プログラム言語論 補足資料

亀山, 2010/05/10

1 ラムダ式 (1 引数の関数)

- $f(x) = e$ となる関数 f のことを $\lambda x.e$ と書く。
- $\lambda x.e$ では、x は局所変数 (外からは見えない変数)。
- $(\lambda x.e)(x) = e$ が成立する。
- ML の一種である OCaml 言語では、$\lambda x.e$ を `fun` x `->` e と書く。

2 miniML の構文の定義

変数 x と定数 c の構文は、miniC と同じ (定数は、整数定数と真偽値定数)。

$$
\begin{aligned}
\text{式}\quad e ::= {} & x \mid c \mid e + e' \mid e * e' \mid e = e' \mid e > e' \\
& \mid \texttt{if}\ e\ \texttt{then}\ e'\ \texttt{else}\ e'' \mid \texttt{fun}\ x\ \texttt{->}\ e \mid e\ e' \\
& \mid \texttt{let}\ x = e\ \texttt{in}\ e' \mid \texttt{let rec}\ x\ y = e\ \texttt{in}\ e' \\
& \mid \texttt{print}\ e \mid \texttt{show}\ e \\
& \mid (e, e') \mid \texttt{fst}\ e \mid \texttt{snd}\ e \\
& \mid (e)
\end{aligned}
$$

関数適用 (関数呼出し):

- $e_1\ e_2$ と、ただ並べて書く。(括弧をつけてもよい。)
- e_1 の計算結果が、関数 `fun` x `->` e_0 になったら、その関数を、e_2 という引数で呼び出し、その結果を返す。
- (`fun` x `->` $x + 1$)$(2 + 3)$ は、6 になる。

let 式 (`let` $x = e$ `in` e')

- $x = e$ という (局所的な) 環境のもとで e' を計算する。
- (`fun` x `->` e')e と同じ計算になる。

let rec 式 (`let rec` $fx = e$ `in` e')

- e' の中で f を使ってよい (再帰呼び出し)。

注意すべき点: 関数は 1 引数のものしかない。変数に値を代入することはできない。C 言語の for 文や while 文はなく、関数の再帰呼び出しを使ってループ構造を実現する。

3 miniML の意味の定義

「式」しかない事と、「状態変化」がない事により、miniC の意味論よりだいぶ簡単になる。

3.1 値

計算結果を意味する「値」(あたい、value) を以下のように定義する。

$$\text{値 } v ::= x \mid c \mid (\texttt{fun } x \texttt{ -> } e,\ \sigma)$$

ここで、$(\texttt{fun } x \texttt{ -> } e, \sigma)$ というのが不思議な値である。関数型言語では、計算結果が「関数」になることがあり、それを後で使えるようにするため、関数本体と、その関数を計算したときの環境とをセットにしたものをクロージャ (closure、閉包) と呼ぶ。クロージャがなぜ必要かについては、講義を参照のこと。

3.2 式の意味

環境 σ のもとで、式 e を計算した結果を $\sigma(e)$ と書く。これを e に関する場合分けで定義する。

- $e = x$ のとき: $\sigma(e)$ は、σ における x の値である。
- $e = c$ のとき: $\sigma(e) = c$ である。
- $e = e_1 + e_2$ のとき: $\sigma(e) = \sigma(e_1) + \sigma(e_2)$ である。
- $e = \texttt{if } e_1 \texttt{ then } e_2 \texttt{ else } e_3$ のとき: $\sigma(e_1) = \texttt{true}$ のとき、$\sigma(e) = \sigma(e_1)$ で、$\sigma(e_1) = \texttt{false}$ のとき、$\sigma(e) = \sigma(e_2)$ である。それ以外のとき、$\sigma(e)$ は値を持たない。
- $e = (\texttt{let } x = e_1 \texttt{ in } e_2)$ のとき、$e = (\texttt{fun } x \texttt{ -> } e_2)e_1$ と見なして計算する。(`let rec` は、少し面倒)
- $e = \texttt{fun } x \texttt{ -> } e_1$ のとき：$\sigma(e) = (\texttt{fun } x \texttt{ -> } e_1,\ \sigma)$ である。
- $e = e_1\ e_2$ のとき：
  - $\sigma(e_1) = (\texttt{fun } x \texttt{ -> } e_3,\ \sigma_3)$ かつ、$\sigma(e_2) = v$ となるとき: $\sigma_4 = ((x = v) \oplus \sigma_3)$ とおいた上で、$\sigma(e) = \sigma_4(e_3)$ である。
  - 上記のようにならないとき、$\sigma(e)$ は値を持たない。

