Closure-Creating Interpreters

CIS352 — Fall 2022

Kris Micinski

In the last several lectures we have discussed the λ-calculus

Today, we will build a metacircular interpreter for the call-by-value λ-calculus

Our interpreter will be like that of IfArith's—we will use Racket's features to implement λ

We will consider λ-calculus extended with numbers and +

```
e ::= n
    | x
    | (+ e e)
    | (lambda (x) e)
    | (e e)
```

Following our convention, we can represent this in Racket via a type predicate…

```
(define (expr? e)
  (match e
    [(? number? n) #t]
    [`(+ ,(? expr? e0) ,(? expr? e1)) #t]
    [(? symbol? x) #t]
    [`(lambda (,(? symbol? x)) ,(? expr? e)) #t]
    [`(,(? expr? e0) ,(? expr? e1)) #t]
    [_ #f]))
```

Our job is to define a function called **interp** which interprets each expression to its value

So first, we must **define** the values of our language

Like IfArith, our language includes numbers. But unlike IfArith, we also include λ

How do we represent λ as a value?

Remember: a programming language's values are the **results** of computation

So this is equivalent to asking: what will our interpreter return?

```
((lambda (x) 3) 4) ⇓ 3

(lambda (x) 3) ⇓ …?
```

One option: lambdas evaluate to **text itself**

```
(lambda (x) 3) ⇓ (lambda (x) 3)
```

This gives us a **textual reduction** semantics, i.e., exactly the reduction rules we've been studying in the last few lectures

Unfortunately—in lieu of advanced representations—textual reduction semantics can be often **very slow** because they perform *explicit substitution*

We would like each computation step to be O(1), so that a program which otherwise takes O(f(n)) time takes O(f(n)) time to compute (rather than O(f(n) * n) or worse!)

Instead, our machine will use **closures** to perform substitution **lazily**. We will do this by tracking an **environment** in which variables are looked up.

When returned as results, a λ must track its free variables. We bundle the λ and its environment together, and this is called a **closure**

```
closure ::= (closure (lambda (x) e) env)
```

Thus, we will have two kinds of values: numbers and closures

```
env = variable -> value

value ::= n
        | (closure (lambda (x) e) env)
```

Thus, we will have two kinds of values: numbers and closures

```
env = variable -> value

value ::= n
        | (closure (lambda (x) e) env)
```

Note: environments and values are **mutually recursive**

As a sidenote, Haskell uses the STG machine to enable lazy graph reduction

https://www.microsoft.com/en-us/research/wp-content/uploads/1992/04/spineless-tagless-gmachine.pdf

Let's decide how to handle each of these cases…

```
;; numbers
(interp n env) ⇓ n
;; variable lookup
(interp x env) ⇓ (hash-ref env x)
```

```
;; plus
If…
-(interp e0 env) ⇓ n0
-(interp e1 env) ⇓ n1
-n′ = n0 + n1
-————————————————
(interp `(+ ,e0 ,e1) env) ⇓ n′
```

```
;; λ
(interp `(lambda (,x) ,e) env) ⇓
   (closure (lambda (x) e) env)
```

```
;; apply (i.e., call-and-return)

If…
–(interp e0 env) ⇓
   (closure (lambda (x) e) env+)
–(interp e1 env) ⇓ v
–(interp e (hash-set env+ x v)) ⇓ v’

Then…
(interp `(,e0 ,e1) env) ⇓ v’
```

How can we take these rules and implement them as a Racket function?

- Recursive function **interp**, match on expression:
  - Base cases are λ, numbers, and variables
  - Recursive cases are + and apply
    - Apply first evaluates function expr to a closure
    - Then evaluates body of closure after updating the formal parameter in the *stored* environment