プログラミング入門

「環境モデル」

二宮 崇（ ninomiya@cs.ehime-u.ac.jp ）

教科書

Structure and Interpretation of Computer Programs, 2nd Edition: Harold Abelson, Gerald Jay Sussman, Julie Sussman, The MIT Press, 1996

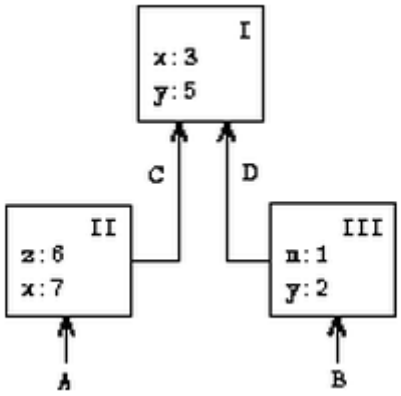
標準部品化力、オブジェクトおよび状態（第３章の続き）

3.2 評価の環境モデル

今までの置き換えモデルではうまく説明できないことがわかりました。そこで、環境モデルと呼ぶモデルについてこれから考えていきます。

環境モデルは、環境とフレームから構成されます。各フレームは、変数に割り当てられた値を保持する表（変数の束縛表）になっていて、環境はフレームの並びになります。ある環境である変数 x を参照するときにその環境が指すフレームを順に参照していき最初に出会った変数 x の値を変数 x の値とします。そのため、環境はフレームの並びと考えられるわけです。具体的には、環境はフレームのリストと考えればよく、フレームは変数と値の連想リストと考えれば良いです。例えば、環境 E は（F1　F2　…　Fn）というリストで、各 Fi は（(x 3) (y 5) (z 2)）という連想リストとなるわけです。環境 E において変数を探すときは、前から順に F1, F2, …とフレームの並び順に探していくことになります。また、環境 E の cdr 部分にあたる環境 E’=(F2 F3 … Fn)は E の外側の環境（enclosing environment）と呼ばれ、一番外側の環境は大域環境と呼ばれます。

環境モデルは四角と矢印を使った図でよく表現されます。例えば、次のような環境 ABCD を考えてみましょう。



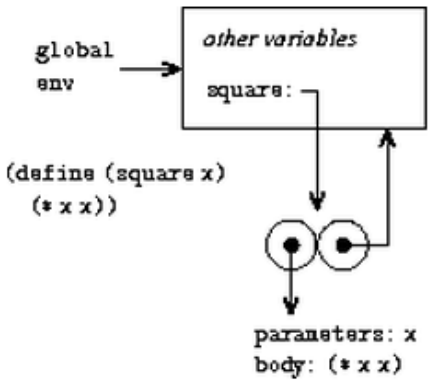
各四角の箱がフレームを表していて、それぞれに変数の値が格納されています。また、各フレームは別フレームを指すポインタをもっており、指されているフレームは外側の環境になります。ただし、一番外側の環境である大域環境は他のフレームを指すポインタをもちません。上の図では、ABCD は環境へのポインタとなっています。変数 x を環境 B で探すときは、まずフレーム III をみます。ここには x に対応する値がみつからないので、外側のフレームに進んでフレーム I をみます。ここで、x に対応する値 3 がみつかるので、環境 B では x の値は 3 となります。一方、環境 A では、最初にフレーム II をみます。ここで、x に対応する値 7 がみつかるので、環境 A における x の値は 7 となります。

3.2.1 評価の規則

評価の順番は今までどおりで、まず、組合せの部分式を評価し、それから、引数を演算子に適用します。ただ、このとき、置き換えモデルでの評価を環境モデルでの評価に置き換えます。

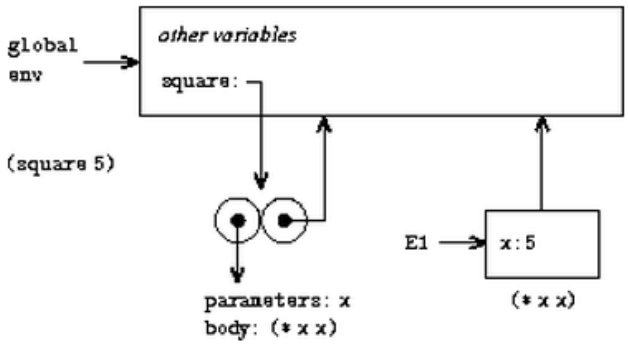
環境モデルでは、関数はコードと環境へのポインタのペアになっていて、lambda 式を評価することによって得られます。また、関数は常に lambda 式から得られる、とします。例えば、(define (square x) (* x x))は(define square (lambda (x) (* x x)))のシンタックスシュガーとして定義されるので、関数のための define は必要なく、lambda 式を変数に束縛しさえすればよい、ということになります。インタプリタが lambda 式を処理する際、(lambda (x) (* x x))を処理することを「lambda 式を評価する」と呼び、((lambda (x) (* x x) 5)を処理して実際に計算を行うことを「関数を評価する」「関数を適用する」「関数を実行する」と呼びます。関数(=コードと環境のペア)は、lambda 式を評価することによって得られますが、lambda 式を評価した環境がそ

