

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

Corky Cartwright
January 28, 2022

Minor Challenge

- LC does not include a recursive binding operation (like Scheme **letrec** or **local** or Java method definition). 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

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 a recursive 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 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 `(define new-e (lambda (sym) (if (=? sym 'f) (eval rhs new-e) (e sym))))` where `eval` is the meta-interpreter. Scheme/Racket also includes a local recursive binding construct called `letrec`.

Scheme/Racket `letrec` is akin to `let` except that it performs recursive binding instead of conventional binding, *i.e.*, that the new environment created by `letrec` is used to evaluate all 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 is unavailable (sometimes represented by the error value `*void*`) until the evaluation of the `rhs` is complete. This trick works for `letrec` constructs that introduce new function definitions 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 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 **letrec** and **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.

Weakly Motivated Y

- Most interesting simple lambda-expression in LC:
 $((\text{lambda } x (x x)) (\text{lambda } x (x x)))$
This expression is typically called Ω
- In both call-by-value LC and call-by-name LC,
 $((\text{lambda } x (x x)) (\text{lambda } x (x x))) \rightarrow$
 $((\text{lambda } x (x x)) (\text{lambda } x (x x)))$
In other words, Ω reduces to itself.
- Key problem: must express the equivalent of recursive binding. Can we use a self application pattern like Ω to build an expanding tower of applications of a free variable f ? How about
 $((\text{lambda } x (f(x x))) (\text{lambda } x (f(x x))))?$
which reduces (in *call-by-name* LC) to:
 $(f ((\text{lambda } x (f(x x))) (\text{lambda } x (f(x x)))))$
which we enclose in a lambda-abstraction:
 $(\text{lambda } f (f ((\text{lambda } x (f(x x))) (\text{lambda } x (f(x x))))))$
to produce call-by-name Y .
- The least-fixed-point operator Y is applied to the abstracted body of the function we want to recursively define, yielding the least-fixed-point of the functional denoted by the abstracted body.
- For simplicity (call-by-value is uglier!), let's expand call-by-name LC to include the primitive operations required to express a simple definition of factorial:
 $\text{if-then-else, zero?, *, -}$
and use Scheme-like syntax (except for binding occurrences of variables) to explore an example.

Example: Factorial

- The body of a recursive definition of factorial (using **define** or **letrec** in Racket without parentheses around the abstracted variable **n**) is:

```
(lambda n [if (zero? n) 1 (* n (fact (- n 1)))])
```

and the corresponding functional is:

```
(lambda fact (lambda n [if (zero? n) 1 (* n (fact (- n 1)))]))
```

- In call-by-name LC,

```
(Y (lambda fact (lambda n [if (zero? n) 1 (* n (fact (- n 1)))]))) → (skipping a step)
```

```
((lambda fact (lambda n [if (zero? n) 1 (* n (fact (- n 1)))]))  
 ((lambda x ((lambda fact (lambda n [if (zero? n) 1 (* n (fact (- n 1)))])) (x x))  
  (lambda x ((lambda fact (lambda n [if (zero? n) 1 (* n (fact (- n 1)))])) (x x))))) →
```

```
(lambda n [if (zero? n) 1  
  (* n (((lambda x ((lambda fact (lambda n [if (zero? n) 1 (* n (fact (- n 1)))])) (x x))  
    (lambda x ((lambda fact (lambda n [if (zero? n) 1 (* n (fact (- n 1)))])) (x x)))))  
  (- n 1))])
```

which is a value.

- Let's apply this expression to the value: **1**:

```
((lambda n [if (zero? n) 1  
  (* n (((lambda x ((lambda fact (lambda n [if (zero? n) 1 (* n (fact (- n 1)))])) (x x))  
    (lambda x ((lambda fact (lambda n [if (zero? n) 1 (* n (fact (- n 1)))])) (x x)))))  
  (- n 1))])  
 1) →  
[if (zero? 2) 1 (* 1 (((lambda x ((lambda fact (lambda n [if (zero? n) 1 (* n (fact (- n 1)))])) (x x))  
  (lambda x ((lambda fact (lambda n [if (zero? n) 1 (* n (fact (- n 1)))])) (x x)))))  
  (- 1 1)))] → (skipping a step)  
(* 1 (((lambda x ((lambda fact (lambda n [if (zero? n) 1 (* n (fact (- n 1)))])) (x x))  
  (lambda x ((lambda fact (lambda n [if (zero? n) 1 (* n (fact (- n 1)))])) (x x)))))  
  (- 1 1))) →
```

