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

Corky Cartwright
January 31, 2018
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?

• What environment representation must we use?
Advantages 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 assigns a meaning to 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 \( rhs \) (right-hand-side) that is evaluated in the extended environment by constructing the environment

\[
\text{(define new-e (lambda (sym) (if (=? sym f) (eval rhs new-e) (e sym))))}
\]

where \textit{eval} is the meta-interpreter. We can introduce recursive binding without the side effect introduced by \textit{define} by using the the Scheme construct \textit{letrec}.

Scheme \textit{letrec} is akin to \textit{let} except that it performs recursive binding instead of conventional binding, \textit{i.e.}, that the new environment created by \textit{letrec} is used to evaluate all subexpressions on the rhs of the symbol definition added by \textit{letrec} (see the syntax for \textit{let} in the previous lecture). Note that the binding of the new symbol is unavailable (sometimes represented by the error value \textit{*void*}) until the evaluation of the rhs is complete. This trick works for \textit{letrec} constructs that introduce new function definitions by not for other kinds of data unless the constructors for that form of data are “lazy” (delaying the evaluation of their arguments until demanded by an accessor operation).
A Bigger Challenge

• Assume that we want to write LC in a purely functional language without a recursive binding construct (say functional Scheme without \texttt{letrec}.

• Key problem: must expand \texttt{letrec} into \texttt{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 (po)
- finitary basis (set of finite approximations) is a po that is
  - countable
  - closed under LUBs on finite bounded subsets (implies $\bot$ exists)
    - counterexample: “home-plate” cpo where finite elements not a basis
      - typically no limit points in finitary basis (exceptions are rare and pathological)
- chain-complete (in a po, a chain is a countable ascending sequence of elements $b_0 \leq b_1 \leq \ldots \leq b_k \leq \ldots$): every chain has a LUB
- Given a finitary basis with no limit points, chain-completion generates a complete partial order (cpo) including all limit points with finitary basis as basis.
- Examples:
  - flat domains (integers, booleans, finite strict trees)
  - lazy tree domains (potentially infinite trees generated by finite set of node types)
Key Mathematical Concepts

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

Domains

- flat domains
- strict function spaces on flat domains (CBV)
- lazy trees of boolean (of D where D is flat)
- continuous functions $A \rightarrow B$ where $A$ and $B$ are domains

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