Comp 311
Principles of Programming Languages
Lecture 11
The Semantics of Recursion II

Corky Cartwright
Swarat Chaudhuri
September 16, 2010
Recursive Definitions

• Given a Scott-domain $D$, we can write equations of the form:
  \[ f = E_f \]
  where $E_f$ is an expression constructed from constants in $D$, operations (continuous functions) on $D$, and $f$.

• Example: let $D$ be the domain of Scheme unary functions on numbers. Then
  \[
  \text{fact} = \\
  (\lambda (n) (\text{if} \ (\text{zero?} \ n) \ 1 \ (* \ n \ (\text{fact} \ (- \ n \ 1)))))
  \]
  is such an equation.
Solutions to Recursion Equations

Given an equation:

\[ f = E_f \]

what is a solution? All of the constants and operations in \( E_f \) are known except \( f \).

A solution is any function \( f^* \) such that

\[ f^* = E_{f^*} \]

is a solution. But there may be more than one solution. We want to select the “best” solution. Note that \( f^* \) is an element of whatever domain \( D^* \) is the type of \( E_f \). In the most common case, it is \( D \to D \), for a domain of values \( D \), but it can be \( D, D^k \to D, \ldots \) The best solution (the one that always exists, is unique, and is computable) is the least solution under the approximation ordering in \( D^* \).
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, it has a least element \( \text{bot}_{D^*} \). Hence, \( \text{bot}_{D^*} \) approximates every solution to the equation \( f = E_f \).

Now form the function \( F: D^* \to D^* \) defined by
\[
F(f) = E_f
\]
or equivalently,
\[
F = \lambda f . E_f
\]
Consider the sequence \( S: \text{bot}_{D^*}, F(\text{bot}_{D^*}), F(F(\text{bot}_{D^*})), ..., F^k(\text{bot}_{D^*}), ... \).

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

Proof. \( \text{bot}_{D^*} \leq F(\text{bot}_{D^*}) \) by the definition of \( \text{bot}_{D^*} \). If \( M \leq N \), then \( F(M) \leq F(N) \) by monotonicity. Hence, \( F^k(\text{bot}_D) \leq F^{k+1}(\text{bot}_D) \) 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 \( f^* = \bigcup F^k(\text{bot}_{D^*}) \). Since \( F \) is continuous

\[
F(f^*) = F(\bigcup F^k(\text{bot}_{D^*})) \\
= \bigcup F^{k+1}(\text{bot}_{D^*}) \quad \text{(by continuity)} \\
= \bigcup F^k(\text{bot}_{D^*}) \quad \text{(since \( \text{bot}_{D^*} \leq F(\text{bot}_{D^*}) \))} \\
= f^*
\]

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 DrScheme.
How Can We Compute $f^*$ Given $F$?

Need to construct $F^\infty(\bot)$ from $F$ using only abstraction and application. We need to define an operator $Y$ such that:

$$Y(F) = f^* = F^\infty(\bot).$$

Idea: use syntactic trick in $\Omega$ to build a potentially infinite stack of $F$s.

• 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 \( \lambda \)-expression defining a functional like \( \text{FACT} \)

\[ (\lambda F. (\lambda x. F(x x)) (\lambda x. F(x x))) G \]

\[ = G((\lambda x. G(x x)) (\lambda x. G(x x))) \]

\[ = \ldots \text{ (divergence forced by CBV)} \]
What If We Use Call-by-name?

By assumption \( G \) must have the form \((\lambda f. (\lambda n. M))\)

\[
(\lambda F. (\lambda x. F(x x)) (\lambda x. F(x x))) \ G
\]

\[
=> (\lambda x. G(x x)) (\lambda x. G(x x)) \quad <**>
\]

\[
=> G \quad <**>
\]

\[
= (\lambda f. (\lambda n. M)) \quad <**>
\]

\[
=> (\lambda n. M[f:=<**>]) \quad <**>
\]

which is a value. If this value \(<**>\) is applied to a value \( k \) and \( M[f:=<**>][n:=k] \) does not require evaluating an occurrence of \(<**>\), then the computation returns a base answer determined by \( M \). Otherwise, \(<**>\) is unwound once, as in the computation above to produce \(<**>\) applied to its argument. If the argument is less than \( k \) (in some well-founded ordering) this process eventually terminates when \( k \) reaches a value that does not force the evaluation of \(<**>\). At this point, the subcomputation \(<**> \ b\) returns a base value and the enclosing computation (not involving recursive calls \(<**>\)) is performed, returning a value. If the top-level application of

**Exercise:** how can we workaround the divergence problem to create a version of the \( \mathbf{Y} \) operator that works for call-by-value Scheme and Jam? Hint: if \( N \) is a divergent term denoting a unary function, then \( \lambda x. Nx \) is an “equivalent” term that is not divergent (assuming \( x \) does occur in \( N \)).