CS4613 Lecture 7: More About Objects

David Bremner

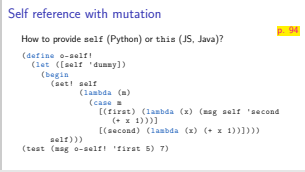
January 29, 2024

Self reference with mutation

p. 94

How to provide `self` (Python) or `this` (JS, Java)?

```
(define o-self!
  (let ([self 'dummy])
    (begin
      (set! self
            (lambda (m)
              (case m
                [(first) (lambda (x) (msg self 'second
                   (+ x 1)))]
                [(second) (lambda (x) (+ x 1))])))
      self)))
(test (msg o-self! 'first 5) 7)
```



1. The methods discussed in the book for providing `this` are very similar to methods used to provide recursive functions in interpreters. In that case the mutated self-reference is in the interpreter data structures.

… on stacker

p. 94

```
(defvar o-self!
  (let ([self 0])
    (begin
      (set! self
            (lambda (m)
              (if (equal? m "first")
                  (lambda (x) ((self "second") (+ x
                     1)))
                  (if (equal? m "second")
                      (lambda (x) (+ x 1))
                      (error "no such member")))))
      self)))
((o-self! "first") 5)
```

stacker

... without mutation 1/3

In order to convince ourselves that mutation is not a mandatory feature, we can implement the same thing without mutation. Let's see how python does it.

```
class Thing:
    def __init__(self, x):
        self.x=x

    def add(self, y):
        return self.x + y
```

… without mutation 1/3

In order to convince ourselves that mutation is not a mandatory feature, we can implement the same thing without mutation. Let's see how python does it.

```
class Thing:
    def __init__(self, x):
        self.x=x

    def add(self, y):
        return self.x + y
```

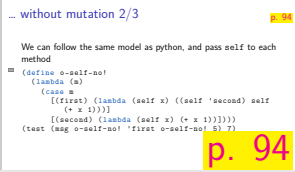
1. The book calls this unfortunate as a surface syntax. It is somewhat error prone, but it allows writing in an object oriented style in languages like C

... without mutation 2/3

We can follow the same model as python, and pass `self` to each method

no

```
(define o-self-no!
  (lambda (m)
    (case m
      [(first) (lambda (self x) ((self 'second) self
         (+ x 1)))]
      [(second) (lambda (self x) (+ x 1))])))
(test (msg o-self-no! 'first o-self-no! 5) 7)
```



1. Unlike the book, here we explicitly use the object twice here, once to find the method, and once as a method parameter

... without mutation 3/3

The `self` parameter is implicit in Python, and we can do the same

```
(define (msg/self obj selector . args)
  (apply (obj selector) obj args))
```

This also simplifies our sample class.

no2

```
(define o-self-no!
  (lambda (m)
    (case m
      [(first) (lambda (self x) (msg/self self 'second
         (+ x 1)))]
      [(second) (lambda (self x) (+ x 1))])))
(test (msg/self o-self-no! 'first 5) 7)
```

Motivation

Structural recursion is great, but

p. 95

```
(define-type BinTree
  [Empty]
  [Leaf (val : Number)]
  [Node (left : BinTree) (right : BinTree)])
```

All uses of the data type must change if it changes.

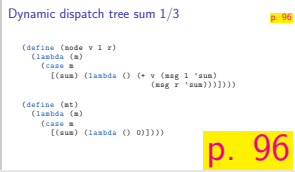
tsum

```
(define (tsum tree)
  (type-case BinTree tree
    [(Empty) 0]
    [(Leaf num) num]
    [(Node left right) (+ (tsum left) (tsum right))]))
```

Dynamic dispatch tree sum 1/3


```
(define (node v l r)
  (lambda (m)
    (case m
      [(sum) (lambda () (+ v (msg l 'sum)
                            (msg r 'sum)))])))

(define (mt)
  (lambda (m)
    (case m
      [(sum) (lambda () 0)])))
```



