TP n° 6

Paradigmes et interprétation
Licence Informatique
Université Côte d'Azur

Présentation de l'interpréteur

Dans ce TP, on travaille sur un nouvel interpréteur : `lambda.rkt`. Il s'agit d'un interpréteur basé sur le lambda-calcul. On détaille ci-dessous son fonctionnement.

Langage de haut niveau et analyse syntaxique

Les expressions du langage s'écrivent dans une syntaxe usuelle. La grammaire utilisée est la suivante :

```
<ExpS> ::= <Number>
         | <Symbol>
         | <Constant>
         | {lambda {<Symbol>+} <ExpS>}
         | {<ExpS> <ExpS>+}
         | {let {[<Symbol> <ExpS>]+} <ExpS>}
         | {letrec {[<Symbol> <ExpS>]} <ExpS>}
         | {if <ExpS> <ExpS> <ExpS>}

<Constant> ::= true | false | zero? | add1 | sub1 | + | - | * | /
             | pair | fst | snd
```

La représentation du langage est faite dans le type `ExpS`. L'analyse syntaxique est donnée dans la fonction `parse : (S-Exp -> ExpS)`.

Langage interne et interpréteur

Le langage interne est le lambda-calcul. Il n'y a donc que les identificateurs, la lambda-abstraction et l'application de fonction. Les parenthèses servent à forcer la priorité dans les applications successives (normalement associatives à gauche).

```
<Exp> ::= <Symbol>
        | λ<Symbol>.<Exp>
        | <Exp> <Exp>
        | (<Exp>)
```

Les termes sont représentés par le type `Exp`. Ils seront évalués dans la fonction `interp`. Leur évaluation se faisant par substitution, il n'y a pas d'environnement et le résultat est un terme réduit. Pour éviter de boucler sur certaines expressions faiblement normalisables, on ne réalise les β-réductions qu'en tête de terme. Par exemple, le terme `(λx.x) y` sera réduit en `y` mais le terme `λy.(λx.x) y` ne sera pas réduit (alors qu'il est normalisable en `λy.y`).

La fonction `interp` est fournie. De plus, comme l'évaluation d'un terme donne un terme, des fonctions utilitaires vous sont données pour afficher les résultats :

- `expr->string` renvoie une représentation du terme passé en paramètre.
  `(expr->string (lamE 'x (lamE 'y (idE 'y))))` renvoie `"λx.λy.y"`.
- `expr->number` renvoie le nombre représenté par le terme passé en paramètre ou renvoie une erreur `"not a number"` si le terme n'est pas la représentation valide d'un nombre.
  `(expr->number (lamE 'x (lamE 'y (idE 'y))))` renvoie `0`.
- `expr->boolean` renvoie le booléen représenté par le terme passé en paramètre ou renvoie une erreur `"not a boolean"` si le terme n'est pas la représentation valide d'un booléen.
  `(expr->boolean (lamE 'x (lamE 'y (idE 'y))))` renvoie `#f`.
- `interp-number` et `interp-boolean` encapsulent les deux fonctions précédentes pour qu'elles puissent prendre une s-expression en argument.
  `(interp-number `{lambda {x} {lambda {y} y}})` renvoie `0`.
  `(interp-boolean `{lambda {x} {lambda {y} y}})` renvoie `#f`.

Passage du langage de haut niveau au langage interne

Pour passer d'une expression du langage de haut niveau à un terme, on appelle la fonction `desugar : (ExpS -> Exp)`. Son rôle est de retirer le sucre syntaxique pour obtenir un terme. Dans l'interpréteur qui vous est fourni, seul les cas les plus simples sont implémentés. L'intégralité du TP consiste à ajouter les cas manquants. **Vous ne devez modifier que cette fonction !**

Conseils :

- La plupart des ajouts au langage correspondent à la définition de constantes.
- Pour les autres, définissez des fonctions utilitaires spécialisées.
- Le retrait du sucre syntaxique peut se faire par couches. Par exemple, transformez les `let` en applications de fonctions à plusieurs paramètres dans un premier temps. Transformez ensuite les fonctions à plusieurs paramètres et les appels à plusieurs arguments par curryfication.

Travail à effectuer

Curryfication

Curryfiez les fonctions à plusieurs paramètres. Il y a toujours au moins un paramètre.

```
(test
  (expr->string (desugar (parse `{{lambda {x y z} body} a b c})))
  "(λx.λy.λz.body) a b c")
```

Profitez-en pour ajouter le `let`. On peut lier simultanément plusieurs identificateurs.

```
(test
  (expr->string (desugar (parse `{let {[x a] [y b] [z c]} body})))
  "(λx.λy.λz.body) a b c")
```

Arithmétique

Ajoutez les entiers naturels au langage ainsi que l'incrément, l'addition et la multiplication.

```
(test (interp-number `10) 10)
(test (interp-number `{add1 1}) 2)
(test (interp-number `{+ 1 2}) 3)
(test (interp-number `{* 3 4}) 12)
```

Booléens

Ajoutez les booléens au langage ainsi que le branchement conditionnel `if`. Vous pouvez supposer que les identificateurs commençant par un underscore (caractère `_`) sont réservés.

```
(test (interp-boolean `true) #t)
(test (interp-boolean `false) #f)
(test (interp-number `{if true 0 1}) 0)
(test (interp-number `{if false 0 1}) 1)
```

Profitez-en pour ajouter le prédicat `zero?`.

```
(test (interp-boolean `{zero? 0}) #t)
(test (interp-boolean `{zero? 1}) #f)
```

Paires

Ajoutez les paires au langage.

```
(test (interp (desugar (parse `{fst {pair a b}}))) (idE 'a))
(test (interp (desugar (parse `{snd {pair a b}}))) (idE 'b))
```

Profitez-en pour ajouter le décrement et la soustraction au langage.

```
(test (interp-number `{sub1 2}) 1)
(test (interp-number `{- 4 2}) 2)
(test (interp-number `{- 1 2}) 0)
```

Récursion

Implémentez la récursion dans le langage.

```
(test (interp-number
       `{letrec {[fac {lambda {n}
                        {if {zero? n}
                            1
                            {* n {fac {- n 1}}}}}}]}
          {fac 6}})
      720)
```

Bonus : la division

En utilisant les expressions définies précédemment, implémentez la division. Vous pouvez considérer que la division euclidienne de m par n peut se calculer par $Div(m, n) = DivInter(m, n, n)$ où

$$DivInter(m, n, k) = \begin{cases} 1 + DivInter(m, n, n) & \text{si } k = 0 \\ 0 & \text{si } m = 0 \\ DivInter(m - 1, n, k - 1) & \text{sinon} \end{cases}$$