Assignment 4: Dynamic Scope

On two occasions I have been asked [by members of Parliament],–“Pray, Mr. Babbage, if you put into the machine wrong figures, will the right answers come out?” […]

—Charles Babbage, 1864

Assignment

This assignment has three parts. We expect you to turn in **three interpreters**: `value-of-fn`, `value-of-ds`, and a third, `value-of-scopes` with your implementation of dynamic scope. During this assignment we will also extend the interpreter to work with cons, cdr, and car in part two. The main task of this assignment is to implement dynamic scope. **You must complete `value-of-scopes` to get an S- or above on this assignment**.

You should be able to use the a4-student-tests.rkt file to test your solutions.

```
> (require "a4-student-tests.rkt")
> (test-file #:file-name "a4.rkt")
...
```

and that should get you going. Of course, **these tests are not exhaustive; you should add your own tests as well**.

Part I

Using one of your interpreters from last week's assignment as a starting point, create two interpreters that are representation independent with respect to closures. You should add a set of two new closure helpers for each: `apply-closure-fn`, `closure-fn`, `apply-closure-ds`, and `closure-ds`. Since you are free to pick a single representation of environments, you should change either `empty-env-fn` or `empty-env-ds` to `empty-env`.

```
1. ''value-of-fn'' should use a functional representation of closures.
2. ''value-of-ds'' should use a data-structural representation of closures.
```

Your interpreters must work for at least these test cases. Of course, **these tests are not exhaustive; you should use your own tests as well**. Remember to place the definition of the test macro above where tests appear in your file.

```
> (value-of-fn
    '((lambda (x) (if (zero? x)
                      12
                      47))
       0)
     (empty-env))
12
> (value-of-fn
   '(let ([y (* 3 4)])
      ((lambda (x) (* x y)) (sub1 6)))
   (empty-env))
60
> (value-of-fn
   '(let ([x (* 2 3)])
      (let ([y (sub1 x)])
        (* x y)))
   (empty-env))
30
> (value-of-fn
   '(let ([x (* 2 3)])
      (let ([x (sub1 x)])
        (* x x)))
   (empty-env))
25
> (value-of-ds
    '((lambda (x) (if (zero? x)
                      12
                      47))
       0)
     (empty-env))
12
> (value-of-ds
   '(let ([y (* 3 4)])
      ((lambda (x) (* x y)) (sub1 6)))
   (empty-env))
60
> (value-of-ds
   '(let ([x (* 2 3)])
      (let ([y (sub1 x)])
        (* x y)))
   (empty-env))
30
> (value-of-ds
   '(let ([x (* 2 3)])
      (let ([x (sub1 x)])
        (* x x)))
   (empty-env))
25
```

Part II

The second part of this week's assignment is to add *dynamic scope* to your interpreter. Choose either `value-of-fn` or `value-of-ds` from Part I as a starting point.

Explanation of dynamic scope

The interpreters we have been writing so far have been implemented in such a way that, if there are variables that occur free in a procedure, they take their values from the environment in which the `lambda` expression is defined. We accomplish this by creating a closure for each procedure we see, and we save the environment in the closure. This technique is called *static binding of variables*, or *static scope*. Lexical scope is a kind of static scope.

Alternatively, we could implement our interpreters such that any variables that occur free in the body of a procedure get their values from the environment from which the procedure is *called*, rather than from the environment in which the procedure is *defined*.

For example, consider what would happen if we were to evaluate the following expression in an interpreter that used lexical scope:

```
(let ([x 2])
  (let ([f (lambda (e) x)])
    (let ([x 5])
      (f 0))))
```

Our lexical interpreter would add `x` to the environment with a value of `2`. For `f`, it would create a closure that contained the binding of `x` to `2`, and it would add `f` to the environment with that closure as its value. Finally, the inner `let` would add `x` to the environment with a value of `5`. Then the call `(f 0)` would be evaluated, but since it would use the value of `x` that was saved in the closure (which was `2`) rather than the value of `x` that was current at the time `f` was called (which was `5`), the entire expression would evaluate to `2`.

Under dynamic scope, we wouldn't save the value of `x` in the closure for `f`. Instead, the application `(f 0)` would use the value of `x` that was current in the environment at the time it was called, so the entire expression would evaluate to `5`.

As you can see, dynamic scope is a little strange, but it does have its uses.

Consider this slightly modified version of the interpreter which uses a higher-order representation for closures. Notice that the `lambda` line now returns a procedure that takes two arguments, where the second argument is an environment. This change in the `lambda` line also requires a change in the application line: we now have to pass in the environment, since the `rator` is a two-argument function.

