

Comp 311
Principles of Programming Languages
Lecture 8
Meta-interpreters II

Corky Cartwright
September 11, 2009

Representation Tricks

- We described closures (the meaning of program lambda-expressions) as `<code, env>` pairs
 - Are other representations possible/defensible? Yes, particularly in a functional language.
 - Closures can be represented as (Scheme) functions. Idea: wrap **`lambda (v) ...`** around code applying the pair closure in our meta-interpreter to **`v`**.
- What about environment representations?

Revised Meta-interpreter

```
:: V = Const | V → V
```

```
:: Binding = (make-Binding Sym V) ; Note: Sym  
not Var
```

```
:: Env = (listOf Binding)
```

```
:: R Env → V
```

```
(define eval ... <unchanged> ...)
```

```
:: (V → V) V → V
```

```
(define apply (lambda (cl v) (cl v)))
```

```
:: Proc Env → (V → V)
```

```
(define (make-closure M env)
```

```
  (lambda (v)
```

```
    (eval (proc-body M)
```

```
      (cons (make-binding (proc-param M) v)  
            env))))
```

Closures as Functions

- Mathematically elegant
- Questionable from software engineering perspective. Why? Functions are opaque. Their internal form cannot be examined. (Why?) Closures as structures, in contrast, are open to inspection.
- Not literally possible in languages like Java that support inner classes rather than closures. But there is a Java equivalent: return a class implementing an interface **Lambda<V,V>**, the *strategy/command* design pattern. Java formulation has essentially the same advantages and disadvantages as the Scheme formulation.

Meta-interpreter with Environments as Functions

```
:: V = Const | V → V
```

```
:: Binding = (make-Binding Sym V) ; Note: Sym not Var
```

```
:: Env = Sym → V
```

```
:: R Env → V
```

```
(define eval ... <unchanged> ...)
```

```
:: (V → V) V → V
```

```
(define apply (lambda (cl v) (cl v)))
```

```
:: Proc Env → (V → V)
```

```
(define (make-closure M env)
```

```
  (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 is that functions are opaque in output while structure closures are not.
- Not literally possible in languages like Java that support inner classes rather than closures. But there is a Java equivalent: return a class implementing an interface **Lambda<Sym,V>**, the *strategy/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 Meta-interpreter

- *Call-by-name* (CBN) beta-reduction. Recall that in our syntactic interpreter 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*.
- *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, *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, the corresponding argument is never evaluated.
- 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 closures)?