Example: Factorial cont.

```
(* 1
  ((lambda n [if (zero? n) 1
    (* n (((lambda x ((lambda fact (lambda n [if (zero? n) 1 (* n (fact (- n 1))])) (x x))
      (lambda x ((lambda fact (lambda n [if (zero? n) 1 (* n (fact (- n 1))])) (x x)))))
      (- n 1)))]))
    (- 1 1))) →
(* 1
  [if (zero? (- 1 1)) 1
    (* (- 1 1) (((lambda x ((lambda fact (lambda n [if (zero? n) 1 (* n (fact (- n 1))])) (x x))
      (lambda x ((lambda fact (lambda n [if (zero? n) 1 (* n (fact (- n 1))])) (x x)))))
      (- (- 1 1) 1)))])) → (skipping two steps)
(* 1 1) →
1
```

- Let **FACT** abbreviate

```
(lambda fact (lambda n [if (zero? n) 1 (* n (fact (- n 1))]))
```

- Then **(Y FACT)** [in Scheme notation] equals (in one reduction step)

```
((lambda x ((lambda fact (lambda n (if (zero? n) 1 (* n (fact (- n 1))))) (x x))
  (lambda x ((lambda fact (lambda n (if (zero? n) 1 (* n (fact (- n 1))))) (x x)))))
```

which reduces to the value

```
(lambda n [if (zero? n) 1
  (* n (((lambda x ((lambda fact (lambda n [if (zero? n) 1 (* n (fact (- n 1))])) (x x))
    (lambda x ((lambda fact (lambda n [if (zero? n) 1 (* n (fact (- n 1))])) (x x)))))
    (- n 1)))]))
```

which is a “pure” lambda-abstraction denoting the least fixed-point of **FACT**

- Hence, in principle, we don’t need recursive binding constructs like **letrec** but software development would excruciatingly painful without them. Even the purest Haskellite takes such constructs for granted.

A Deeper Dive Into Functions as Data

- Computation is incremental—not monolithic.
- Slogan: general computation is successive approximation (typically in response to successive demands for more information) to a potentially infinite result.
- 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 computable, potentially infinite tree.

Mathematical Foundations

A *partially ordered set* (**po**) is a set S together with a binary relation \leq (a subset of $S \times S$) such that:

- \leq is *reflexive*: $\forall x \ x \leq x$.
- \leq is *anti-symmetric*: $\forall x, y \ x \leq y$ and $y \leq x$ implies that $x = y$.
- \leq is *transitive*: $\forall x, y, z \ x \leq y$ and $y \leq z$ implies that $x \leq z$.

A (Scott) *domain of computation* **D** (such as 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** (under the same relation as **D**) that is *finitely consistent*, i.e., closed under *least upper bounds* (LUBs) on finite bounded subsets (implying the existence of a *least* element \perp , which is 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**. All such elements are 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 $b_0 \leq b_1 \leq \dots \leq b_k \leq \dots$ (a countable ascending sequence) in **B** has a LUB in **D**.

Mathematical Foundations cont.

- A **po** with that is chain-complete is called a **cpo** (*complete partial order*).
- Every computational domain can be formalized as a Scott-domain.
- 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. So for computer scientists (who presumably are interested only in countably-based domains, the choice between using chains and directed sets is immaterial.
- A domain is *flat* iff all elements of the domain except \perp are maximal.

Examples of (Scott) domains:

- flat domains: integers, booleans, finite trees with no undefined (\perp) leaves, finite arrays, finite tables, finite graphs;
- lazy tree domains: potentially infinite trees with a finite set of node types and undefined (\perp) leaves. Each node type has a fixed arity. Any node constructor accommodating \perp as a leaf in some argument position is said to be *lazy*.

Key Mathematical Concepts

Computable functions on domains:

- monotonic with respect to approximation order \leq
- continuous (functions preserve the limits of chains)
- Typically but not necessarily strict (diverge if an argument diverges)

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

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

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

More Examples

Domains

- flat domains: all data types except functions and lazy algebraic constructions
- strict function spaces on flat domains (call-by-value)
- lazy trees of booleans
- continuous functions $\mathbf{A} \rightarrow \mathbf{B}$ where \mathbf{A} and \mathbf{B} are domains (the fully general case is very expensive to implement; call-by-name is not enough!)

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