-dessus : un appel d'une fonction fait un "push", une fonction qui renvoie un résultat fait un "pop".

Dans l'exécution ci-dessus, les parties colorées en violet et en orange sont désolantes.

Une première amélioration peut être apportée ainsi :

```
let rec l_mul_if = fun l ->
  match l with
  | [] -> 1
  | x::xs -> if x=0 then 0 else x*l_mul_if xs
```

Le calcul de `k2 = l_mul_if l` se déroule ainsi : Caml
s'appelle récursivement sur `[4;1;0;3;5]` en se souvenant qu'il faudra multiplier le résultat par 2, puis
s'appelle récursivement sur `[1;0;3;5]` en se souvenant qu'il faudra multiplier le résultat par 4, puis
s'appelle récursivement sur `[0;3;5]` en se souvenant qu'il faudra multiplier le résultat par 1, puis
renvoie 0, puis
renvoie 0, puis
renvoie 0.

On est content, car le `if then else` permet d'éviter les appels récursifs inutiles après avoir rencontré `0`. Voici une troisième version de cette fonction, pour être moins désolés :

```
let l_mul_exc = fun linit ->
  let rec lme_aux = fun l ->
    match l with
    | [] -> 1
    | x::xs -> if x=0 then raise (E 5420) (* entier arbitraire *)
                      else x*lme_aux xs
  in
  try lme_aux linit
  with (E n) -> 0
```

La fonction interne `lme_aux` fait les appels récursifs que faisait `l_mul_naive`. Cette fonction interne est appelée dans

```
  try lme_aux linit
  with (E n) -> 0
```

On voit ici que `lme_aux` est appelée avec `linit`, la "liste initiale", à savoir l'argument de `l_mul_exc`. Cet appel est "surveillé" par l'intermédiaire de la construction `try.. with`.

Au moment où on rencontre le premier `0` dans la liste, le `if` a pour effet de *lever une exception*, en faisant `raise (E 5420)`. À ce moment-là, tous les appels en attente sont court-circuités, et on remonte d'un coup jusqu'au `try.. with`. L'effet de `with (E n) -> ..` est de *rattraper l'exception*. Ici, on rattrape l'exception, et on renvoie directement zéro.

Intuitivement, tout se passe comme si on passait directement de l'état

`2 * (4 * ★)`

(★ désigne l'endroit où a lieu le calcul) à l'état où l'on a renvoyé `0`. On est content, car l'exception permet de court-circuiter le retour aux appels récursifs en attente (représentés ci-dessus par `2 * (4 *`) pour renvoyer directement `0`, sans calculer `4*0` puis `2*0`.

On peut remarquer que dans le cas présent, l'entier qui est passé avec l'exception est inutile. Dans d'autres situations, ça peut être un entier utile à un calcul, ou bien un code correspondant à un type d'erreur, ou autre.

Si un calcul "protégé" par un `try.. with` ne provoque pas d'exception, alors le résultat est renvoyé directement, et la branche `with` n'est pas utilisée. Ce sera le cas si on calcule `l_mul_exc [2;4;6]`, par exemple.

Si une exception n'est pas rattrapée par un `try.. with`, comme par exemple dans

```
let f x = raise (E (x+1)) in f 3
```

on "se prend l'exception dans la figure". C'est un usage assez courant des exceptions, qui correspondent à des siuations catastrophiques (`failwith` est souvent employé ainsi).

1.2 Comment traiter les exceptions en fouine

Pour le rendu 3 en version intermédiaires, il faudra ajouter `raise` et `try.. with` à fouine, et exécuter les programmes qui utilisent ces constructions.

L'énoncé précise qu'on n'a pas le droit de s'appuyer sur les exceptions de Caml pour exécuter les exceptions fouine.

La première approche que vous pouvez envisager pour exécuter un programme fouine comportant des exceptions est d'étendre le type des valeurs. Cette extension peut être faite de manière plus ou moins élégante, elle peut en particulier se propager de manière pas très propre à tout le code.

Ce faisant, on "mime" les exceptions, mais le comportement d'un programme qu'on exécute avec fouine correspondra à la description donnée plus haut, avec la partie orange désolante. Mais c'est une solution qui marche.

Si vous vous sentez plus ambitieux, et plus à l'aise, vous pouvez utiliser l'idée des continuations pour exécuter les exceptions en fouine (Section 2.2). Attention : il ne s'agit pas de traduire un programme fouine avec exceptions vers un programme sans exceptions, pour ensuite l'exécuter avec fouine, mais bien d'ajouter des continuations à l'interprète fouine, dont le rôle est de gérer les exceptions.

