Exercise 3.11

Problem statement

In 3.2.3 we saw how the environment model described the behavior of procedures with local state. Now we have seen how internal definitions work. A typical message-passing procedure contains both of these aspects. Consider the bank account procedure of 3.1.1:

```
(define (make-account balance)
  (define (withdraw amount)
    (if (>= balance amount)
        (begin (set! balance
                     (- balance
                        amount))
               balance)
        "Insufficient funds"))
  (define (deposit amount)
    (set! balance (+ balance amount))
    balance)
  (define (dispatch m)
    (cond ((eq? m 'withdraw) withdraw)
          ((eq? m 'deposit) deposit)
          (else (error "Unknown request:
                       MAKE-ACCOUNT"
                      m))))
  dispatch)
```

Show the environment structure generated by the sequence of interactions

```
(define acc (make-account 50))

((acc 'deposit) 40)
90

((acc 'withdraw) 60)
30
```

Where is the local state for acc kept? Suppose we define another account

```
(define acc2 (make-account 100))
```

How are the local states for the two accounts kept distinct? Which parts of the environment structure are shared between acc and acc2?

Solution

With the first instruction, `(define acc (make-account 50))` the variable `acc` is bound in the global environment to a a function with a body that essentially is the lambda of dispatch. This function however does not point to the global environment, but to a new environment E1, that contains the `balance` variable and its binding of 50 as well as the definitions of `withdraw`, `deposit` and

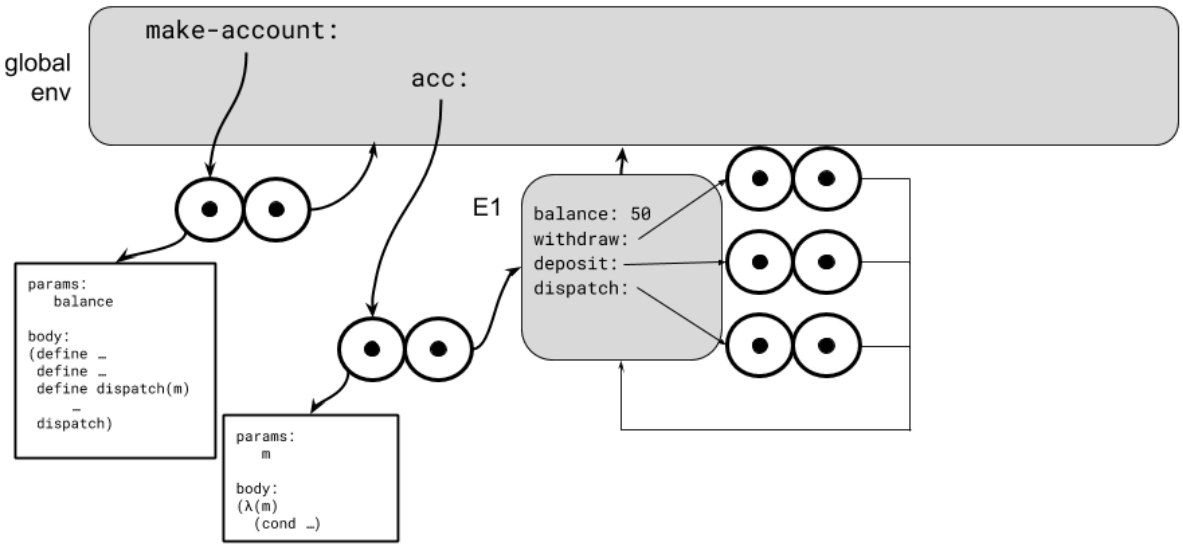

Figure 1: Frame structure created after `(define acc (make-account 50))`

`dispatch`. These definitions are functions that point back to the environment E1. The `withdraw` and `deposit` functions have a parameter `amount` and the dispatch function has a parameter `m`. The resulting environment structure is drawn in Figure 1.

The `(acc 'deposit)` part of the `((acc 'deposit) 40)` creates a frame E2 where the `m` variable is bound to the value `'deposit`. The result will return the function `deposit` that points back to environment 1. Once this function is applied on the value 40, a new frame with `amount` bound to value 40 is created. The `balance` variable in environment E1 is incremented by 40. The environment structure that results is shown if Figure 2.

Withdrawing from the account with `((acc 'withdraw) 50)` mimics the call for depositing. Two new environments are created, E4 and E5 that point to E1. E4 binds variable `m` to `'withdraw` and E5 binds variable `amount` to 50. The result of applying the `withdraw` procedure sets the value of `balance` in E1 to 40. The frame structure for this call is shown if Figure 3.

But what happens when we call `(define acc2 (make-account 100))`? Are we in actual danger of overwriting the balance of account one? Actually no, as we've seen in the previous exercise, acc2 will be pointing at a new environment E1', which will contain a separate binding for `balance`, and separate withdraw and deposit functions that will point back to E1'. The frame structure resulting from creating a second account is shown in Figure 4

Depending on the implementation of Scheme, the anonymous functions could share the same body, i.e. `acc` sharing the same body with `acc2` and `withdraw`, `balance` & `dispatch` for `acc` and `acc2`.

