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

Corky Cartwright

February 10, 2021

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 both `let` and `lambda`, the expression defining the value of a variable cannot refer to itself because the corresponding variable is out of scope. Moreover, it is not yet bound.

- Solution: does functional Scheme (or Ocaml) contain a recursive binding construct? (Yes for function definitions [`define` in Scheme].)

- What environment representation must we use?

Advantages of Representing Environments as Functions

anguages that support functions as values (or an OO equivalent like anonymous inner lasses [Java] or anonymous delegates [C#]) support the dynamic definition of recursive unctions. So we can write a purely functional interpreter that assigns a meaning to a ecursive binding by constructing a new environment (a function) that recurs on itself refers to itself). In Scheme/Racket, given a function **`e`** that represents the current nvironment, we can extend **`e`** with a new binding of symbol **`f`** to an AST **`rhs`** (right-hand-de) that is evaluated in the extended environment by constructing the environment

```
(define new-e (lambda (sym) (if (=? sym 'f) (eval rhs new-e) (e sym))))
```

vhere **`eval`** is the meta-interpreter. Scheme/Racket also includes a local recursive binding onstruct called **`letrec`**.

cheme/Racket **`letrec`** is akin to **`let`** except that it performs recursive binding instead of onventional binding, *i.e.*, that the new environment created by **`letrec`** is used to evaluate ll subexpressions on the right-hand-side (*rhs*) of the symbol definition added by **`letrec`** see the syntax for **`let`** in the previous lecture). Note that the binding of the new symbol unavailable (sometimes represented by the error value **`*void*`**) until the evaluation of ne *rhs* is complete. This trick works for **`letrec`** constructs that introduce new function efinitions but 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 peration).

A Bigger Challenge

- Assume that we want to write LC in a purely functional language without a recursive binding construct (say functional Scheme without `letrec`).
- Key problem: must expand `letrec` into `lambda`.
- There is no simple solution to this problem. We need to invoke syntactic magic or (equivalently) develop some sophisticated mathematical machinery (which motivates the syntactic magic). The syntactic magic (for call-by-name) is the `Y` operator from the pure lambda calculus.

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. This generalization is very powerful. In the framework of sequential computation with aborting error elements (like the result of division by zero), every function can be canonically represented by a potentially infinite t

Mathematical Foundations

A (Scott) domain of computation **D** (like streams, trees, partial functions as graphs) is a partially ordered set (**po**) with the following properties:

- **D** has a countable subset **B** (set of *finite* approximations), called the *finitary basis*, which is a **po** that is *finitely consistent*, *i.e.*, closed under LUBs on finite bounded subsets (implying a unique least elt $\perp$ exists as the LUB of the empty set). We will restrict our attention to finitary bases **B** where no element b in **B** is the LUB of an infinite subset of **B** (called *finitely-founded*). Since **B** is a basis, every element *d in* **D** is the LUB of the finite elements that approximate it.
- **D** is *chain-complete*: every chain $b_0 \leq b_1 \leq \ldots \leq b_k \leq \ldots$ (a countable ascending sequence) in **B** has a LUB in **D**.
- A **po** with that is chain-complete is called a **cpo** (*complete partial order*).

Every computational domain can be formalized as a Scott-doain

Note: in the reference monograph, directed sets are used instead of chains. When the finitary basis is countable, it is straightforward to show that *chain-complete* and *directed-complete* are equivalent.

Examples of (Scott) domains:

- flat domains: integers, booleans, finite trees with no undefined ($\perp$) leaves;
- lazy tree domains: potentially infinite trees with a finite set of node types and undefined ($\perp$) leaves.

Key Mathematical Concepts

Computable functions on domains:

- monotonic (universal)
- continuous (universal)
- strict (typical in practical programming languag

For a brief, intuitive overview, see the topic notes for
lecture 11

https://www.cs.rice.edu/~javaplt/411/19-spring/Notes/11/06.

For an in-depth treatment of (Scott) domains, see the
monograph linked under references for lecture 10.

Examples

Domains

- flat domains
- strict function spaces on flat domains (CBV)
- lazy trees of booleans
- continuous functions $A \longrightarrow B$ where A and B are domains

The notion of continuity here is very important; it enables interchanging function application and the LUB operation on chains.