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

Corky Cartwright

January 27, 2023

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 could use the `Y` operator construction instead, but it adds unacceptable overhead. So we will use environment mutation 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. In other words, 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))))
```

where `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 he *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` 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

Two syntactic ideas

- Most interesting simple lambda-expression in LC:
  `((lambda x (x x)) (lambda x (x x)))`
  This expression is typically called **Ω**; it diverges because it reduces via beta-reduction to itself (an identical expression).
  - In both call-by-value LC and call-by-name LC,
    `((lambda x (x x)) (lambda x (x x)))` → `((lambda x (x x)) (lambda x (x x)))`
  - Can we use a self-application pattern like **Ω** to build an expanding tower of applications of a free variable `g` bound to a function? How about
    `((lambda x (g(x x))) (lambda x (g(x x))))`?
    which reduces to:
    `(g ((lambda x (g(x x))) (lambda x (g(x x)))))`
    which reduces without terminating, generating progressively larger expressions of the form
    `(g (g ... ((lambda x (g(x x))) (lambda x (g(x x)))) ... ))`
- Recursion can be expressed as an infinite *lambda-abstraction*. Assume $f = E_f$ where all free occurrences of `f` inside the *lambda-abstraction* E_f are assumed equivalent to E_f. Then we deduce
  $(f\ x) = (E_f\ x) = (E_{E_f}\ x) = \ldots = (E_{E \ldots}\ x)$
  $f = (\text{lambda } x\ (E_f\ x)) = (\text{lambda } x\ (E_{E_f \ldots}\ x)) = \ldots = (\text{lambda } x\ (E_{E \ldots}\ x))$
- Since infinite lambda expressions are not legal programs, we have to construct this potentially infinite tree incrementally on demand so we produce an expansion just large enough to compute `f` for a given input `x`. No terminating application of `f` can use more than a finite "prefix" of the infinite expansion. Hence, for a given `x`, if `(f x)` terminates, only requires a finite expansion of the tree for `f` is required to compute the value of `(f x)`. How can we adaptively build the requiesite approximation?.

What is **Y**?

- Let **F = (lambda f E_f)**. E_f is an expression defining **f** in terms of itself. The recursive definition of factorial is a good example of such an expresion. If we augment LC with boolean constants **true** and **false** , the boolean function **zero?** that tests for **0**, the non-strict function **if-then-else** mapping **boolean × int × int → int**, the binary multiplication function *****, and the unary function **sub1** that subtracts **1** from its argument, then we can define

  FACT = (lambda fact (lambda n (if (zero? n) 1 (* n (fact (sub1 n))))).

  In this example, E_{fact} **= (lambda n (if (zero? n) 1 (* n (fact (sub1 n)))))**.

- Note that the reduction rules for **if** only evaluate the consequent or alternative argument but not both. Hence the sub-expression with the recursive call may not be evaluated in some cases. The function **(lambda f E_f)** maps any function **f** which serves as "dummy" seed into E_f which encloses calls on **f** in a wrapper expression E_f (typically a conditional expression) that includes control branches that terminate without applying **f**. If we construct a potentially infinite tower **T** of applications of a variable **g**, we only need to apply **T** to **(lambda f E_f)** (often called the functional corresponding to the recursive definition of **f**) to get the (least) solution to the definition **f = E_f**.

- We can easily define the infinite tower **T** by abstracting the tower-building expression from the previous slide with respect to the intended input variable **g**,

  (lambda g (g ((lambda x (g(x x))) (lambda x (g(x x))))))

  This is the standard **Y** operator. We will subsequently perform a similar derivation using more rigorous semantic tools.

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 (sub1 1)))])`
  and the corresponding functional is:
  `(lambda fact (lambda n [if (zero? n) 1 (* n (fact (sub1 1)))]))`
- In call-by-name LC,

```
(Y (lambda fact (lambda n [if (zero? n) 1 (* n (fact (sub1 n)))]))) → (skipping a step)

((lambda fact (lambda n [if (zero? n) 1 (* n (fact (sub1 n)))]))
 ((lambda x ((lambda fact (lambda n [if (zero? n) 1 (* n (fact (sub1 n)))])) (x x))
    (lambda x ((lambda fact (lambda n (if (zero? n) 1 (* n (fact (sub1 n)))))) (x x))))) →