2 (Intermédiares motivés et) Avancés : continuations

2.1 Programmation par continuations, principes

La *programmation par continuations* est un style de programmation où les fonctions ont toutes, intuitivement, un paramètre supplémentaire, qui est une fonction que l'on appelle pour renvoyer le résultat.

Ainsi, la définition ci-dessous de la fonction `f` à gauche devient, en style par continuation, la définition à droite :

```
let f = fun x -> x*x
```

```
let f = fun x -> fun k -> k (x*x)
```

ou, si l'on veut,

```
let f x k = k (x*x)
```

Ici `k` est la *continuation*. `k` est une fonction, qui représente le "futur du calcul" effectué par `f`. À titre d'intuition, alors que dans la version originale de `f` (à gauche), on renvoie le résultat de l'évaluation de `x*x`, comme s'il y avait un `return` implicite, avec les continuations, on appelle `k` en lui passant ce résultat.

On peut adopter systématiquement le style par continuations, ce qui donne, pour une fonction qui appelle une autre fonction :

```
let f x = x*(g x)
```

```
let f x k = g x (fun r -> k (x*r))
```

À droite, `f` passe directement la main à `g`, en lui fournissant la continuation `fun r -> k (x*r)`. Cette continuation se lit comme "lorsque `g` aura fini de calculer et renverra un résultat `r`, on multiplie par `x` et on envoie le résultat à `k`, la continuation qui avait été transmise lors de l'appel originel à `f`."

2.2 Implémenter les exceptions à l'aide de continuations

Vous pouvez prendre la fonction `eval` qui est le cœur de l'interprète fouine, et la réécrire en style par continuation.

En particulier, pour les appels récursifs, comme par exemple pour évaluer `e1 + e2`, on enchaîne les continuations, ce qui donne, à quelques détails près :

```
eval e2 env (fun n2 -> eval e1 env (fun n1 -> k (n1+n2)) )
```

où `k` est la continuation qui "attend" le résultat de l'évaluation de `e1+e2`.

Pour traiter les exceptions à l'aide de continuations, on travaille avec *deux continuations* : au lieu d'écrire `fun k ->`, on écrit `fun (k,kE) ->`. `k` est le futur du calcul, au sens usuel pour les continuations. `kE` est le "futur exceptionnel". Un calcul peut choisir entre répondre "tout va bien, et le résultat est `5`", ce qui se traduit par l'appel `k 5`, ou dire "il y a un problème, je lève une exception, que je décore avec l'entier `12`", ce qui se traduit par l'appel `kE 12`.

Dit autrement, la seconde continuation, `kE`, correspond au `try. .with` qui est en train de surveiller le calcul courant. Tant qu'on ne fait pas de `raise`, on répond à `k`. Pour faire un `raise`, on répond à `kE`.

2.3 Traduire pour éliminer les continuations

Une autre approche, qui est demandée dans une partie spécifique de l'énoncé du rendu 3 pour les binômes avancés, consiste à programmer une traduction (ou *compilation*) de (fouine+exceptions) vers fouine.

On adopte l'approche décrite à la partie 2.2, pas pour interpréter les programmes, mais pour engendrer un programme sans exceptions (et, soit dit en passant, fort illisible).

On présente ici les grandes lignes de la traduction, à vous d'affiner et de trouver le reste[1]. La traduction de `e`, notée ⟦e⟧, se fait par induction sur l'expression `e`. On décrit ci-dessous un certain nombre de cas intervenant dans cette définition.

- ⟦12⟧ = `fun k -> k 12`
  L'entier `12` est traduit en la fonction qui attend une continuation et lui envoie l'entier `12`.
- ⟦`(fun x -> e)`⟧ = `fun k -> k (fun x ->` ⟦e⟧`)`
  C'est le même principe. Comme une fonction est une valeur, on renvoie la valeur à `k`, sauf qu'il faut traduire cette valeur avant de l'envoyer (d'où le ⟦e⟧).
- ⟦`e1 + e2`⟧ = `fun k ->` ⟦`e2`⟧ `(fun v2 ->`
  ⟦`e1`⟧ `(fun v1 -> let v = v1+v2 in k v) )`
  Pour évaluer `e1+e2`, on évalue `e2`, en lui passant une continuation qui évaluera `e1`, puis renverra le résultat final à `k`, la continuation originelle.
  Le code ci-dessus peut aussi être écrit ainsi :

```
⟦e1 + e2⟧ = fun k ->
  let k2 = fun v2 ->
    let k1 = fun v1 ->
      let v = v1+v2 in k v in
    ⟦e1⟧ k1 in
  ⟦e2⟧ k2
```

