TP n° 7

Paradigmes et interprétation
Licence Informatique
Université Côte d'Azur

Récursion paresseuse

On repart de l'interpréteur `letrec-mut.rkt` et on cherche à lui ajouter l'évaluation paresseuse de l'interpréteur `more-lazy.rkt`.

On souhaite que l'évaluation des expressions se fasse désormais de manière paresseuse. Pour cela, il faut introduire à nouveau les promesses. Celles-ci prendront la place des valeurs dans l'environnement. Comme les valeurs étaient mutables, les promesses doivent l'être aussi. On en déduit le nouveau type `Binding` :

```
(define-type Binding
  [bind (name : Symbol) (val : (Boxof Thunk))])
```

Pour l'implémentation de `letrec`, il n'est plus possible d'utiliser la valeur `(undefV)` dans l'environnement. À la place, on crée une nouvelle variante de promesse qui tiendra le même rôle :

```
(define-type Thunk
  [delay (e : Exp) (env : Env) (mem : (Boxof (Optionof Value)))]
  [undef]) ; promesse non definie
```

Achevez l'implémentation proposée.

Les parties suivantes sont globalement indépendantes.

Utilisation prématurée d'une valeur récursive

Avant d'effectuer le test qui suit, **sauvegarder votre travail** ! Que se passe-t-il si vous interprétez l'expression `{letrec {[x x]} x}` ?, retrouve-t-on la valeur `(undefV)` ? Si ce n'est pas le cas, apportez les modifications à l'interpréteur pour rétablir le comportement attendu.

Paires

On cherche à ajouter les paires à notre langage :

```
<Exp> ::=  ...
         | {pair <Exp> <Exp>}
         | {fst <Exp>}
         | {snd <Exp>}
```

L'expression `pair` construit la paire de valeurs obtenues par évaluation de ses deux sous-expressions. Les expressions `fst` et `snd` permettent de récupérer respectivement le premier et le second élément d'une paire. L'évaluation des paires se fait de manière paresseuse. Les arguments d'une expression `pair` ne sont évalués que si leur valeur est nécessaire (lors d'un appel à `fst` ou `snd`).

1. Modifiez le type `Exp` et la fonction `parse` pour prendre en charge les expressions `pair`, `fst` et `snd`.
2. Ajoutez une nouvelle variante de valeur `pairV` pour représenter une paire.
3. Modifiez la fonction `interp` pour gérer ces nouvelles expressions **avec une évaluation paresseuse**. Un nouveau type d'erreur `"not a pair"` sera lancé si les arguments de `fst` et `snd` ne s'évaluent pas à des paires.

```
(test (interp-expr
       `{letrec {[numbers-from {lambda {n}
                                 {pair n
                                       {numbers-from {+ n 1}}}}]}
          {let {[ints {numbers-from 0}]}
            {fst {snd {snd {snd ints}}}}}})
      (numV 3))
```

Récursion croisée

On veut désormais étendre `letrec` pour implémenter la récursion croisée. L'expression autorise plusieurs définitions et chacune peut faire référence à tout ou partie des autres.

```
Expr ::= ...
       | {letrec {[<Sym> <Expr>]+} <Expr>}
```

Généralisez ce qui a été fait en début de TP pour autorisez des définitions mutuellement récursives. Un exemple d'utilisation classique est donné ci-dessous.

```
(test (interp-expr
       '{letrec {[even? {lambda {n} {if n
                                        {odd? {- n 1}}
                                        1}}]
                 [odd? {lambda {n} {if n
                                       {even? {- n 1}}
                                       0}}]}
          {even? 5}})
      (numV 0))
```

Vous trouverez ci-dessous un exemple non trivial de ce qui est réalisable en combinant `letrec` et les listes paresseuses.

```
(test (interp-expr
       `{letrec
            {; curryfied map2 on infinite lists
             [map2 {lambda {f}
                     {lambda {l1}
                       {lambda {l2}
                         {pair {{f {fst l1}} {fst l2}}
                               {{{map2 f} {snd l1}} {snd l2}}}}}}}]
             ; curryfied list-ref
             [list-ref {lambda {l}
                         {lambda {n}
                           {if n
                               {{list-ref {snd l}} {- n 1}}
                               {fst l}}}}]
             ; curryfied addition function
             [add {lambda {x} {lambda {y} {+ x y}}}]
             ; infinite fibonacci sequence !!!
             ; (list 0 1 1 2 3 5 8 13 ... )
             [fibo {pair 0
                         {pair 1
                               {{{map2 add} fibo} {snd fibo}}}}]}
          {{list-ref fibo} 7}})
      (numV 13))
```