Call-by-name vs. Call-by-value
Fixed-Point Operators

Given a recursive definition in a call-by-value language

\[ f \equiv E_f \]

where \( E_f \) is an expression constructed from constants in the base language and \( f \). What does it mean?

Example: let \( D \) be the domain of Scheme values. Then the base operations are continuous functions on \( D \) and

\[ \text{fact} \equiv \]

\[ \text{map } n \text{ to if } n = 0 \text{ then } 1 \text{ else } n * \text{fact}(n - 1) \]

is a recursive definition of a function on \( D \).

In a call-by-name language (\( \text{map } n \text{ to } \ldots \) is interpreted using call-by-name), the meaning of \( \text{fact} \) is

\[ Y(\text{map } f \text{ to } E_f) \]

What if \( \text{map} \) (\( \lambda \)-abstraction) has call-by-value semantics?
Defining \( Y \) in a Call-by-value Language

We want to define \( Y_v \), a call-by-value variant of \( Y \).

Key trick: use \( \eta \) (eta)-conversion to delay the evaluation.

In the mathematical literature on the \( \lambda \)-calculus, \( \eta \)-conversion is often assumed as an axiom. In models of the pure \( \lambda \)-calculus, it typically holds.

Definition: \( \eta \)-conversion is the following equation:

\[
M = \lambda x . Mx
\]

where \( x \) is not free in \( M \). If the \( \lambda \)-abstraction used in the definition of \( Y \) has call-by-value semantics, then given the functional \( F \) corresponding to recursive function definition, the computation \( YF \) diverges. We can prevent this from happening by \( \eta \)-converting both occurrences of \( F(x x) \) within \( Y \).
What Is the Code for $Y_v$?

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

- Does this work for Scheme (or Java with an appropriate encoding of functions as anonymous inner classes)? Yes!
- Let $G$ be some functional $\lambda f. M$, like $\text{FACT}$, for a recursive function definition. $G$ and $M$ are values ($\lambda$-expressions). Then

$Y_v G = \lambda x. (\lambda y. (G(x x)) y) (\lambda y. (G(x x)) y) = \lambda y. (G((\lambda y. (G(x x)) y) (\lambda y. (G(x x)) y)) y$ is a value.

- Hence, $G(Y_v G) = (\lambda f. M)(Y_v G) = M[f:=Y_v G]$, which is a value.
- It is straightforward to prove (using conversion rules) that

$Y_v G = G(Y_v G)$
Loose Ends

- Meta-errors
- Read the notes!
- rec-let (in notes)