Comp 411
Principles of Programming Languages
Lecture 9
Meta-interpreters III

Corky Cartwright
January 30, 2017
Major Challenge

LC does not include a recursive binding operation (like Scheme `letrec` or `local`). How would we define `eval` for such a construct?

- Key problem: the closure structure for a recursive `lambda` must include an environment that refers to itself!
- In imperative Java, how would we construct such an environment. Hint: how do we build “circular” data structures in general in Java? Imperativity is *brute force*. But it works. We will use it in Project 3 and thereafter.
Minor Challenge

How could we define an environment that refers to itself in *functional* Scheme (or Ocaml)?

- Key problem: observe that in `let` and `lambda` the expression defining the value of a variable cannot refer to itself.
- Solution: does functional Scheme (or Ocaml) contain a recursive binding construct?
- How can we use this construct to define a recursive environment?
- What environment representation must we use?
Advantage of Representing Environments as Functions

Languages that support functions as values (or an OO equivalent like anonymous inner classes [Java] or anonymous delegates [C#]) support the dynamic definition of recursive functions. So we can write a purely functional interpreter that handles recursive binding by constructing a new environment (a function) that recurs on itself (refers to itself). In Scheme, given a function \( e \) that represents the current environment, we can extend \( e \) with a new binding of symbol 'f' to an AST \( \text{rhs} \) (right-hand-side) that is evaluated in the extended environment by constructing the environment

\[
(\text{define new-e (lambda (sym) (cons (cons sym (eval rhs new-e)) e))})
\]

where \texttt{eval} is the meta-interpreter.
A Bigger Challenge

Assume that we want to write LC in a purely functional language without a recursive binding construct (say functional Scheme without `define` and `letrec`)?

• Key problem: must expand `letrec` into `lambda`

• No simple solution to this problem. We need to invoke syntactic magic or (equivalently) develop some sophisticated mathematical machinery.
Key Intuitions

• Computation is incremental—not monolithic.
• Slogan: general computation is successive approximation (typically in response to successive demands for more information).
• Familiar example: a program mapping a potentially infinite input stream of characters to a potentially infinite output stream of characters. Generalization: infinite trees mapped to infinite trees.
Mathematical Foundations

Domains of computations (like streams, trees, partial functions as graphs):

- partially ordered set (\( \text{po} \))
- finitary basis (set of finite approximations)
  - countable
  - closed under LUBs on finite bounded subsets
- chain
- chain-complete
- complete partial order (\( \text{cpo} \))
- “home-plate” \( \text{cpo} \) (not domain; finite elements not a finitary basis)
- bottom (\( \bot \))
- flat domain (monolithic set of values formulated as domain)
  - integers, booleans, strings, conventional finite lists, ASTs
Key Mathematical Concepts

Computable functions:
• monotonic (universal)
• continuous (universal)
• strict (typical)
Examples

Domains

• flat domains
• strict function spaces on flat domains
• lazy trees of boolean (of $D$ where $D$ is flat)
• factorial functional

See “Domain Theory: An Introduction” in References for Lectures 10-12