1. These constructor definitions are simplified compared to the book. Since `self` is unused, these examples skip defining it
2. The key point is that we are able to write the `sum` method for nodes without knowing what kind of object we are summing. The usual kind of trade-off applies: with this completely dynamic implementation we can't know until runtime if that object even supports a `sum` method

Dynamic dispatch tree sum 2/3

tsum2

```
(define a-tree
  (node 10
        (node 5 (mt) (mt))
        (node 15 (node 6 (mt) (mt)) (mt))))

(test (msg a-tree 'sum) (+ 10 5 15 6))
```

Dynamic dispatch tree sum 3/3

p. 96

Suppose we want to introduce a new node type

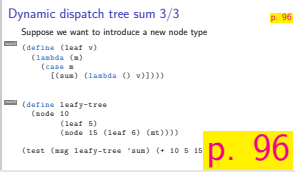
tsum3

```
(define (leaf v)
  (lambda (m)
    (case m
      [(sum) (lambda () v)])))
```

tsum3

```
(define leafy-tree
  (node 10
        (leaf 5)
        (node 15 (leaf 6) (mt))))

(test (msg leafy-tree 'sum) (+ 10 5 15 6))
```



1. The key point here is that the method definition in `node` does not change
2. None of this should be taken to suggest that the dynamic-dispatch version is the best for all situations, or even for the situation illustrated here.
3. In particular spreading the algorithm definition across different objects might or might not be desirable

What is inheritance?

p. 98

- In simplest terms, inheritance is when a method not found in the current object is searched for in or more parent objects.
- In our object model (assuming `parent-object` is initialized), this could look like

```
(case m
  ...
  [else (parent-object m)])
```

Inheritance in Java 1/2

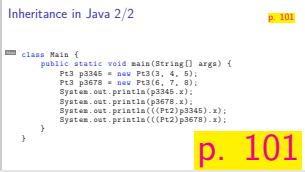
Pt2
```
class Pt2 {
    public int x;
    Pt2(int x, int y) {
        this.x = x - 3;
        System.out.println("Pt2(" + x + "," + y+")");
    }
}
```

Pt3
```
class Pt3 extends Pt2 {
    public int x;
    Pt3(int x, int y, int z) {
        super(x,y);  this.x=x+7;
        System.out.println("Pt3 with " + z);
    }}
```

Inheritance in Java 2/2

Main

```
class Main {
    public static void main(String[] args) {
        Pt3 p3345 = new Pt3(3, 4, 5);
        Pt3 p3678 = new Pt3(6, 7, 8);
        System.out.println(p3345.x);
        System.out.println(p3678.x);
        System.out.println(((Pt2)p3345).x);
        System.out.println(((Pt2)p3678).x);
    }
}
```



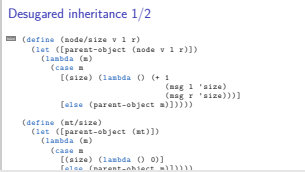
1. We can see both constructors running from the first 4 lines of output
2. We can see that separate objects are allocated for the superclass from the second 4 lines of output

Desugared inheritance 1/2

tsize

```
(define (node/size v l r)
  (let ([parent-object (node v l r)])
    (lambda (m)
      (case m
        [(size) (lambda () (+ 1
                              (msg l 'size)
                              (msg r 'size)))]
        [else (parent-object m)]))))

(define (mt/size)
  (let ([parent-object (mt)])
    (lambda (m)
      (case m
        [(size) (lambda () 0)]
        [else (parent-object m)]))))
```



1. This is simplified to use a fixed base class. The version in the book is more suitable for something like mixins, discussed at the end of the chapter.

Desugared inheritance 2/2

```
(define a-tree/size
  (node/size 10
             (node/size 5 (mt/size) (mt/size))
             (node/size 15
                        (node/size 6 (mt/size)
                           (mt/size))
                        (mt/size))))

(test (msg a-tree/size 'sum) (+ 10 5 15 6))
(test (msg a-tree/size 'size) 4)
```