以上。(print と let-rec の意味は省略した)

4 miniML のインタープリタ

インタープリタは、意味論をそのまま反映したプログラムである。miniC のインタープリタは、意味論が複雑なことに対応して結構複雑であった。(文や式の定義を相互再帰的に呼んでいた。) 一方、miniML の方は、意味論が非常にすっきりしているので、対応してインタープリタもだいぶ簡単である。ほとんど上記の定義をそのまま書くだけ、といった感じになる。詳細は、演習の際に学習してほしい。

5 「対」のデータ

式の構文の最後の行：

$$\text{式 } e, f ::= \cdots \mid (e, f) \mid \texttt{fst } e \mid \texttt{snd } e$$

直感的には、(e, f) は e と f の計算結果 v_1, v_2 を「対 (つい、ペア)」にしたデータのことである。(要素数 2 の配列、あるいは、C 言語の struct 型で 2 要素の構造体を作ったときのデータと似ている。)

`fst` e と `snd` e は、それぞれ、対の第 1、第 2 要素である。

式の意味の追加：

- $e = (e_1, e_2)$ のとき: $\sigma(e) = (\sigma(e_1), \sigma(e_2))$ である。
- $e =$ `fst` e_1 のとき: $\sigma(e_1) = (v_2, v_3)$ ならば、$\sigma(e) = v_2$ であり、それ以外のときは、値を持たない。
- $e =$ `snd` e_1 のとき: $\sigma(e_1) = (v_2, v_3)$ ならば、$\sigma(e) = v_3$ であり、それ以外のときは、値を持たない。

6 print と show

- `print` e は、e の計算結果 (値) を印刷する。
- `show` e は、e の値によって、「スタックの印刷」か「ヒープの印刷」を行なう。(e が 0 のとき前者、1 のとき後者。)

```
print (1,(2,fun x-> x+1));; (プリント文はどんな値でも適用可能)
show 0 ;; (スタックの表示)
show 1 ;; (ヒープの表示)
```

7 MinML の例題

・let 式を使った簡単な例:

```
let x = 1 in
  let y = 2 in
    x + y

let f = (fun x -> x + 1) in
  f (f 3)

(let x = 1 in
   x + x)
+ 5

(fun x -> if x = 1 then 2 else 3) 1

(fun x -> if x = 1 then 2 else 3) 10
```

・ブロック構造，変数のスコープ

```
let x=5 in
  (let f=(fun y->x+y) in
    let x=10 in
     (f 20) + x * 2)
  + x * 3

let rec f x =
     if x=0 then 1
     else x * f (x-1)) in
  (f 10)
```

・関数をデータとして扱う

```
let f = (fun x -> fun y -> x + y) in
  f 3 5

let f = (fun x -> fun y -> x + y) in
  (f 3) 5

let f = (fun x -> fun y -> x + y) in
  let g = f 3 in
    (g 5) + (g 7)

(fun f -> f (f 3)) (fun x -> x + 1)

(fun f -> fun x -> f (f x)) (fun y -> y + 1) 3
```

・値呼び，名前呼び

```
let f = fun x -> (print x; 2) in
  (fun y -> y + y + y) (f 1)

let f = fun x -> (print x; 2) in
  (fun y -> (y 1) + (y 1) + (y 1)) f

let f = fun x -> (print 1; 5) in
  let g = fun x -> (print 2; 7) in
    f g
```

・対 (ペア) を表すデータ型

```
(1,2)

fst (1,true)

fst (snd (1,(true,3)))

(fun x -> fun y -> ((x,y),(y,x))) 10 true

(fun x -> (fst x) ((snd x) 10))
   ((fun z -> z * 3), (fun y -> y + 2))
```

参考: OCaml の使い方について: インターネット上に解説テキストがいくつかあるので検索してみるとよい。本家 (フランス INRIA 研究所) の解説は、`http://caml.inria.fr/pub/docs/manual-ocaml/` である。また、日本語での教科書も 3 冊 (それぞれの著者は、京都大学五十嵐淳氏、お茶の水女子大学浅井健一氏、OCaml-Nagoya グループ) 出版されている。