- ⟦`let x = e1 in e2`⟧ = `fun k -> (` ⟦`e1`⟧ `(fun x ->` ⟦`e2`⟧ `k) )`
  On évalue `e1`, puis `e2`.
  On est astucieux en gardant le nom de variable `x` dans la version traduite. Ainsi, le résultat de l'évaluation de `e1` remplacera bien `x` dans `e2` au moment de passer à l'évaluation de `e2`.
- ⟦`x`⟧ = `fun k -> k x`
  La variable est traitée comme une valeur : on la passe directement à `k`. On peut remarquer que c'est cohérent, puisqu'une variable sera toujours substituée par une valeur.
- ⟦`e1 e2`⟧ = `fun k ->`
  ⟦`e2`⟧ `(fun v2 ->`
  ⟦`e1`⟧ `(fun v1 -> v1 v2 k) )`
  Pour l'application, c'est "presque" comme pour une addition : on évalue `e2`, puis `e1`, simplement on n'additionne pas les deux valeurs pour passer le résultat à `k`. À la place, on déclenche un calcul, en appliquant `v1` (qui est une fonction) à son argument `v2` et à la continuation `k`.
  Pour vous assurer que vous avez bien compris, relisez la traduction des fonctions, et vérifiez que "ça colle" lorsqu'on écrit `v1 v2 k`.

- **Gestion des exceptions.**
  Ce qui précède concerne la traduction par continuations du langage fonctionnel, sans exceptions.
  Afin de pouvoir traiter les exceptions, on doit en réalité adopter une transformation par continuations où l'on passe *un couple* de continuations, et non pas une seule continuation. Ce couple s'écrit typiquement

[1] Vous pouvez bien sûr nous contacter si vous avez des doutes.

`(k,kE)`, où `k` est la continuation normale, comme dans ce qui précède, et `kE` est la continuation exceptionnelle, que l'on invoque lorsque l'on veut lancer une exception.

Ainsi, l'opération `raise` se traduit de la manière suivante :

$[\![$`raise (E e)`$]\!]$ = `fun (k,kE) ->` $[\![$`e`$]\!]$ `((fun v -> kE v), kE)`

On évalue `e`, et on passe le résultat de cette évaluation à `kE`. À noter que l'on pourrait écrire cela de manière équivalente `fun (k,kE) ->` $[\![$`e`$]\!]$ `(kE,kE)` (prenez un peu de recul pour vous convaincre que ceci fait sens). On remarque donc que l'on passe deux fois la continuation `kE` : si l'évaluation de `e` lève une exception, elle est gérée par `kE`, et si ça n'est pas le cas, la valeur obtenue est transmise à `kE`. La continuation `k`, elle, est jetée à la poubelle (on pourrait donc écrire `fun (_,kE) ->` $[\![$`e`$]\!]$ `(kE,kE)` pour souligner cela).

À noter également que comme prévu, il n'y a pas de trace de `raise` dans le programme ainsi obtenu : on traduit vers fouine sans exceptions.

Ce qu'il reste à faire :

- on vous laisse compléter la définition suivante :
  $[\![$`try e1 with (E n) -> e2`$]\!]$ `= fun (k,kE) -> ..`
- il s'agit aussi de reprendre les définitions données plus haut afin de passer à chaque fois un couple de continuations.

2.4 Programmer la traduction

Le code produit par la traduction par continuations (appelée CPS pour *continuation passing style*) est rapidement illisible.

Vous pouvez faire usage de déclarations, comme dans l'explication de la traduction de la somme ci-dessus, pour rendre les choses un peu plus lisibles.

S'agissant de l'implémentation, il serait bon de ne pas recopier directement la version papier de la traduction dans un fichier `.ml`, avec moult `Fun(..)` et `Pair(..)`. Essayez de définir des outils vous permettant d'écrire de manière lisible les morceaux de code traduit.

Optimisations : réductions administratives. L'option `-opt` implémente une version spécialisée de la traduction. L'idée est la suivante : dans certains cas, $[\![$`e`$]\!]$ est une expression qui peut se réduire vers une expression `e'`, car on peut déjà faire certaines réductions β, dans la mesure où l'on a un argument qui est une valeur. C'est le cas en particulier si l'un des deux arguments (ou les deux) d'une application est une variable.

Il s'agit donc de :

- trouver des cas de ce style, pour lesquels on peut définir une version optimisée de la traduction ;
- se donner la possibilité de raffiner la traduction pour n'appliquer la version "bateau" que lorsqu'on ne peut pas faire autrement, et appliquer la version optimisée lorsque c'est possible ;
- documenter les optimisations que vous avez implémentées dans le README (ou dans un fchier à part, mentionné dans le README, si vous tenez absolument à faire du LaTeX).