Recursive Definitions

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

$$f = E_f \quad \text{[Note: } f(x_1, \ldots, x_n) = M_f \iff f = \lambda x_1, \ldots, x_n . M_f\text{]}$$

where $E_f$ 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

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

Now form the function $F: D^* \rightarrow D^*$ 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 the recursive definition of a function, $F$ is a functional.

Consider the sequence $S$: $\bot_{D^*}$, $F(\bot_{D^*})$, $F(F(\bot_{D^*}))$, $F^k(\bot_{D^*})$, ... Claim $S$ is an ascending chain (chain for short) in $D^* \rightarrow D^*$.

**Proof.** $\bot_D \leq F(\bot_{D^*})$ by the definition of $\bot_D$. If $M \leq N$ then $F(M) \leq F(N)$ by monotonicity. Hence, $F^k(\bot_D) \leq F(F^k(\bot_D))$ 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 $f^*$ is a fixed point of $F$

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 $D$ into $D$ which can be represented as graphs (sets of pairs) over $D - \{\perp_D\}$. Recall that $D$ is the domain of Jam values.
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 $\lambda$-calculus 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) to:
  $F((\lambda x. F(x \, x)) \, (\lambda x. F(x \, x)))$
- Reduces to (in $k$ steps) to:
  $F^k((\lambda x. F(x \, x)) \, (\lambda x. F(x \, x)))$
What Is the Code for Y?

In Haskell (or other language with call-by-name)
\[ Y = \lambda F. (\lambda x. F(x \ x))(\lambda x. F(x \ x)) \]

Hence,
\[ Y(\text{FACT}) \]
\[ = (\lambda x. \text{FACT}(x \ x))(\lambda x. \text{FACT}(x \ x)) \]
\[ = \text{FACT}((\lambda x. \text{FACT}(x \ x))(\lambda x. \text{FACT}(x \ x))) \]
\[ = \lambda n. \text{if } n=0 \text{ then 1 else } n^*((\lambda x. \text{FACT}(x \ x))(\lambda x. \text{FACT}(x \ x))) \]

implying \( Y(\text{FACT}) \) 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? \( Y(\text{FACT}) \)
\[ = (\lambda x. \text{FACT}(x \ x))(\lambda x. \text{FACT}(x \ x)) \]
\[ = \text{FACT}((\lambda x. \text{FACT}(x \ x))(\lambda x. \text{FACT}(x \ x))) \]
\[ = \text{FACT}((\text{FACT}(\ldots)) \text{ (diverging like } \Omega) \]


Why Does Call-by-name Work?

By assumption \( G \) corresponding to a recursive function definition must have the form \( \lambda f. \lambda n. M \). Hence,

\[
(\lambda F.((\lambda x.F(x \ x)) (\lambda x.F(x \ x)))) \ G
= G ((\lambda x.G(x \ x)) (\lambda x.G(x \ x)))
= (\lambda f.\lambda n.M) ((\lambda x. G(x \ x)) (\lambda x.G(x \ x)))
= \lambda n.M[f \leftarrow (\lambda x. G(x \ x)) (\lambda x.G(x \ x))]\]

which is a value. If the evaluation of \( M \) does not require evaluating an occurrence of \( f \), then \( (\lambda x. G(x \ x)) (\lambda x.G(x \ x)) \) 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 \( f = \lambda n. M \).

**Exercise:** how can we workaround this problem to create a version of the \( \text{Y} \) operator that works for call-by-value Scheme and Jam? Hint: if \( M \) is a divergent term denoting a unary function \( \lambda x.Mx \) (where \( x \) is not free in \( M \)) is an “equivalent” term called the *eta* \([\eta]\) conversion of \( M \) that is not divergent! As a concrete example, assume that \( M \) is the term \( \Omega \).