Comp 411
Principles of Programming Languages
Lecture 8
Meta-interpreters II

Corky Cartwright

January 25, 2023

Representation Tricks

- We described closures (the meaning of lambda-abstractions) as `<code, env>` pairs.
  - Are other representations possible/defensible? Yes, particularly in a functional language.
  - Closures can be represented as (Scheme/Racket) functions Idea: wrap `(lambda (v) ...)` around code `...` that applies the pair closure in our meta-interpreter to `v`.
- What about environment representations?
  - A functional representation mapping symbols to values is elegant but not good software engineering.

Alternate CBV Meta-interpreter

```
;; V = Const | V → V
;; Binding = (make-Binding Sym V) ; Note: Sym not Var
;; Env = (list-of Binding)
;; Closure = V → V                ; Note: an opaque rep
;; eval: R Env → V                ; Note: an opaque rep
(define eval … <unchanged> …)     ; Assumes API for closure

;; apply: Closure V → V  ; assumes that Closure rep is V → V
(define apply (lambda (cl v) (cl v)))

;; make-closure: Proc Env → Closure
(define (make-closure M env)
  (lambda (v)
    (eval (proc-body M)
      (cons (make-binding (proc-param M) v) env))))
```

This code does not encapsulate the representation of closures. We explicitly use a closure as a function and we use `make-closure` as a function name (which is legal but a bad idea in real code). How would the code this change if we encapsulated it? Think OO.

Closures as Functions

- Mathematically elegant
- Disadvantageous from software engineering perspective. Why? Functions are opaque Their internal form generically cannot be examined. (Why?) Closures as structures, in contrast, are open to inspection.
- Not literally possible in languages like Java 5+ that support inner classes rather than closures. But there is a Java 5+ equivalent: return a class implementing an interface **`Lambda<V,V>`** with an explicit **`apply`** method, leveraging the *strategy/command* design pattern. The addition of "lambda-expressions" to Java 8, provides functional notation for this idiom. This hack can be used even in assembly/machine language, but it is so messy that it is impractical.
- The Java formulation has essentially the same advantages and disadvantages as the Scheme formulation. Note: Comp 310 formerly relied on a course library with interfaces **`Ilambda<In,Out>`**. In Java 8+, closures can be used in source code but they are implemented as anonymous inner classes!

•

CBV Meta-interpreter with Environments as Functions

```
;; V = Const | V → V
;; Binding = (make-Binding Sym V)      ; Note: Sym not Var
;; Env = Sym → V
;; Closure = V → V

;; eval: R Env → V
(define eval … <unchanged> …)

;; apply: Closure V → V
(define apply (lambda (cl v) (cl v)))

;; make-closure: Proc Env → Closure
(define (make-closure M env)   ;; name make-closure is sneaky
  (lambda (v)
    (eval (proc-body M) (extend (proc-param M) v env))))

(define lookup (lambda (s env) (env s)))
(define extend (lambda (s1 v env)
  (lambda (s2) (if (equal? s1 s2) v (env s2)))))
```

Environments as Functions

- Mathematically elegant
- Questionable from software engineering perspective. Why?
- Functions are generally not finite and cannot be treated as tables.
- Environments, in contrast, are finite functions. One consequence of the fact that functions are infinite objects in the general case: functions are opaque in output while concrete closures (data structures representing finite tables) are not.
- Not literally possible in languages like Java 8-13 that support inner classes rather than closures. But there is a Java equivalent: a singleton class implementing an interface `Lambda<Sym,V>` the *strategy* (or *command*) design pattern. Java formulation has essentially the same advantages and disadvantages as the Scheme formulation.

Exercise: revise our previous correct meta-interpreters to use `extend` instead of `cons`. Explicitly define `lookup` and `extend`.

Important Variations on Our CBV Meta-interpreter

- *Call-by-name* (CBN) beta-reduction. Recall that in our syntactic intepreter for LC that we chose to *restrict* beta-reduction to *values*. In practice, this restriction is very important in languages with *mutable* data. But LC does not (yet) support *mutation*. In CBN, beta-reduction is unrestricted.
- *Call-by-need* evaluation of arguments. There is no syntactic equivalent since this evaluation policy is a meta-interpreter based optimization of *Call-by-name*. In the presence of mutation (or equality comparison on "functions" [comparing addresses of function representations!]), *call-by-need* is not equivalent to *call-by-name*.

Call-by-name Discussion

- In *Call-by-name* syntactic interpretation, no argument is evaluated until its value is demanded by a primitive operation (only **+** in LC). If a parameter is never evaluated in the body of function, the corresponding argument is never evaluated.
- Efficiency disadvantage: if a parameter is evaluated multiple times, so is the corresponding argument!
- Thought exercise: how can we defer the evaluation of an argument expression? Hint: think about the closure representation of 0-ary functions.
- What about call-by-need? How do we evaluate a closure at most once. Think OO; we need to add a field to our closure representation. That field is initialized when the argument closure is first evaluated.