の関数の環境となって、コードは lambda 式の本文から得られます。例えば、(define (square x) (* x x))に対しては、次の環境モデルが得られます。



lambda 式は、パラメータと本体として定義され、関数の環境は、square を作った環境を指すようになります。また、このコードと環境のペアで表現された関数は関数オブジェクトと呼ばれます。また、define は現在の環境フレームで変数を束縛する、ということを行います。set!はその環境における変数を探して新しい値に変更することを行います。set!しようとした変数が束縛されてなければエラーが返ってきます。

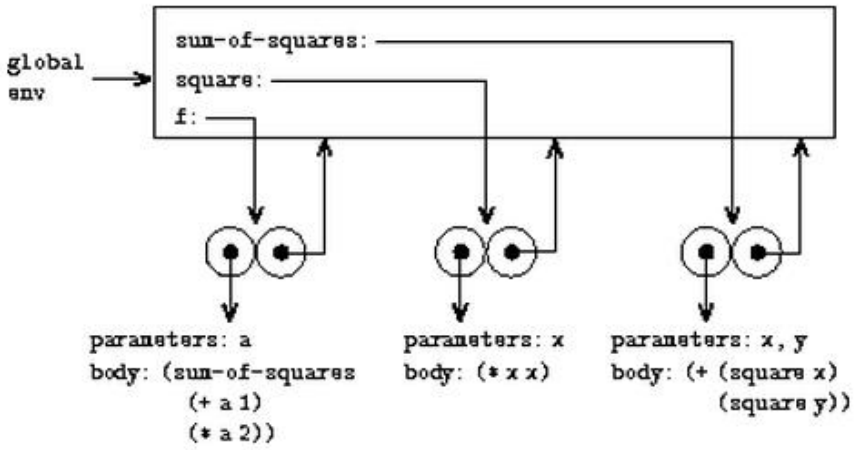
関数の作り方の次は、関数の評価の方法について説明します。関数に引数を適用させるには、まず、パラメータを引数の値に束縛する新しいフレームを作ります。この新しいフレームの外部環境は関数が持つ環境を指します。そして、この新しい環境で、関数を実行します。例えば、(square 5)を実行すると次のような環境が作られます。



この E1 という新しい環境の中では x は 5 であるから、(square 5)は 25 を返すようになります。E1 の外部環境は、square が持つ環境と同じになるので、この場合は square と同様大域環境を指すようになります。また、このように新しい環境を作ることによって局所変数が実現されていて、外で定義されている x と干渉することなく、関数が実行されることがわかります。

3.2.2 単純な関数の作用

もう一つ例をみてみましょう。



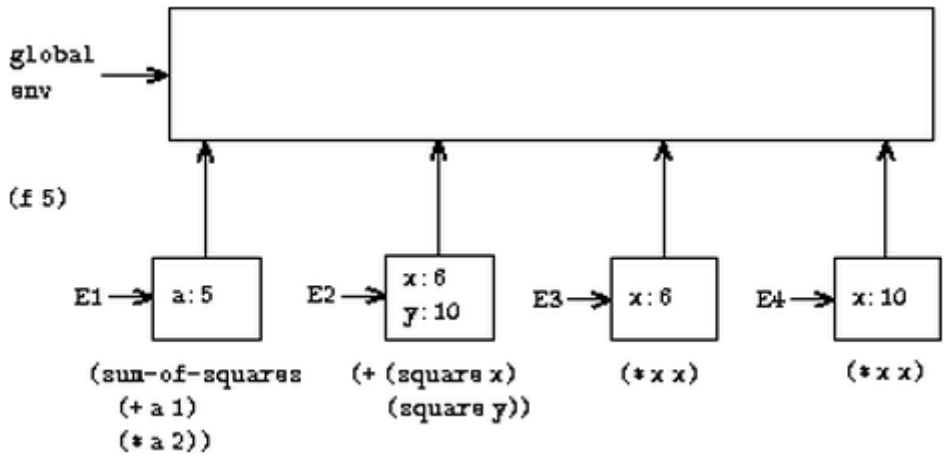
これは、

```
(define (square x) (* x x))

(define (sum-of-squares x y) (+ (square x) (square y)))

(define (f a) (sum-of-squares (+ a 1) (* a 2)))
```

