Paradigmes et interprétation
Examen du 01 mars 2019

Licence Informatique 3e année
Université Nice Sophia Antipolis

Durée: 2h00
Tous documents autorisés

Ce sujet est basé sur une variante de l'interpréteur `variable.rkt` dont le code est donné en annexe. Les changements apportés sont les suivants :

— La liaison locale (`let`) n'est plus implémentée par sucre syntaxique.
— L'appel par référence a été retiré pour simplifier l'interpréteur.

Le sujet est composé de deux parties indépendantes. Lorsqu'une question vous demande d'ajouter une variante dans un type ou de modifier le traitement d'une variante dans une fonction, ignorez les autres variantes dans votre réponse. Représentez l'omission par `...` dans le code que vous écrivez.

1 La boucle while

On cherche à implémenter la boucle `while` dans l'interpréteur.

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

Question 1
Ajoutez une variante `whileC` dans le type `ExprC` pour représenter la nouvelle expression.

Question 2
Modifiez la fonction `parse` pour réaliser l'analyse syntaxique de la nouvelle expression.

Question 3
Pour interpréter l'expression `{while cnd body}`, on évalue `body` tant que `cnd` est vraie. On considère que faux est représentée par `(numV 0)`. Tout autre valeur est considérée comme vraie. Modifiez `interp` pour prendre en compte la nouvelle expression. La valeur d'une boucle `while` est `(numV 0)` indépendamment de `cnd` et `body`.

Question 4
On considère l'expression suivante :

```
{let {[res 0]}
  {begin {let {[v-cnd cnd]}
           {let {[not-cnd 1]}
             {begin {while v-cnd
                           {begin {set! v-cnd {set! not-cnd 0}}
                                  {set! res if-true}}}
                    {while not-cnd
                           {begin {set! not-test 0}
                                  {set! res if-false}}}}}}}
         res}}
```

(a) Identifiez les variables libres de l'expression.

(b) Exprimez plus simplement ce que fait l'expression en fonction des variables libres.

(c) Déduisez-en comment implémenter le branchement conditionnel par sucre syntaxique sur la boucle `while`.

Question 5
Rappelez comment implémenter l'expression {`and e1 e2`} par sucre syntaxique à partir du branchement conditionnel.

Question 6
Rappelez comment implémenter l'expression {`and e1 e2 ... eN`} par sucre syntaxique à partir du `and` binaire. L'expression {`and e1 e2 ... eN`} réalise la conjonction de toutes les expressions.

Question 7
Comment procéderiez-vous pour retirer les différentes couches de sucre syntaxique ?

2 Changement de représentation de l'environnement

On suppose désormais que la boucle `while`, le branchement conditionnel `if` et les connecteurs booléens `and`, `or` et `not` sont dans le langage.

Le but de cette partie est de modifier la représentation de l'environnement. L'environnement sera représenté par une liste non-vide de dictionnaires mutables. Chaque dictionnaire contient un ensemble de noms de variables associés à des adresses mémoires. Toutes les variables d'un même dictionnaire correspondent à une portée du code. Le dictionnaire en tête de liste correspond à la portée locale tandis que le dernier dictionnaire correspond à la portée globale.

Un dictionnaire est une table d'association de type `(hashof 'a 'b)`. Voici les primitives associées et des exemples d'utilisations.

```
; crée un dictionnaire à partir d'une liste de paires clé / valeur
make-hash : ((listof ('a * 'b)) -> (hashof 'a 'b))
; renvoie une option contenant la valeur associée à une clé ou (none) si la
; clé est invalide
hash-ref : ((hashof 'a 'b) 'a -> (optionof 'b))
; ajoute ou modifie une association clé / valeur dans le dictionnaire
hash-set! : ((hashof 'a 'b) 'a 'b -> void)

(define d1 (make-hash empty)) ; un dictionnaire vide
(define d2 (make-hash (list (pair 'a 10)))) ; un dictionnaire où 'a est lié à 10
(hash-ref d1 'a) ; renvoie (none)
(hash-ref d2 'a) ; renvoie (some 10)
(hash-set! d1 'a 20) ; 'a est désormais lié à 20 dans d1
(hash-ref d1 'a) ; renvoie (some 20)
```

On change donc la définition de l'environnement en :

```
(define-type-alias Env (listof (hashof symbol Location)))
```

Tous les environnements que vous manipulerez contiendront au moins un dictionnaire (celui de la portée globale).

