Solution aux exercices d'Olivier Danvy aux JFLAs 2014

Matthieu Sozeau

January 10, 2014

Chapter 1

Library Exercises_danvy_jfla

danvy-ex1.txt exercices pour JFLA 2014 8 janvier 2014 Olivier Danvy <danvy@cs.au.dk>

Module ARITH.

---

1. Les expressions arithmétiques

Expressions:
n : int
e : expression e ::= n | e - e
p : programme p ::= e
Exemples:
p0 = 0 p1 = 10 - 6 p2 = (10 - 6) - (7 - 2) p3 = (7 - 2) - (10 - 6) p4 = 10 - (2 - 3)
Valeurs:
v : valeur v ::= n
Valeurs expressibles:
ve : valeur_expressible ve ::= v
Interprétation:

---

n => n
e1 => n1 e2 => n2 n1 - n2 = n3

---

e1 - e2 => n3
On interprète le programme e en la valeur n si le jugement
e => n
est satisfait.
Exercice 0: Programmer l'interprète ci-dessus (disons en OCaml) et le tester sur les exemples.

évalue_0 : expression -> valeur_expressible interprète_0 : programme -> valeur_expressible

```
Require Import Arith.

Definition val := nat.

Inductive expr :=
| cst : nat → expr
| minus : expr → expr → expr.

Definition prog := expr.

Coercion cst : nat >-> expr.

Fixpoint eval (e : expr) : val :=
  match e with
    | cst n ⇒ n
    | minus x x' ⇒ eval x - eval x'
  end.

Definition interp (p : prog) : val := eval p.
Delimit Scope expr with expr.
Notation " x - y " := (minus x y) : expr.
Bind Scope expr with expr.
Example p0 : expr := 0.
Example p1 : expr := (10 - 6)%expr.
Example p2 : expr := ((10 - 6) - (7 - 2))%expr.
Example p3 : expr := ((7 - 2) - (10 - 6))%expr.
Example p4 : expr := (10 - (2 - 3))%expr.

Notation check x y := (refl_equal : interp x = y) (only parsing).

Check (check p0 0).
Check (check p1 4).
Check (check p2 0).
Check (check p3 1).
Check (check p4 10).
```

Exercice 1: CPS-transformer (en appel par valeur, de gauche à droite) la fonction évalue_0, et l'appeler dans la fonction principale de l'interprète avec comme continuation initiale la fonction identité.

évalue_1 : expression -> (valeur_expressible -> 'a) -> 'a interprète_1 : programme -> valeur_expressible

```
Fixpoint eval_1 {A} (e : expr) (k : val → A) : A :=
  match e with
    | cst n ⇒ k n
    | minus x x' ⇒
      eval_1 x (fun xv ⇒
```

```
                    eval_1 x' (fun xv' ⇒ k (xv - xv')))
  end.
Definition interp_1 (p : prog) : val := eval_1 p (fun x ⇒ x).
Notation check_1 x y := (refl_equal : interp_1 x = y) (only parsing).
Check (check_1 p0 0).
Check (check_1 p1 4).
Check (check_1 p2 0).
Check (check_1 p3 1).
Check (check_1 p4 10).
```

Exercice 2: Défonctionaliser la continuation de évalue_1.

cont ::= ...

continue_2 : cont -> valeur -> valeur évalue_2 : expression -> cont -> valeur interprète_2 : programme -> valeur

Le type de données (“data type”) cont représente la grammaire des contextes.