を環境モデルで表現したものです。ここで、(f 5)を評価すると、次のようになります。



まず、E1 というフレームがつくられて、a に 5 が束縛されます。ここで、(+ a 1)と(* a 2)が実行されて、(sum-of-squares 6 10)が実行されます。E1 からみているので、大域環境

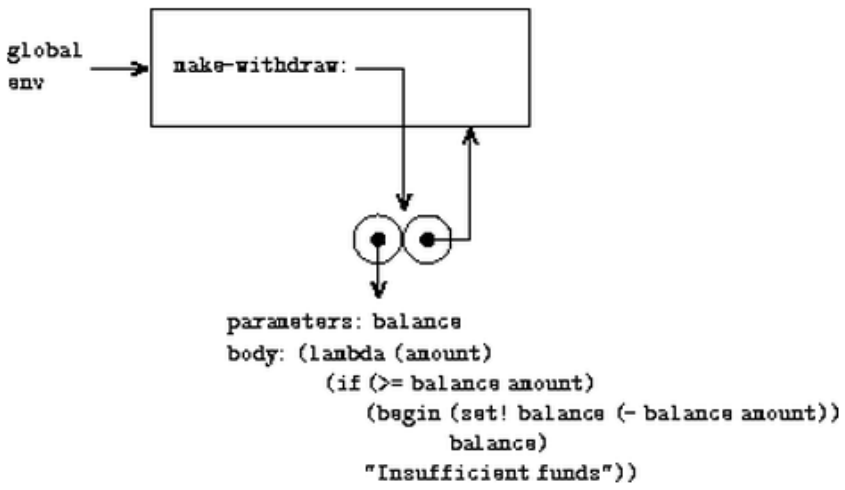
の sum-of-squares が実行されます。sum-of-squares の実行で、E2 というフレームがつくられて、x に 6、y に 10 が束縛され、(+ (squares 6) (squares 10))が実行されます。まず、(square 6)が実行され、E3 という環境で x に 6 を束縛し、36 という結果を得ます。続いて、(square 10)が実行され、E4 という環境で x に 10 を束縛し、100 という結果を得ます。これによって、(+ 36 100)=136 の結果が得られるようになります。

3.2.3 局所変数の入れ物としてのフレーム

次に、元々の環境モデルを導入した動機になっていた代入について考えてみましょう。次の make-withdraw についてみてみましょう。

```
(define (make-withdraw balance)
   (lambda (amount)
      (if (>= balance amount)
          (begin (set! balance (- balance amount))
                 balance)
          "Insufficient funds")))
```

次の図は、大域環境でこの関数を定義した結果を示しています。



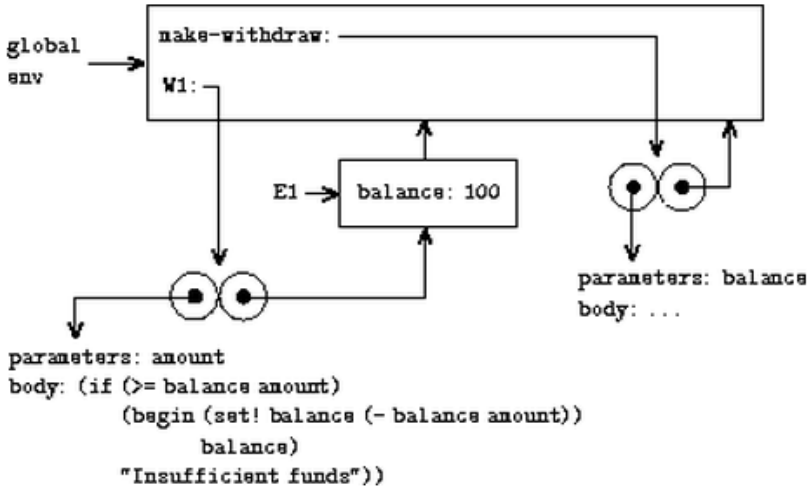
これに対し、

```
(define W1 (make-withdraw 100))
```

