Parameter Passing Mechanisms

Shahaf Shperberg, Lior Zur-Lotan and Avi Hayoun

December 27, 2017

Contents

1 Parameter Passing Mechanisms 1
  1.1 Refresher .................................................. 1
  1.2 Call-by-value vs Call-by-sharing .................................. 1
  1.3 Call-by-sharing vs Call-by-reference ................................ 2
  1.4 Call-by-sharing vs Call-by-name .................................... 2
  1.5 Call-by-name vs Call-by-need ....................................... 2

2 Call-by-name 2

1 Parameter Passing Mechanisms

1.1 Refresher

Call by Value The function receives a copy of the object.

Call by Sharing Any change to the contents of the object are persistent outside the function. Changes to the variable representing the object are local to the function.

Call by Reference The function is passed an alias of the parameter. Any changes to both the contents and the variable representing the object are persistent outside the function.

Call by Name Lazy evaluation of the parameter - it is re-evaluated each time it is used in the function, but it is not evaluated at all if it is not necessary.

Call by Need A cached version of Call by Name - the parameter is evaluated only once and only if necessary. Any other use of the parameter uses the pre-evaluated value.

1.2 Call-by-value vs Call-by-sharing

(let ((w "abc"))
  ((lambda (x) (string-set! x 0 #\z)) w)
  w)

Using the Call-by-value mechanism, the value of the expression is "abc", while using the call-by-sharing mechanism, the value is "zbc".
1.3 Call-by-sharing vs Call-by-reference

(let ((a 1)
     (b 2))
  ((lambda (x y) (set! x y)) a b)
  a)

  Using the Call-by-sharing mechanism, the value of the expression is 1, while using the call-by-reference mechanism, the value is 2.

1.4 Call-by-sharing vs Call-by-name

(let ((x 5))
  ((lambda (y) 3)
   (begin (set! x (+ x 1)) x))
  x)

  Using the Call-by-sharing mechanism, the value of the expression is 6, while using the call-by-name mechanism, the value is 5.

  ((lambda (x) (set! x 3)) y)

  Using the Call-by-sharing mechanism, the evaluating the expression will raise a runtime error, since y is undefined. Using the call-by-name mechanism, evaluating the expression will not raise an error, but will return the #<void> object.

1.5 Call-by-name vs Call-by-need

(let ((x 5))
  ((lambda (y) y y)
   @(begin (set! x (+ x 1)) x))
  x)

  Using the Call-by-name mechanism, the value of the expression is 7, while using the call-by-sharing mechanism, the value is 6.

2 Call-by-name

Passing a parameter \(x\) by name to a function \(\text{foo}\) requires delaying the evaluation of \(x\) until it is used in the body of \(\text{foo}\).

(define foo
  (lambda (x y) (if (< 0 y) y x)))

(foo @expr (+ 2 3))
The standard method for delaying the evaluation of an expression in Scheme is by turning it into a function.
Doing this would require modifying foo to apply x whenever its value is needed.

```
(define foo
  (lambda (x y) (if (< 0 y) y (x))))

(foo @expr (+ 2 3))
```

Unfortunately, this will not work in the general case; what will happen if y is passed by name as well? What will happen if we pass x by sharing?

Since we declare the parameter passing mechanism at the call site and not in the function definition, we must handle all parameters identically.

```
(define foo
  (lambda (x y) (if (< 0 (y)) (y) (x))))

(foo @expr (+ 2 3))
```

In this case assume that the expression passed by sharing y is evaluated and the result is wrapped by a lambda at the call site.
This is only a partial solution, since it ignores side-effects.

```
(define bla
  (lambda (x y) (if (not (null? y)) (set! (x) (y)))))

(foo @expr (+ 2 3))
```

In order to support both the delayed evaluation required for call-by-name, as well as side-effects, we need to construct a more elaborate object, with a setter to support side effects.