Les deux fonctions mutuellement récursives évalue_2 et continue_2 implémentent une machine abstraite, c’est à dire un système de transition d’états.

```
Inductive cont :=
| C0 : cont
| C1 : expr → cont → cont
| C2 : val → cont → cont.
Fixpoint apply_cont (cnt : nat) (k : cont) (n : val) :=
  match cnt with 0 ⇒ 0 | S cnt ⇒
  match k with
    | C0 ⇒ n
    | C1 v' k' ⇒ eval_2 cnt v' (C2 n k')
    | C2 n' k ⇒ apply_cont cnt k (n' - n)
  end
  end

with eval_2 (cnt : nat) (e : expr) (k : cont) : val :=
  match cnt with 0 ⇒ 0 | S cnt ⇒
    match e with
      | cst n ⇒ apply_cont cnt k n
      | minus x x' ⇒ eval_2 cnt x (C1 x' k)
    end
  end.
Definition apply_cont2 := apply_cont 30.
Definition eval_2' := eval_2 30.
Definition interp_2 (p : prog) : val := eval_2' p C0.
Notation check_2 x y := (refl_equal : interp_2 x = y) (only parsing).
```

```
Check (check_2 p0 0).
Check (check_2 p1 4).
Check (check_2 p2 0).
Check (check_2 p3 1).
Check (check_2 p4 10).

End ARITH.

Module ARITHERROR.
```

---

2. Les expressions arithmétiques, avec des erreurs

Expressions:
n : nat
e : terme e ::= n | e - e
p : programme p ::= e
Exemples:
p0 = 0 p1 = 10 - 6 p2 = (10 - 6) - (7 - 2) p3 = (7 - 2) - (10 - 6) p4 = 10 - (2 - 3)
Valeurs:
v : valeur v ::= n
Erreurs:
s : erreur
Valeurs expressibles:
ve : valeur_expressible ve ::= v | s
Interprétation:

---

n => n
e1 => s

---

e1 - e2 => s
e1 => n1 e2 => s

---

e1 - e2 => s
e1 => n1 e2 => n2 n1 < n2

---

e1 - e2 => “underflow”
e1 => n1 e2 => n2 n1 >= n2 n1 - n2 = n3

---

e1 - e2 => n3
On interprète le programme e en la valeur n si le jugement
e => n
est satisfait, et en l’erreur s si le jugement

e => s
est satisfait.
Exercice 0: Programmer l'interprète ci-dessus et le tester sur les exemples.
évalue_0 : terme -> valeur_expressible interprète_0 : programme -> valeur_expressible

```
Require Import Arith.
Require Import Arith.
Require Import String.

Definition val := nat.

Definition error := string.

Inductive expr :=
| cst : nat → expr
| minus : expr → expr → expr.

Inductive expr_val :=
| value (v : val)
| exn (e : error).

Definition prog := expr.

Coercion cst : nat >-> expr.

Definition ret := value.

Definition bind (x : expr_val) (f : val → expr_val) : expr_val :=
  match x with
    | value v ⇒ f v
    | exn _ ⇒ x
  end.

Fixpoint eval (e : expr) : expr_val :=
  match e with
    | cst n ⇒ ret n
    | minus x x' ⇒
      bind (eval x) (fun v ⇒
      bind (eval x') (fun v' ⇒
        if NPeano.ltb v v' then
          exn ("Underflow"%string)
        else
          ret (v - v')))
  end.

Definition interp (p : prog) : expr_val := eval p.
Notation " x - y " := (minus x y) : expr.
Delimit Scope expr with expr.
Bind Scope expr with expr.
Example p0 : expr := 0.
```

```
Example p1 : expr := (10 - 6)%expr.
Example p2 : expr := ((10 - 6) - (7 - 2))%expr.
Example p3 : expr := ((7 - 2) - (10 - 6))%expr.
Example p4 : expr := (10 - (2 - 3))%expr.

Notation check_ok x y := (refl_equal : interp x = value y) (only parsing).
Notation check_error x := (refl_equal : interp x = exn ("Underflow"%string)) (only parsing).

Check (check_ok p0 0).
Check (check_ok p1 4).
Check (check_error p2).
Check (check_ok p3 1).
Check (check_error p4).
```

Exercice 1: CPS-transformer (en appel par valeur, de gauche à droite) la fonction évalue_0, et l'appeler dans la fonction principale de l'interprète avec comme continuation initiale la fonction identité.