と続いて、

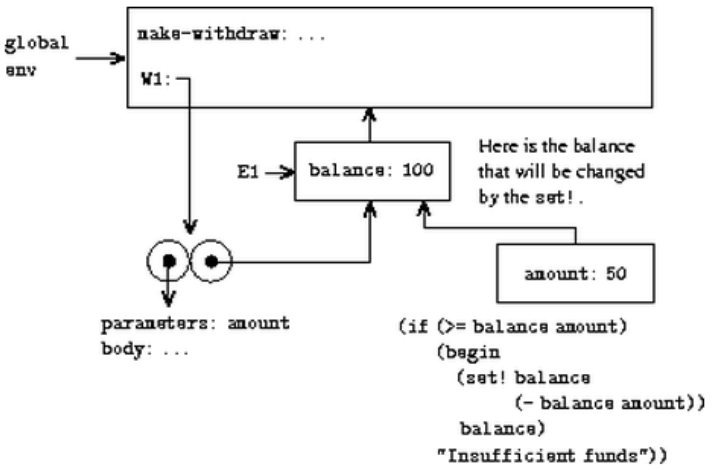
(W1 50)

とした場合について考えてみます。まず、make-withdraw を実行します。次の図をみてください。

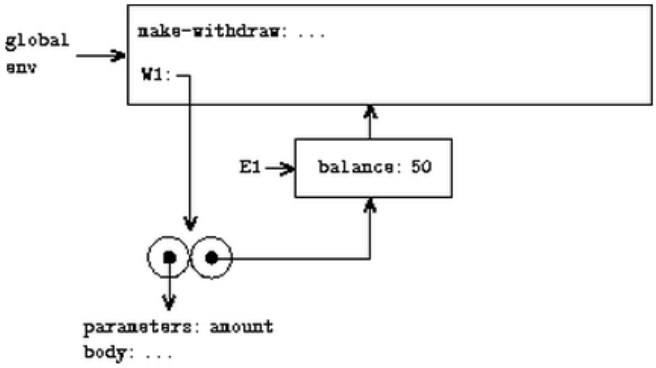


ここで、まず、make-withdraw を実行することにより、新しい環境 E1 が作られて、そこに balance という変数が用意され、引数の値 100 が束縛されます。この新しい環境で make-withdraw が実行されることに注意すると、lambda 式で表現された関数はその新しい環境を自身の環境として指すようになります。最後に、この新しい関数が返されて、大域環境の W1 に束縛されるようになります。

続いて、(W1 50)を実行するとどうなるかみてみましょう。その環境モデルは次のようになります。



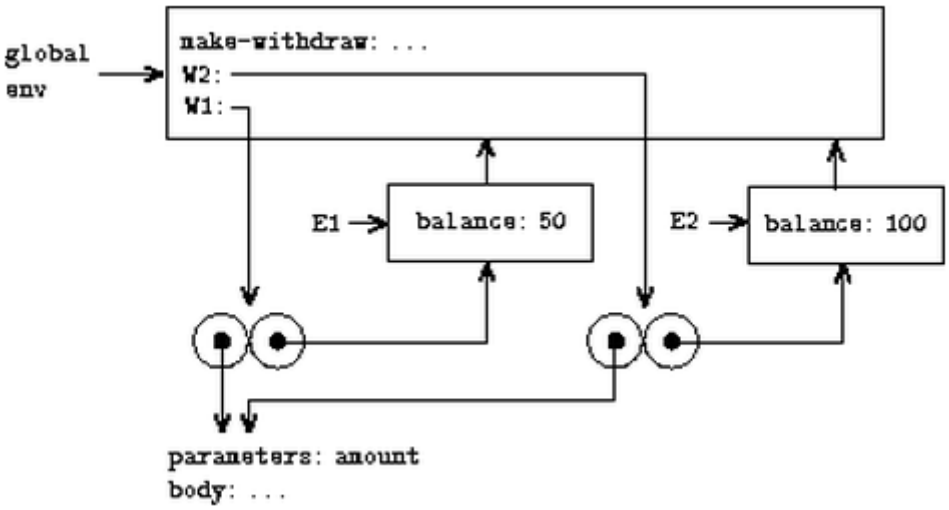
まず、W1 に束縛されている関数をみてみましょう。(W1 50)を実行することにより新しいフレームが作られますが、このフレームは W1 の関数の環境を外部環境として指すようになるので、上の図のように、amount を持つ新しいフレームは E1 が指しているフレームを指すようになります。この結果、新しい環境で実行される関数は、W1 が指している関数が持つ balance を set!で書き換えることができるようになっています。W1 呼び出し後の状況は次のようになります。



