Comp 411
Principles of Programming Languages
Lecture 11
The Semantics of Recursion II

Corky Cartwright

February 4, 2022

Recursive Definitions

Given a Scott-domain **D**, we can write equations of the form:

$\mathbf{f = E_f}$ [Note: $\mathbf{f(x_1,...,x_n) = M_f} \Leftrightarrow \mathbf{f = \lambda\ x_1,...,x_n\ .\ M_f}$]

where $\mathbf{E_f}$ is an expression constructed from constants in **D**, operations (continuous functions) on **D**, and variables.

Example: let **D** be the domain of Jam values. Then .

```
fact = map n to if n = 0 then 1 else n * fact(n - 1)
```

is such an equation.

Equations of this form are called *recursive definitions*.

Solutions to Recursion Equations

- Given a recursion equation:

  $\mathtt{f = E_f}$

  what is a solution? All of the constants and operations in $\mathtt{E_f}$ are known except $\mathtt{f}$ and all variables other than $\mathtt{f}$ are explicit parameters that have values (or potential values in the case of call-by-name provided as inputs). All functions in $\mathtt{E_f}$ are continuous.
- A solution to this equation is any continuous function $\mathtt{f}$ such that $\mathtt{f = E_f}$, or alternatively is a fixed point of the function(al) $\lambda\mathtt{f.E_f}$.
- But there may be more than one solution. We want to select the *best* solution $\mathtt{f^*}$. Note that $\mathtt{f^*}$ is an element of whatever domain $\mathtt{D^*}$ corresponds to the type of $\mathtt{E_f}$. In the most common case, it is $\mathtt{D} \rightarrow \mathtt{D}$, but it can be $\mathtt{D}$, $\mathtt{D} \rightarrow \mathtt{D}$, . . . , $\mathtt{D}^k \rightarrow \mathtt{D}$, ... The best solution $\mathtt{f^*}$ (which always exists and is unique and *computable* for a any domain in $\mathtt{D^*}$) is the *least* solution under the approximation ordering in $\mathtt{D^*}$.

Constructing the Least Solution

How do we know that any solution exists to the equation $f = E_f$? We will construct the least solution and prove it is a solution!

Since the domain D^* for f is a Scott-Domain, this domain has a least element $\perp_{D^*}$ that approximates every solution to the equation.

Now form the function $F: D^* \to D^*$ defined by $F(f) = E_f$, or equivalently, $F = \lambda f.E_f$ where $\lambda f.E_f$ is *monotonic* and *continuous* (by a lemma we skipped). Note that for a recursive definition of a function, F is a *functional*.

Consider the sequence S: $\perp_{D^*}, F(\perp_{D^*}), F(F(\perp_{D^*})), \ldots, F^k(\perp_{D^*}), \ldots$

Claim: S is an ascending chain (chain for short) in $D^* \to D^*$.

Proof. $\perp_D \leq F(\perp_{D^*})$ by the definition of $\perp_D$. If $M \leq N$ then $F(M) \leq F(N)$ by monotonicity. Hence, $F^k(\perp_D) \leq F(F^k(\perp_D))$ by induction on k. Q.E.D.

Claim: S has a least upper bound f^*.

Proof. Trivial. S is a chain in D^* and hence must have a least upper bound because D^* is a Scott-Domain. If D^* is a function domain, then f^* is continuous by definition.

Proving $\mathbf{f^*}$ is a fixed point of $\mathbf{F}$

Must show: $F(f^*) = f^*$ where $F = \lambda f.E_f$

Claim: By definition $f^* = \sqcup F^k(\bot_{D^*})$ Since F is continuous

$$F(f^*) = F(\sqcup F^k(\bot_{D^*})) = \sqcup F^{k+1}(\bot_{D^*}) = \sqcup F^k(\bot_{D^*})$$
$$= f^* .$$

Note: The second step above relies on the continuity of F and the third depends on the fact that $F^0(\bot_{D^*}) = \bot_{D^*} \leq F(\bot_{D^*})$.