évalue_1 : terme -> (valeur_expressible -> 'a) -> 'a interprète_1 : programme -> valeur_expressible

```
Definition bind' (f : val → expr_val) (x : expr_val) : expr_val :=
  bind x f.
Definition catch_error {A} v (k : expr_val → A) (f : val → A) :=
  match v with
    | exn e ⇒ k v
    | value v ⇒ f v
  end.

Fixpoint eval_1 {A} (e : expr) (k : expr_val → A) : A :=
  match e with
    | cst n ⇒ k (ret n)
    | minus x x' ⇒
      eval_1 x (fun v ⇒
        catch_error v k
          (fun v ⇒
            eval_1 x' (fun v' ⇒
                         catch_error v' k
                                     (fun v' ⇒
                                        if NPeano.ltb v v' then
                                          k (exn ("Underflow"%string))
                                        else
                                          k (ret (v - v'))))))
  end.
```

Exercice 2: Diviser la continuation valeur_expressible -> 'a en deux: (valeur -> 'a) * (erreur -> 'a) et adapter évalue_1 et interprète_1.

évalue_2 : terme -> (valeur -> 'a) -> (erreur -> 'a) -> 'a interprète_2 : programme -> valeur_expressible

```
Fixpoint eval_2 {A} (e : expr) (s : val → A) (k : error → A) : A :=
  match e with
    | cst n ⇒ s n
    | minus x x' ⇒
      eval_2 x
          (fun v ⇒
              eval_2 x' (fun v' ⇒
                              if NPeano.ltb v v' then
                                k ("Underflow"%string)
                              else
                                s (v - v')) k) k
  end.

Definition interp_2 (e : prog) := eval_2 e value exn.

Notation check_ok2 x y := (refl_equal : interp_2 x = value y) (only parsing).
Notation check_error2 x := (refl_equal : interp_2 x = exn ("Underflow"%string)) (only
parsing).

Check (check_ok2 p0 0).
Check (check_ok2 p1 4).
Check (check_error2 p2).
Check (check_ok2 p3 1).
Check (check_error2 p4).
```

Exercice 3: Spécialiser le co-domaine des continuations et de la fonction d'évaluation pour qu'il ne soit plus polymorphique mais qu'il soit valeur_expressible, et court-circuiter la seconde continuation pour ne pas continuer en cas d'erreur.

évalue_3 : terme -> (valeur -> valeur_expressible) -> valeur_expressible interprète_3 : programme -> valeur_expressible

(NB. Maintenant il n'y a plus qu'une continuation et elle n'est appliquée que s'il n'y a pas eu d'erreur.)

```
Fixpoint eval_3 (e : expr) (s : val → expr_val) : expr_val :=
  match e with
    | cst n ⇒ s n
    | minus x x' ⇒
      eval_3 x
          (fun v ⇒
              eval_3 x' (fun v' ⇒
                              if NPeano.ltb v v' then
                                exn ("Underflow"%string)
                              else
                                s (v - v')))
```

```
  end.

Definition interp_3 (e : prog) := eval_3 e value.

Notation check_ok3 x y := (refl_equal : interp_3 x = value y) (only parsing).
Notation check_error3 x := (refl_equal : interp_3 x = exn ("Underflow"%string)) (only
parsing).

Check (check_ok3 p0 0).
Check (check_ok3 p1 4).
Check (check_error3 p2).
Check (check_ok3 p3 1).
Check (check_error3 p4).
```

Exercice 4: Défonctionaliser la continuation de évalue_3.

cont ::= ...

continue_4 : cont -> valeur -> valeur_expressible évalue_4 : terme -> cont -> valeur_expressible interprète_4 : programme -> valeur_expressible

Le type de données (“data type”) cont représente la grammaire des contextes. (NB. A-t-il changé par rapport à l’exercice précédent?)