ここで、もう一つ別の払い出しオブジェクトを作ったとしましょう。

(define W2 (make-withdraw 100))

すると、次のようになって、別々の balance の値をもつことができるようになることがわかります。

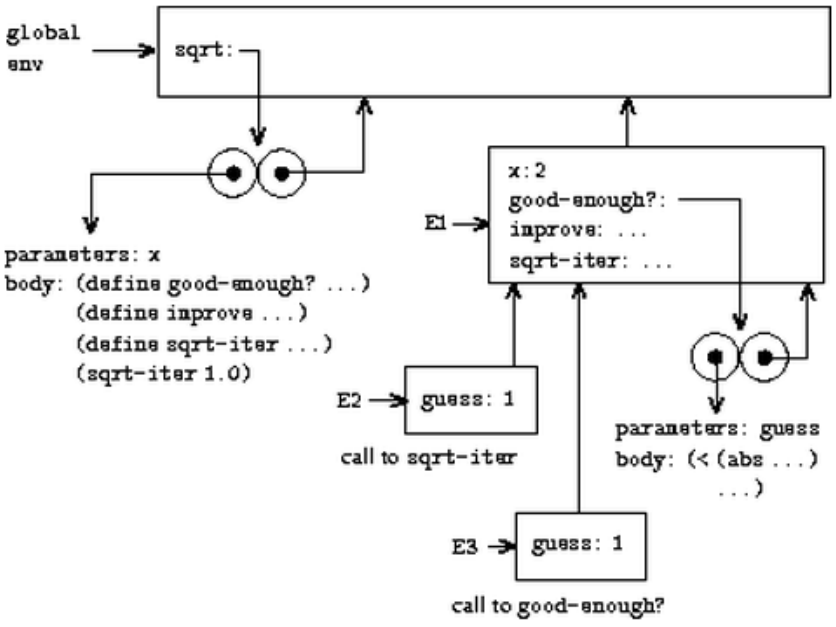


3.2.4 内部定義

続いて、内部定義がどのようになるのかみてみましょう。例として、次のプログラムをみてみましょう。

```
(define (sqrt x)
  (define (average x y) (/ (+ x y) 2))
  (define (square x) (* x x))
  (define (good-enough? guess) (< (abs (- (square guess) x)) 0.001))
  (define (improve guess) (average guess (/ x guess)))
  (define (sqrt-iter guess)
    (if (good-enough? guess)
        guess
        (sqrt-iter (improve guess))))
  (sqrt-iter 1.0))
```

次の図は、(sqrt 2)の評価中で、局所関数 good-enough?が guess を 1 にし、一回目に呼び出されたところを示しています。



まず、大域環境にある sqrt が呼び出されると、新しい環境 E1 がつくられ、パラメータ x は 2 に束縛されます。次に E1 で本体が実行されます。まず、define によって、good-enough?や improve や sqrt-iter などの内部定義が作られます。注意するのは、ここで、環境 E1 にこれらの関数が定義されている、ということです。局所関数を定義したあと、(sqrt-iter 1.0)が E1 の中で評価されます。ここで、また新しい環境 E2 が作られ、guess に 1 が束縛され、そこで、sqrt-iter が実行されます。sqrt-iter の中で、最初に good-enough?が実行されるため、また新しく環境 E3 が作られます。ここでもまた、guess に 1 という値が束縛され、good-enough?が実行されます。ただし、E3 が指す外部環境は E2 ではなく E1 であることに注意してください。good-enough?を実行する、ということは、E1 で定義されている good-enough?を実行することになるのですが、この good-enough?が指している環境は E1 になっているので、E3 も E1 を指すようになるのです。

まとめると次のことが言えます。

・局所関数の名前は関数が実行されるときの環境で束縛されるので、大域環境の名前と衝突しません。

・局所関数はパラメータの名前を自由変数とすることで、外部環境の変数(外側の関数のパラメータなど)にアクセスすることができます。局所関数は外側の関数の評価環境の下で評価されるからです。