(lambda n [if (zero? n) 1
              (* n (((lambda x ((lambda fact (lambda n [if (zero? n) 1 (* n (fact (sub1 n)))])) (x x))
                       (lambda x ((lambda fact (lambda n [if (zero? n) 1 (* n (fact (sub1 n)))])) (x x)))))
                   (sub1 n)))])
```

  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 (sub1 n)))])) (x x))
                        (lambda x ((lambda fact (lambda n [if (zero? n) 1 (* n (fact (sub1 n)))])) (x x)))))
                    (sub1 n)))])
 1)  →
[if (zero? 2) 1 (* 1 (((lambda x ((lambda fact (lambda n [if (zero? n) 1 (* n (fact (sub1 n)))])) (x x))
                         (lambda x ((lambda fact (lambda n [if (zero? n) 1 (* n (fact (sub1 n)))])) (x x)))))
                     (- 1 1)))] → (skipping a step)
(* 1 (((lambda x ((lambda fact (lambda n [if (zero? n) 1 (* n (fact (sub1 n)))])) (x x))
                    (lambda x ((lambda fact (lambda n [if (zero? n) 1 (* n (fact (sub1 n)))])) (x x)))))
      (sub1 1))) →
```

Example: Factorial cont.

```
(* 1
   ((lambda n [if (zero? n) 1
                  (* n (((lambda x ((lambda fact (lambda n [if (zero? n) 1 (* n (fact (sub1 n)))])) (x x))
                           (lambda x ((lambda fact (lambda n [if (zero? n) 1 (* n (fact (sub1 n)))])) (x x)))))
                       (sub1 n)))])
    (sub1 1))) →
(* 1
   [if (zero? (sub1 1)) 1
       (* (sub1 1) (((lambda x ((lambda fact (lambda n [if (zero? n) 1 (* n (fact (sub1 n)))])) (x x))
                           (lambda x ((lambda fact (lambda n [if (zero? n) 1 (* n (fact (sub1 n)))])) (x x)))))
                       (sub1 (sub1 1)))]) → (skipping two steps)
(* 1 1) →
1
```

- Let **FACT** abbreviate
  (lambda fact (lambda n [if (zero? n) 1 (* n (fact (sub1 n)))]))
- Then **(Y FACT)** [in Scheme notation] equals (in one reduction step)

```
((lambda x ((lambda fact (lambda n (if (zero? n) 1 (* n (fact (sub1 n)))))) (x x))
   (lambda x ((lambda fact (lambda n (if (zero? n) 1 (* n (fact (sub1 n)))))) (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 (sub1 n)))])) (x x))
                       (lambda x ((lambda fact (lambda n [if (zero? n) 1 (* n (fact (sub1 n)))])) (x x)))))
                   (sub1 n)))])
```

  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 Haskelite 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 produce 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*. This is a deep technical result that is largely unknown (even by most theoretical computer scientists). See the paper Observable Sequentiality and Full Abstraction for more details.

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 $b_0 \leq b_1 \leq \ldots \leq b_k \leq \ldots$ (a countable ascending sequence) in **B** has a LUB in **D**.

Building intuition regarding Scott domains: *draw* or *visualize the Hasse diagrams of the finite elements* of the domains in question. There are many good online references. See, for example, this page.

Mathematical Foundations cont.

- A **po** with that is chain-complete is called a **cpo** (*complete partial order*). So every Scott Domain is a countably-based **cpo**. We are only interested in finitely founded Scott domains, where every finite element has only finitely many elements below it. Every such domain can be represented as a set of lazy trees (in the sense that nodes can have undefined [⊥] children).
- Every computational domain can be formalized as a Scott-domain.
- In the reference monograph (derived from a monograph by Dana Scott), 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 ⊥ are maximal. In such a domain, all elements are *finite*.
- The binary relation ≤ intuitively corresponds to *approximation* in the sense that $a \leq b$ iff a is a tree prefix of b. The binary relation symbol is often written as 2291.

Examples of (Scott) domains:

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

Key Mathematical Concepts

Computable functions on domains:

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

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

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

For an in-depth treatment of (Scott) domains, see the monog
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 **A** → **B** where **A** and **B** are domains (the fully general case is very expensive to implement; call-by-name is not enough; the sequential subset of all production languages only supports observably sequential functions).

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