Question 8
Modifiez la fonction `extend-env` pour qu'elle supporte la nouvelle représentation de l'environnement. L'association est ajoutée dans la portée locale.

Question 9
Modifiez la fonction `lookup` pour qu'elle supporte la nouvelle représentation de l'environnement. On cherche tout d'abord la liaison dans la portée locale. Si on ne la trouve pas, on recommence avec l'éventuelle portée englobante (la suivante dans la liste) ou on échoue.

Question 10
Une nouvelle portée est ajoutée à l'environnement uniquement lors d'un appel de fonction. Dans `interp`, donnez le nouvel environnement dans lequel on évalue le corps d'une fonction.

On cherche désormais à ajouter l'expression `global` au langage.

```
<Expr> ::= ...
         | {global <Sym> <Expr>}
```

L'expression `{global id expr}` indique que `id` désigne une variable de la portée globale dans `expr`. Une erreur `"free identifier"` se produit si `id` n'est pas définie dans la portée globale. On suppose que `ExprC` et `parse` ont été correctement mis à jour.

```
(define-type ExprC
  ...
  [globalC (id : symbol) (body : ExprC)])
```

Question 11

Définissez une fonction `get-global-dic : (Env -> (hashof symbol Location))` qui prend en paramètre un environnement et renvoie le dictionnaire correspondant à la portée globale.

Question 12

(a) Rappelez le principe de l'appel par référence.

(b) En déduire une manière d'implémenter l'expression `global` dans `interp`.

Question 13

(a) Rappelez la différence entre portée lexicale et portée dynamique.