This interpreter also should be completed to properly work with null?, cons, cdr, and car. Note that we match on `(quote ())` for the empty list: `(cons 1 (cons 2 '()))` should return `(1 2)`.

```
(define value-of-scopes
  (lambda (exp env)
    (pmatch exp
      [`,n (guard (number? n)) n]
      [`,x (guard (symbol? x)) (apply-env env x)]
      [`(quote ()) '()]
      [`(null? ,ls) _____]
      [`(cons ,a ,d) _____]
      [`(car ,ls) _____]
      [`(cdr ,ls) _____]
      [`(* ,nexp1 ,nexp2) (* (value-of-scopes nexp1 env) (value-of-scopes nexp2 env))]
      [`(sub1 ,nexp) (sub1 (value-of-scopes nexp env))]
      [`(if ,t ,c ,a) (if (value-of-scopes t env)
                          (value-of-scopes c env)
                          (value-of-scopes a env))]
      [`(let ((,x ,e)) ,body) (let ((a (value-of-scopes e env)))
                                (value-of-scopes body (extend-env x a env)))]
      [`(lambda (,x) ,body) (lambda (a env^) (value-of-scopes body (extend-env x a env)))]
      [`(d-lambda (,x) ,body) ______]
      [`(,rator ,rand) ((value-of-scopes rator env) (value-of-scopes rand env) env)])))
```

Things to do for Part II

3. First, copy either your `value-of-fn` or `value-of-ds` you defined for Part I, rename it to `value-of-scopes`, and using the above interpreter as a guide, add `d-lambda` to `value-of-scopes`. **Hint: You will need to add one line to your interpreter to handle `d-lambda` expressions, which have almost the same syntax as `lambda` expressions, but**

different semantics. We have added the left side of the `d-lambda` line in the example interpreter above.

We name this new interpreter `value-of-scopes` because this new interpreter can handle both lexical (with lambda forms) and dynamic scope (with d-lambda forms). Copy your `closure` and `apply-closure` helpers for whichever interpreter you copied from above (`value-of-fn` or `value-of-ds`), rename them `closure-scopes` and `apply-closure-scopes`, and modify them as necessary to handle the new `d-lambda` form. Modify the interpreter to use these helpers.

A friendly reminder: if you implement `value-of-scope` using `value-of-ds` as a base, you can easily trace test programs like we did in class using the C311/trace library and see how your environment updates as your interpreter evaluates expressions.

Here are two tests that will pass once you have added `d-lambda` to the interpreter.

```
> (value-of-scopes '(let ([x 2])
                      (let ([f (lambda (e) x)])
                        (let ([x 5])
                          (f 0))))
                   (empty-env))
2
> (value-of-scopes '(let ([x 2])
                      (let ([f (d-lambda (e) x)])
                        (let ([x 5])
                          (f 0))))
                   (empty-env))
5
> (value-of-scopes
    '(let
       ([l (cons 1 (cons 2 (cons 3 '())))])
         ((map (lambda (e) (cons e l))) l))
    (extend-env
      'map
      (value-of-scopes
        '(let ([map (lambda (map)
                      (lambda (f)
                        (lambda (l)
                          (if (null? l) '()
                              (cons (f (car l)) (((map map) f) (cdr l)))))))])
               (map map)) (empty-env))
      (empty-env)))
'((1 1 2 3) (2 1 2 3) (3 1 2 3))
> (value-of-scopes
    '(let
       ([l (cons 1 (cons 2 (cons 3 '())))])
         ((map (d-lambda (e) (cons e l))) l))
    (extend-env
      'map
      (value-of-scopes
        '(let ([map (lambda (map)
                      (lambda (f)
                        (lambda (l)
                          (if (null? l) '()
                              (cons (f (car l)) (((map map) f) (cdr l)))))))])
               (map map)) (empty-env))
      (empty-env)))
'((1 1 2 3) (2 2 3) (3 3))
;; Notice the behavior of let in this next example.
;; we get letrec for free. (This is not a good thing.)
> (value-of-scopes
    '(let
       ([map (d-lambda (f)
               (d-lambda (l)
                 (if (null? l) '()
                     (cons (f (car l)) ((map f) (cdr l))))))])
        (let ([f (d-lambda (e) (cons e l))])
          ((map f) (cons 1 (cons 2 (cons 3 '()))))))
    (empty-env))
'((1 1 2 3) (2 2 3) (3 3))
```

Brainteasers

4. We've been talking a whole lot about representation independence. It would sure be nice to have a single interpreter where we could just pass in the various helper functions. That way we could write the interpreter once, and then swap in whatever representations we wanted. It would be obvious on first inspection of this interpreter that this single interpreter is indeed representation independent with respect to both environments and closures.

So let's do it.

Write a single, curried function named `value-of-ri` that'll take `empty-env`, `extend-env`, `apply-env`, `closure`, and `apply-closure`, and return an interpreter expecting a single expression. You'll need to pass an additional parameter to your closure helper functions; define new functions `closure-fn-ri` and `apply-closure-fn-ri` `closure-ds-ri` and `apply-closure-ds-ri` that take this additional parameter. Go ahead and include your regular `if`, `*`, `sub1`, `zero?`, `let`, forms, along with numbers and booleans, and `lambda`-calculus expressions. You should not pass your helper functions to recursive calls. Here's what the calls to initially kick off the interpreter should look like.

```
>((value-of-ri empty-env-fn extend-env-fn apply-env-fn closure-fn-ri apply-closure-fn-ri) '((lam
5
>((value-of-ri empty-env-ds extend-env-ds apply-env-ds closure-ds-ri apply-closure-ds-ri) '((lam
5
>((value-of-ri empty-env-fn extend-env-fn apply-env-fn closure-ds-ri apply-closure-ds-ri) '((lam
5
>((value-of-ri empty-env-ds extend-env-ds apply-env-ds closure-fn-ri apply-closure-fn-ri) '((lam
5
```

Your solution should involve a letrec.

assignment-4.txt · Last modified: 2014/09/18 19:02 by joshcox