Les deux fonctions mutuellement récursives évalue_4 et continue_4 implémentent une machine abstraite, c’est à dire un système de transition d’états, qui s’arrête soit tout de suite en cas d’erreur, soit à la fin du calcul.

```
Inductive cont :=
| C0
| C1 : expr → cont → cont
| C2 : val → cont → cont.

Fixpoint apply_cont (cnt : nat) (k : cont) (n : val) :=
  match cnt with 0 ⇒ exn ("Out-of-fuel"%string) | S cnt ⇒
  match k with
    | C0 ⇒ value n
    | C1 v' k' ⇒ eval_4 cnt v' (C2 n k')
    | C2 n' k ⇒ if NPeano.ltb n' n then
                  exn ("Underflow"%string)
                else
                  apply_cont cnt k (n' - n)
  end
  end

with eval_4 (cnt : nat) (e : expr) (k : cont) : expr_val :=
  match cnt with 0 ⇒ exn ("Out-of-fuel"%string) | S cnt ⇒
    match e with
      | cst n ⇒ apply_cont cnt k n
      | minus x x' ⇒ eval_4 cnt x (C1 x' k)
    end
```

```
  end.
Definition interp_4 (e : prog) := eval_4 50 e C0.
Notation check_ok4 x y := (refl_equal : interp_4 x = value y) (only parsing).
Notation check_error4 x := (refl_equal : interp_4 x = exn ("Underflow"%string)) (only
parsing).
Check (check_ok4 p0 0).
Check (check_ok4 p1 4).
Check (check_error4 p2).
Check (check_ok4 p3 1).
Check (check_error4 p4).
End ArithError.
```

Le lambda-calcul

Termes:
n : int
x : identificateur
t : terme t ::= x | \x.t | t t
p : programme p ::= t où t est fermé (i.e., sans variables libres)

```
Require Import String.
Notation ident := string.
Require Import List.
Generalizable Variables t u v w.
Definition env (A : Set) := list (ident × A).
Require Import Ascii Bool.
Require Import Natural.Binary.NBinary.
Fixpoint string_eq (s s' : string) : bool :=
  match s, s' with
    | EmptyString, EmptyString ⇒ true
    | String a s, String a' s' ⇒ (N_of_ascii a =? N_of_ascii a')%N && string_eq s s'
    | _, _ ⇒ false
  end.
Fixpoint lookup {A} (e : env A) (x : ident) :=
  match e with
    | (x', a) :: l ⇒ if string_eq x x' then Some a else lookup l x
    | nil ⇒ None
  end.
Delimit Scope term with term.
```

```
Inductive term : Set :=
| var (x : ident)
| abs (x : ident) (t : term)
| app (t : term) (u : term).
Bind Scope term with term.

Coercion var : ident >-> term.
Definition var_of_string (s : string) : term := s.
Coercion var_of_string : string >-> term.

Notation " 'λ' x ',' t " := (abs x%string t%term) (at level 30) : term.
Notation " '@' ( t , u ) " := (app t%term u%term) (at level 20, u at next level) : term.

Inductive support (Γ : env unit) : term → Prop :=
| support_app : ‘(support Γ t → support Γ u → support Γ (app t u))
| support_abs x : ‘(support ((x, tt) :: Γ) t → support Γ (abs x t))
| support_var x v : lookup Γ x = Some v → support Γ (var x).

Definition closed (t : term) := support nil t.

Definition prog := { x : term | closed x }.
```

Exemples:
p0 = (\x.x) p1 = (\x.x) (\x.x) p2 = (\x.\f.f x) (\y.y) (\x.x) `Delimit Scope string with str`.

```
Example p0 : term := (λ "x" , "x"%string)%term.
Example p1 : term := @( p0 , p0 )%term.
Example p2 : term :=
  @(@((λ "x" , λ "f" , @("f"%string, "x"%string)), p0), p0)%term.
Eval compute in p2.
```

Valeurs et environnements:
e : environnement e ::= nil | (identificateur, valeur) :: environnement
v : valeur v ::= (\x.t, e)
Opérateur algébrique:
lookup : identificateur -> environnement -> valeur

```
Inductive value : Set :=
| value_intro : ident → term → env value → value.
```

Contextes d'évaluation:
C ::= □ | $C\ (t,\ e)$ | $v\ C$

```
Inductive context : Set :=
| nilC
| funC (C : context) (t : term) (e : env value)
| argC (v : value) (C : context).
```