(b) Initialement, l'interpréteur utilisait une portée lexicale. Le changement de représentation de l'environnement change-t-il cela ? Et l'introduction de l'expression `global` ? Justifiez votre réponse dans chacun des cas.

```
#lang plai-typed
(require plai-typed/s-exp-match)

(define-syntax-rule (with [(v-id sto-id) call] body)
  (type-case Result call
    [v*s (v-id sto-id) body]))

;;; Définition des types ;;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;

; Représentation des expressions
(define-type ExprC
  [numC (n : number)]
  [idC (s : symbol)]
  [plusC (l : ExprC) (r : ExprC)]
  [multC (l : ExprC) (r : ExprC)]
  [letC (id : symbol) (rhs : ExprC) (body : ExprC)]
  [lamC (par : symbol) (body : ExprC)]
  [appC (fun : ExprC) (arg : ExprC)]
  [setC (s : symbol) (val : ExprC)]
  [beginC (l : ExprC) (r : ExprC)])

; Représentation des valeurs
(define-type Value
  [numV (n : number)]
  [closV (par : symbol) (body : ExprC) (env : Env)])

; Représentation du résultat d'une évaluation
(define-type Result
  [v*s (v : Value) (s : Store)])

; Représentation des liaisons
(define-type Binding
  [bind (name : symbol) (location : Location)])

; Manipulation de l'environnement
(define-type-alias Env (listof Binding))
(define mt-env empty)
(define extend-env cons)

; Représentation des adresses mémoire
(define-type-alias Location number)

; Représentation d'un enregistrement
(define-type Storage
  [cell (location : Location) (val : Value)])

; Manipulation de la mémoire
(define-type-alias Store (listof Storage))
(define mt-store empty)
(define override-store cons)

;;; Analyse syntaxique ;;;
;;;;;;;;;;;;;;;;;;;;;;;;;;

(define (parse [s : s-expression]) : ExprC
  (cond
    [(s-exp-match? `NUMBER s) (numC (s-exp->number s))]
    [(s-exp-match? `SYMBOL s) (idC (s-exp->symbol s))]
    [(s-exp-match? '{+ ANY ANY} s)
     (let ([sl (s-exp->list s)])
       (plusC (parse (second sl)) (parse (third sl))))]
    [(s-exp-match? '{* ANY ANY} s)
     (let ([sl (s-exp->list s)])
       (multC (parse (second sl)) (parse (third sl))))]
    [(s-exp-match? '{let {[SYMBOL ANY]} ANY} s)
     (let ([sl (s-exp->list s)])
       (let ([subst (s-exp->list (first (s-exp->list (second sl))))])
         (letC (s-exp->symbol (first subst))
               (parse (second subst))
               (parse (third sl)))))]
    [(s-exp-match? '{lambda {SYMBOL} ANY} s)
     (let ([sl (s-exp->list s)])
       (lamC (s-exp->symbol (first (s-exp->list (second sl)))) (parse (third sl))))]
    [(s-exp-match? '{set! SYMBOL ANY} s)
     (let ([sl (s-exp->list s)])
       (setC (s-exp->symbol (second sl)) (parse (third sl))))]
    [(s-exp-match? '{begin ANY ANY} s)
     (let ([sl (s-exp->list s)])
       (beginC (parse (second sl)) (parse (third sl))))]
    [(s-exp-match? '{ANY ANY} s)
     (let ([sl (s-exp->list s)])
       (appC (parse (first sl)) (parse (second sl))))]
    [else (error 'parse "invalid input")]))
```

```
;;; Interprétation ;;;
;;;;;;;;;;;;;;;;;;;;;

; Interpréteur
(define (interp [e : ExprC] [env : Env] [sto : Store]) : Result
  (type-case ExprC e
    [numC (n) (v*s (numV n) sto)]
    [idC (s) (v*s (fetch (lookup s env) sto) sto)]
    [plusC (l r)
           (with [(v-l sto-l) (interp l env sto)]
                 (with [(v-r sto-r) (interp r env sto-l)]
                       (v*s (num+ v-l v-r) sto-r)))]
    [multC (l r)
           (with [(v-l sto-l) (interp l env sto)]
                 (with [(v-r sto-r) (interp r env sto-l)]
                       (v*s (num* v-l v-r) sto-r)))]
    [letC (id rhs body)
          (with [(v-rhs sto-rhs) (interp rhs env sto)]
                (let ([l (new-loc sto-rhs)])
                  (interp body
                          (extend-env (bind id l) env)
                          (override-store (cell l v-rhs) sto-rhs))))]
    [lamC (par body) (v*s (closV par body env) sto)]
    [appC (f arg)
          (with [(v-f sto-f) (interp f env sto)]
                (type-case Value v-f
                  [closV (par body c-env)
                         (with [(v-arg sto-arg) (interp arg env sto-f)]
                               (let ([l (new-loc sto-arg)])
                                 (interp body
                                         (extend-env (bind par l) c-env)
                                         (override-store (cell l v-arg) sto-arg))))]
                  [else (error 'interp "not a function")]))]
    [setC (var val)
          (let ([l (lookup var env)])
            (with [(v-val sto-val) (interp val env sto)]
                  (v*s v-val (override-store (cell l v-val) sto-val))))]
    [beginC (l r)
            (with [(v-l sto-l) (interp l env sto)]
                  (interp r env sto-l))]))

; Fonctions utilitaires pour l'arithmétique
(define (num-op [op : (number number -> number)]
                [l : Value] [r : Value]) : Value
  (if (and (numV? l) (numV? r))
      (numV (op (numV-n l) (numV-n r)))
      (error 'interp "not a number")))

(define (num+ [l : Value] [r : Value]) : Value
  (num-op + l r))

(define (num* [l : Value] [r : Value]) : Value
  (num-op * l r))

; Recherche d'un identificateur dans l'environnement
(define (lookup [n : symbol] [env : Env]) : Location
  (cond
    [(empty? env) (error 'lookup "free identifier")]
    [(equal? n (bind-name (first env))) (bind-location (first env))]
    [else (lookup n (rest env))]))

; Renvoie une adresse mémoire libre
(define (new-loc [sto : Store]) : Location
  (+ (max-address sto) 1))

; Le maximum des adresses mémoires utilisés
(define (max-address [sto : Store]) : Location
  (if (empty? sto)
      0
      (max (cell-location (first sto)) (max-address (rest sto)))))

; Accès à un emplacement mémoire
(define (fetch [l : Location] [sto : Store]) : Value
  (cond
    [(empty? sto) (error 'interp "segmentation fault")]
    [(equal? l (cell-location (first sto))) (cell-val (first sto))]
    [else (fetch l (rest sto))]))
```