Jens Nicolay
Academic Year 2017—2018

Higher-Order Programming

Project Assignment

Context

In Chapter 3 of the course we explained how we can structure our programs using objects with local state. A prototypical example of such an object is the "account" object.

```
(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)
(define acc (make-account 100))
((acc 'withdraw) 50) ; 50
((acc 'withdraw) 60) ; "Insufficient funds"
((acc 'deposit) 40) ; 90
((acc 'withdraw) 60) ; 60
```

Constructing objects as shown in the example above requires the definition of a dispatcher that takes a message as input and returns the value of some local variable (one of the two local procedures in the case of the account object). The dispatcher is returned as the value that represents the object.

Assignment

In the course we have seen a number of variations on the ordinary MC-eval. One such variation is the continuation-passing evaluator (CPS-eval) without separate syntactic analysis (`cps-eval.rkt` on Pointcarre in Documents > Project). The project consists of extending CPS-eval with three constructs for supporting the message-passing style of programming with objects as illustrated above: **`dispatch`**, **`send`**, and **`invoke`**.

- `(`**`dispatch`** `<name>*)` creates a dispatcher that understands zero or more names as a message.
- `(`**`send`** `<dispatcher> <name>)` sends a message `<name>` to a dispatcher. `<name>` must be in the list of names specified by the dispatcher. `<name>` is looked up in the environment of the dispatcher, and the resulting value is returned.
- `(`**`invoke`** `<dispatcher> <name> <arguments>*)` sends a message `<name>` to a dispatcher and expects a procedure as return value. The value of applying that procedure to the provided arguments is returned.

To illustrate, here is an example of using **`dispatch`** and **`send`**.

```
(define (box value)
    (dispatch value))

(define b (box 123))
(send b value) ; 123
```

To illustrate the use of **`invoke`**, consider the following example of a counter object.

```
(define (counter n)
    (define (inc) (set! n (+ n 1)))
    (define (get) n)
    (dispatch inc get))

(define c0 (counter 0))
(define c1 (counter 0))

(invoke c0 get) ; 0
(invoke c0 inc)
(invoke c0 get) ; 1
(invoke c1 get) ; 0
```

Finally, using our dispatcher constructs, the initial example involving the account object can be rewritten as follows.

```
(define (make-account balance)
   (define (withdraw amount) …)
   (define (deposit amount) …)
   (dispatch withdraw deposit))

(define acc (make-account 100))
(invoke acc withdraw 50) ; 50
(invoke acc withdraw 60) ; "Insufficient funds"
(invoke acc deposit 40) ; 90
(invoke acc withdraw 60) ; 30
```

Organization

You have to submit the project no later than **Sunday 14th January 2018, 23h59**.

You have to execute the project **individually**. Sharing code will be considered plagiarism.

You submit by uploading **one file** containing your evaluator to the dedicated space on Pointcarre. Use **your name** as the filename.

Hints

- Can the proposed constructs be implemented as special forms, or as procedures?
- Remember the possibility of defining constructs as syntactic sugar.