CSCI-GA-2110: Programming Languages, Fall 2024

PSet 4 - Written

Aditeya Baral
N19186654

1 Passing Self with Macros

Task 1.1 Explain what is wrong/unexpected with the macro implementation of `msg/self`. Concretely, give an example of an object and method invocation where replacing the original definition of `msg/self` with the macro version leads to different behaviors.

1. In the original function implementation, all arguments are evaluated before the function body is executed. This implies that **arguments such as `o`, `m` and all `a` are evaluated in the caller function's context first and only once**, before being passed to the function. The original function `mg/self` thus first applies the method to `self` and then applies the resulting function to the remaining arguments.
2. The macro implementation expands to `((o m) o a ...)`, which is evaluated in the caller's context. This implies that the **macro expands to code that evaluates `o` twice**, once in `(o m)` to get the method and another when passing `o` as the self parameter. This can lead to unexpected results if either `o` or `m` involves side effects or expensive computations. This difference leads to unexpected behavior when the arguments to `msg/self` involve expressions that have side effects or depend on the order of evaluation.
3. In the following example, we see two distinct behaviors exhibited across both implementations:

```
(define counter 0)

(define (make-incrementing-object)
  (set! counter (+ counter 1))
  (lambda (m)
    (case m
      [(get-counter) (lambda (self) counter)]
      [(identity) (lambda (self x) x)])))

;; Function implementation
(define (msg/self-func o m . a)
  (apply (o m) o a))

;; Macro implementation
(define-syntax (msg/self-macro stx)
  (syntax-case stx ()
    [(msg/self-macro o m a ...)
     #'((o m) o a ...)]))

;; Using function implementation
(display "Function implementation:\n")
```

```
(display (msg/self-func (make-incrementing-object) 'get-counter)) ; 1
(newline)
(display counter) ; 1
(newline)

;; Reset counter
(set! counter 0)

;; Using macro implementation
(display "Macro implementation:\n")
(display (msg/self-macro (make-incrementing-object) 'get-counter)) ; 2
(newline)
(display counter) ; 2
(newline)
```

(a) **Function Implementation**:

- `make-incrementing-object` is called once, incrementing `counter` to `1`.
- The `get-counter` method is called, returning `1`.
- The final value of `counter` is `1`.

(b) **Macro Implementation**:

- The macro expands to `((make-incrementing-object) 'get-counter) (make-incrementing-object)`.
- `make-incrementing-object` gets called twice:
  i. To get the method `((make-incrementing-object) 'get-counter)`
  ii. As the first argument to the method `(make-incrementing-object)`
- `counter` is incremented twice, reaching `2`.
- The `get-counter` method is called, returning `2`.
- The final value of `counter` is `2`.

(c) Thus, we see that the object creation function `make-incrementing-object` is evaluated multiple times, leading to unexpected `counter` increments. The key difference is that the **function implementation evaluates its arguments only once before applying the method, while the macro implementation can lead to multiple evaluations of expressions** that have side effects.

2 Continuations

Task 2.1 Using the notation in the above example, write down the continuations bound to `k` in `let/cc` for each of the following:

1. `(+ (+ 3 5) (+ (let/cc k 4) (+ 6 11)))`

```
(+ 8 (+ [...] (+ 6 11)))
```

2.
```
(begin (begin (set-box!  b 1) 1)
(let/cc k (unbox b))
(set-box!  b 2))
```

```
(begin 1 [...] (set-box! b 2))
```

3.
```
(map (begin
(begin (set-box!  b 1) #f)
(lambda (x) (+ (unbox b) x)))
(list (let/cc k 1) 2 3 (+ 2 2)))
```

```
(map (lambda (x) (+ (unbox b) x))
    (list [...] 2 3 (+ 2 2)))
```