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

Corky Cartwright

February 7, 2014

Recursive Definitions

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

  $$\mathbf{f} \stackrel{\text{def}}{=} \mathbf{E}_{\mathbf{f}}$$

  where $\mathbf{E}_{\mathbf{f}}$ is an expression constructed from constants in **D**, operations (continuous functions) on **D**, and **f**.
- 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.
- Such equations are called *recursive definitions.*

Solutions to Recursion Equations

Given a recursion equation:

$$\mathbf{f} \stackrel{\text{def}}{=} \mathbf{E}_{\mathbf{f}}$$

what is a solution? All of the constants and operations in $\mathbf{E}_{\mathbf{f}}$ are known except $\mathbf{f}$.

A solution is any function $\mathbf{f}$ such that $\mathbf{f} = \mathbf{E}_{\mathbf{f}}$.

But there may be more than one solution. We want to select the "best" solution $\mathbf{f^*}$. Note that $\mathbf{f^*}$ is an element of whatever domain $\mathbf{D^*}$ corresponds to the type of $\mathbf{E}_{\mathbf{f}}$. In the most common case, it is $\mathbf{D \rightarrow D}$, but it can be $\mathbf{D}$, $\mathbf{D \rightarrow D}$, . . . , $\mathbf{D^k \rightarrow D}$, ... The best solution $\mathbf{f^*}$ (which always exists and is unique and *computable*) is the *least* solution under the approximation ordering in $\mathbf{D^*}$.

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,

$$F = \lambda f.E_f$$

Consider the sequence S: $\perp_D$, $F(\perp_D)$, $F(F(\perp_D))$, . . . , $F^k(\perp_D)$, . . .

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

Proof. $bot_{D*} \le F(bot_{D*})$ by the definition of Bot_{D*} . If $M \le N$, then $F(M) \le F(N)$ by monotonicity. Hence, $F^k(bot_D) \le F^{k+1}(bot_D)$ for all 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.

Proving f^* is a fixed point of F

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

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

$$
\begin{aligned}
F(f^*) &= F(\bigcup F^k(\bot_{D^*})) \\
&= \bigcup F^{k+1}(\bot_{D^*}) && \text{(by continuity)} \\
&= \bigcup F^k(\bot_{D^*}) && (\text{since } \bot_{D^*} \le F(\bot_{D^*})) \\
&= f^*
\end{aligned}
$$

Q.E.D.

Note: all of the steps in the preceding proof are trivial except for the step justified by continuity.

Examples

Look at factorial in detail using DrRacket stepper.

How Can We Compute f^* Given F?

Need to construct $F^{\infty}(\perp)$ from F. Let

$$Y(F) = f^* = F^{\infty}(\perp).$$

Can we write code for Y?

Idea: use syntactic trick in Ω to build a potentially infinite stack of Fs.

- Preliminary attempt:

$$(\lambda x.\ F(x\ x))\ (\lambda x.\ F(x\ x))$$

- Reduces to (in one step):

$$F\ ((\lambda x.\ F(x\ x))\ (\lambda x.\ F(x\ x)))$$

- Reduces to (in k steps):

$$F^k\ ((\lambda x.\ F(x\ x))\ (\lambda x.\ F(x\ x)))$$

What Is the Code for **Y**?

$\lambda F.\ (\lambda x.\ F(x\ x))(\lambda x.\ F(x\ x))$

- Does this work for Scheme (or Java with an appropriate encoding of functions as anonymous inner classes)? No!
- Why not? What about divergence? Assume **G** is a λ-expression defining a functional like **FACT**

$(\lambda F.(\lambda x.\ F(x\ x))(\lambda x.\ F(x\ x)))G$
$=\ G((\lambda x.\ G(x\ x))(\lambda x.\ G(x\ x)))$
$=\ \ldots$ (diverging)

What If We Use Call-by-name?

By assumption **G** must have the form `λf.λ n . M`

```
(λ F.  (λx. F(x x))(λx. F(x x))) G
  = G ((λx. G(x x))(λx. G(x x)))
  = (λ f.λ n . M) (( λx. G(x x))(λx. G(x x)))
  = λ n .M[x:=(λx.G(x x))(λx.G(x x))]
```

If the evaluation **M** of does not require evaluating an occurrence of **f** , then **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? Hint: if **M** is a divergent term denoting a unary function `λx.Mx` is an "equivalent" term that is not divergent! (As a concrete example, assume that **M** is the term Ω.)