Machine abstraite (dite “CEK”):

```
    <x, e, C>\_eval -> <C, v>\_cont
                       où v = lookup x e
 <\ x.t, e, C>\_eval -> <C, (\ x.t, e)>\_cont
<t0 t1, e, C>\_eval -> <t0, e, [C (t1, e)]>

           <[], v>\_cont -> v
   <[C (t, e)], v>\_cont -> <t, e, [v C]>\_eval
<[(\ x.t, e) C], v>\_cont -> <t, (x, v) :: e, C>\_eval
```

Exercice 0: Programmer cette machine abstraite.

```
Fixpoint eval cnt (t : term) (e : env value) (C : context) :=
  match cnt with 0 ⇒ value_intro "outofbounds"%string t e | S cnt ⇒
  match t with
    | var x ⇒ match (lookup e x) with
                | None ⇒ value_intro "freevar" t e
                | Some v ⇒ cont cnt C v
              end
    | abs x t ⇒ cont cnt C (value_intro x t e)
    | app t0 t1 ⇒ eval cnt t0 e (funC C t1 e)
  end
  end

with cont cnt (C : context) (v : value) :=
  match cnt with 0 ⇒ v | S cnt ⇒
  match C with
    | nilC ⇒ v
    | funC C t e ⇒ eval cnt t e (argC v C)
    | argC (value_intro x t e) C ⇒ eval cnt t ((x, v) :: e) C
  end
  end.

Definition eval' t := eval 6000 t nil nilC.

Definition envtest : env unit := (("x"%string, tt) :: nil).
Eval compute in lookup envtest "x"%string.

Eval compute in eval' p0.
Eval compute in eval' p1.
Eval compute in eval' p2.
```

Exercice 1: Cette machine abstraite est en forme défonctionalisée. La refonctionaliser.

```
Fixpoint eval2 cnt (t : term) (e : env value) (k : value → value) :=
  match cnt with 0 ⇒ value_intro "outofbounds"%string t e | S cnt ⇒
  match t with
    | var x ⇒ match (lookup e x) with
```

```
                | None ⇒ value_intro "freevar" t e
                | Some v ⇒ k v
              end
    | abs x t ⇒ k (value_intro x t e)
    | app t0 t1 ⇒
      eval2 cnt t0 e (fun val ⇒
                        let 'value_intro x t e := val in
                        eval2 cnt t1 e (fun v1 ⇒
                                          eval2 cnt t ((x, v1) :: e) k))
  end
  end.
Eval compute in eval2 50 p2 nil (fun x ⇒ x).
```

Exercice 2: Le résultat de l'Exercice 1 est en CPS. L'exprimer en style direct.

```
Fixpoint eval3 cnt (t : term) (e : env value) :=
  match cnt with 0 ⇒ value_intro "outofbounds"%string t e | S cnt ⇒
  match t with
    | var x ⇒ match (lookup e x) with
                | None ⇒ value_intro "freevar" t e
                | Some v ⇒ v
              end
    | abs x t ⇒ value_intro x t e
    | app t0 t1 ⇒
      let 'value_intro x t e := eval3 cnt t0 e in
      let v1 := eval3 cnt t1 e in
        eval3 cnt t ((x, v1) :: e)
  end
  end.
Eval compute in eval3 50 p2 nil.
```

Exercice 3: Le résultat de l'Exercice 2 est en forme défonctionalisée (dans le sens que les fermetures sont en forme défonctionalisée triviale). Le refonctionaliser, et caractériser le résultat.

```
Inductive func : Set :=
| func_intro : ident → term → env func → func.
Fixpoint eval4 cnt (t : term) (e : env func) :=
  match cnt with 0 ⇒ func_intro "outofbounds"%string t nil | S cnt ⇒
  match t with
    | var x ⇒ match (lookup e x) with
                | None ⇒ func_intro "unbound variable" t nil
                | Some v ⇒ v
              end
    | abs x t' ⇒ func_intro x t' e
```

```
  | app t0 t1 ⇒
    let 'func_intro x t e' := eval4 cnt t0 e in
    let v1 := eval4 cnt t1 e in
      eval4 cnt t ((x, v1) :: e')
end
end.
```

C'est la forme normale de tête du terme.

```
Eval compute in eval4 50 p2 nil.
```