Q.E.D.

Example

Look at factorial in detail by running the DrRacket stepper or conceptualizing strict continuous functions mapping $\mathbb{N}$ into $\mathbb{N}$ where is the domain natural numbers including $\bot$, which can be represented as graphs (sets of pairs) over $\mathbb{N}$-{$\bot$}. The same observation applies to the domain of Jam values which includes $\mathbb{N}$ as a subdomain.

How Can We Compute f^* Given F?

- Need to construct $F^\infty(\bot)$ from F. Can we write code for a function Y such that $Y(F) = f^* = F^\infty(\bot)$.
- Idea: use syntactic trick well known in the λ-calculus to build a potentially infinite stack of Fs, based on an understanding of how evaluation of $\Omega = (\lambda x.(x\ x))\ (\lambda x.(x\ x))$ works.
- Preliminary attempt: $Y(F) = (\lambda x.\ F(x\ x))\ (\lambda x.\ F(x\ x))$
- Reduces to (in one step) to: $F((\lambda x.\ F(x\ x))\ (\lambda x.\ F(x\ x)))$
- Reduces to (in k steps) to: $F^k((\lambda x.\ F(x\ x))\ (\lambda x.\ F(x\ x)))$

How does the Code for Y Work?

In Haskell (or other language with call-by-name)

```
Y = λF. (λx. F(x x)) (λx. F(x x))
```

Hence, `Y(FACT)`

```
= (λx.FACT(x x))(λx.FACT(x x))
= FACT((λx.FACT(x x))(λx.FACT(x x)))
= λn. if n=0 then 1        ; only valid in Call-By-Name!
      else n*((λx.FACT(x x))(λx.FACT(x x)))(n-1)
```

implying `Y(FACT)` reduces to a value!

Does this work for Scheme (or Java with an appropriate encoding of functions as anonymous inner classes)? No! Why not? What about divergence? `Y(FACT)`

```
= (λx.FACT(x x))(λx.FACT(x x))
= FACT((λx.FACT(x x))(λx.FACT(x x)))
= FACT(FACT(...))
```

diverging like Ω) but growing with each reduction

Why Does Call-by-name `Y` Work?

By assumption the functional `G` corresponding to a recursive function definition must have the form `λf. λn. M`. Hence,

```
  (λF.((λx.F(x x)) (λx.F(x x)))) G
= G ((λx.G(x x)) (λx.G(x x)))
= (λf.λn.M) ((λx. G(x x)) (λx.G(x x)))
= λn.M[f ← (λx. G(x x)) (λx.G(x x))]
```

which is a value. If the evaluation of `M` does not require evaluating an occurrence of `f`, then `(λx. G(x x)) (λx.G(x x))` is not evaluated. Otherwise, the binding of `x` is unwound only as many times as required to get to the base case in the definition `f = λn.M`.

Exercise: How can we workaround this problem to create a version of the `Y` operator that works for call-by-value Scheme and Jam?

Why Does Call-by-name `Y` Work?

By assumption the functional `G` corresponding to a recursive function definition must have the form `λf. λn. M`. Hence,

```
  (λF.((λx.F(x x))(λx.F(x x))))G
= G((λx.G(x x))(λx.G(x x)))
= (λf.λn.M)((λx.G(x x))(λx.G(x x)))
= λn.M[f ← (λx.G(x x))(λx.G(x x))]
```

which is a value. If the evaluation of `M` does not require evaluating an occurrence of `f`, then `(λx.G(x x))(λx.G(x x))` is not evaluated. Otherwise, the binding of `x` is unwound only as many times as required to get to the base case in the definition `f = λn.M`. But each unwinding requires a few reduction steps, so this definition is a poor way to implement recursion!

Exercise: how can we workaround this problem to create a version of the `Y` operator that works for call-by-value Scheme and Jam? See the next lecture.