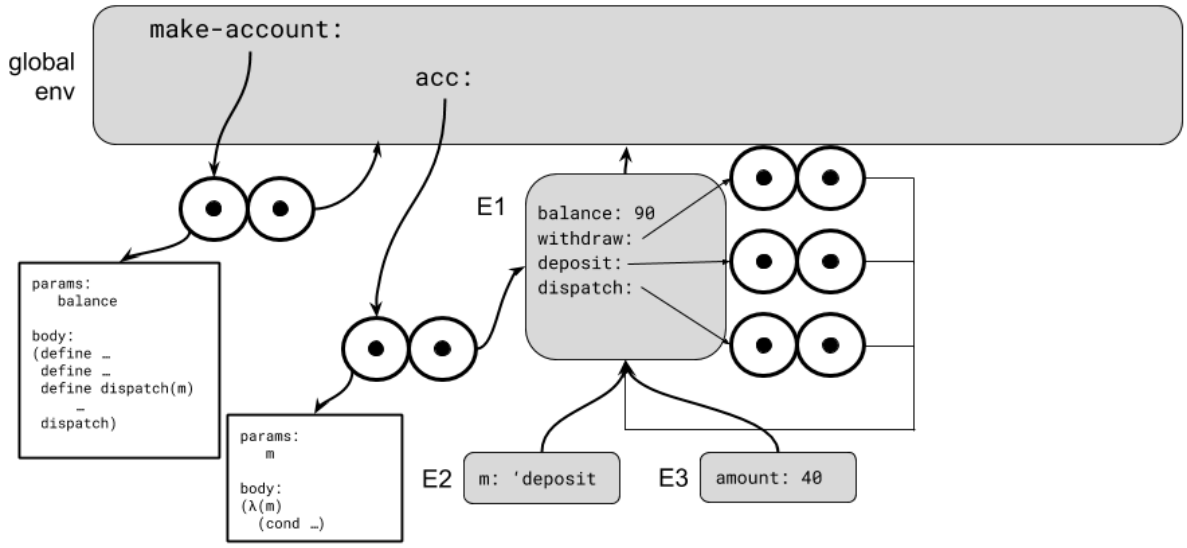

Figure 2: Frame structure created after `((acc 'deposit) 40)`

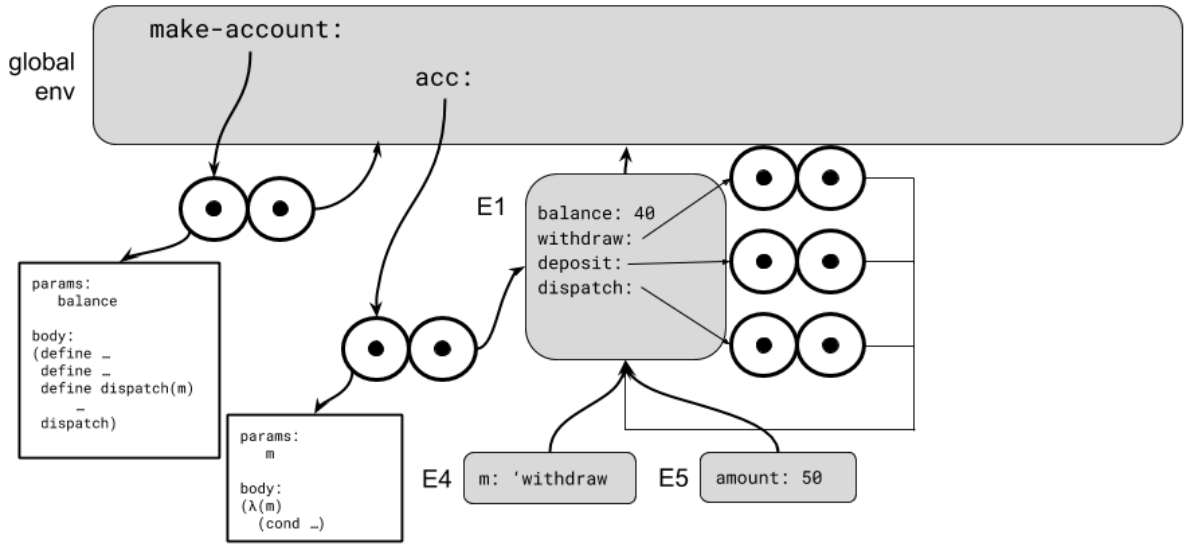

Figure 3: Frame structure created after `((acc 'withdraw) 50)`

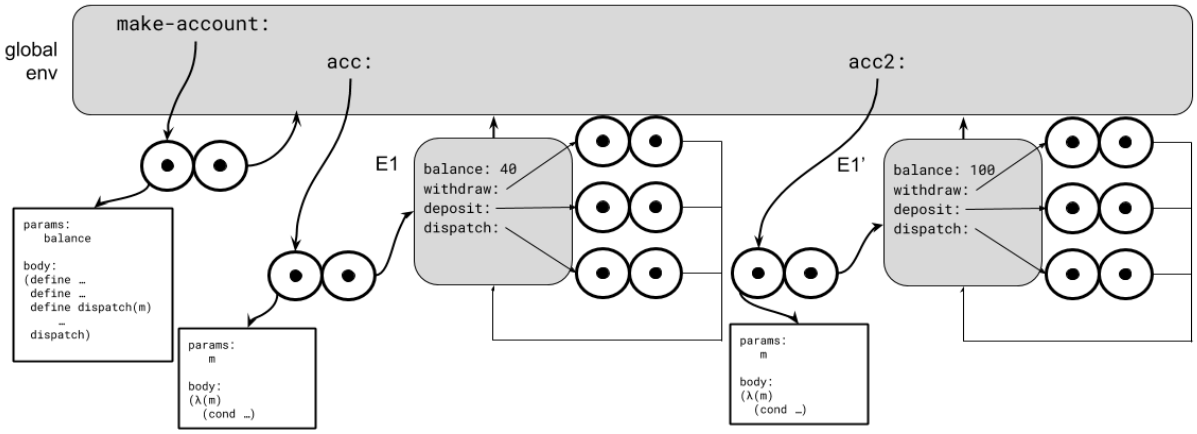

Figure 4: Frame structure created after `(define acc2 (make-account 100))`