TP n° 8

Paradigmes et interprétation
Licence Informatique
Université Côte d'Azur

Dans ce TP, on utilise un nouvel l'interpréteur appelé `store-k.rkt`. Il s'agit du même interpréteur que `let-cc.rkt` dans lequel on a ajouté les états explicites (comme dans `variable.rkt`). On retrouve ainsi les types `Location`, `Storage` et `Store` vus au cours n°5.

La gestion de la mémoire est grandement facilitée par l'architecture imposée par les continuations. Elle est simplement passée (après une éventuelle écriture) entre les différents appels à `interp` et `continue`. Ainsi le type `Result` n'est plus nécessaire et a disparu. Toutes ces modifications sont déjà réalisées et vous êtes invités à vous familiariser avec elles avant de commencer le TP.

Vous allez devoir implémenter plusieurs expressions dans ce cadre. Quelques règles générales s'appliquent. Pour chaque nouvelle expression, il vous faut :

1. ajouter une variante au type `Exp`,
2. modifier la fonction `parse` pour la prendre en compte,
3. ajouter des continuations pour représenter les étapes de calcul :
   — une pour `set!`, `begin` et `if`,
   — deux pour `while`,
   — aucune pour `break`.
4. modifier `interp` et `continue` en conséquence.

Prenez particulièrement garde à ce que les fonctions `interp` et `continue` demeurent récursives terminales. Autrement dit, un seul appel récursif par cas est autorisé.

Mutation

Ajoutez la mutation au langage. La sémantique est identique à celle de `variable.rkt`.

```
<Exp> ::=  ...
         | {set! <Symbol> <Exp>}
         | {begin <Exp> <Exp>}

(test (interp-expr `{let {[x 0]}
                      {begin {set! x 1}
                             x}})
      (numV 1))
```

Branchement conditionnel

Pour le reste du TP, on considère que `(numV 0)` est une représentation du booléen à faux et que tout autre valeur est une représentation du booléen vrai. Ajoutez l'expression `if` au langage.

```
<Exp> ::=  ...
         | {if <Exp> <Exp> <Exp>}

(test (interp-expr `{if 1 2 3}) (numV 2))
(test (interp-expr `{if 0 2 3}) (numV 3))
```

Boucle tant que

Ajoutez la boucle `while` dans le langage. L'expression `{while cnd body}` évalue `body` tant que `cnd` est vraie. La valeur de l'expression est toujours `(numV 0)` : la boucle est utilisée uniquement pour ses effets de bord.

```
<Exp> ::=  ...
         | {while <Exp> <Exp>}

(test (interp-expr `{while 0 x}) (numV 0))
(test (interp-expr `{let {[fac
                           {lambda {n}
                             {let {[res 1]}
                               {begin
                                 {while n ; tant que n est non nul
                                        {begin
                                          {set! res {* n res}}
                                          {set! n {+ n -1}}}}
                                 res}}}]}
                     {fac 6}})
      (numV 720))
```

Il est temps de faire un break

L'expression `break` permet de sortir (lire **s'échapper**) de la boucle `while` courante. Lorsque `break` apparaît dans le **corps** (pas la condition) d'un `while`, on quitte la boucle courante. On reprend le calcul avec l'expression à évaluer après la boucle. La valeur en sortie de boucle est `(numV 0)` comme si elle s'était terminée normalement.

Inspirez-vous de ce qui a été fait avec les gestionnaires d'erreurs pour implémenter l'expression `break`. On produit une erreur `"break outside while"` si `break` est utilisée en dehors d'une boucle.

```
<Exp> ::=  ...
         | break ; mot cle special
```

```
(test (interp-expr `{while 1 break}) (numV 0))
(test/exn (interp-expr `break) "break outside while")
(test/exn (interp-expr `{while break 1}) "break outside while")
(test (interp-expr `{let {[n 10]}
                      {begin
                        {while n ; tant que n est non nul
                          {if {- n 5}
                              {set! n {- n 1}} ; si n n'egale pas 5
                              break}} ; si n egale 5
                        n}})
      (numV 5))
```