Call-by-name vs. Call-by-value Fixed-Points

Given a recursive definition \( f \equiv E_f \) in a call-by-value language 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

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

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

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

\[
Y(\text{map } \text{fact} \text{ to } E_{\text{fact}})
\]

What if \( \text{map} \ (\lambda\text{-abstraction}) \) has call-by-value semantics? \( Y \) does not quite work because evaluations of form \( Y(\text{map } f \text{ to } E_f) \) diverge.
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:

$$M = \lambda x . \ M x$$

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 $Y_v$?

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

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

  is a value. In call-by-value, $Y \ G$ is not a value.
- But $G(Y_v \ G) = (\lambda f. \lambda n. M)(Y_v \ (\lambda f. \lambda n. M)) = \lambda n. M[f:=Y_v(\lambda f. \lambda n. M)]$, which is a value.
- As shown above (using call-by-value $\beta$-conversion) $Y_v \ G = G(Y_v \ G)$ where $G$ is any closed functional $\lambda f. \lambda n. M$.
- Disadvantage of $Y_v$ vs. $Y$: $Y_v$ is arity-specific for recursive function definitions in languages like Jam that support multiple arguments in $\lambda$-expressions. (Note: it works for all curried function definitions.)
Loose Ends

• Meta-errors
• Read the notes!
• letrec (in notes)
Lazy JamVal: a Concrete Example

Consider Jam with call-by-value $\lambda$ and lazy \texttt{cons}. What is the domain of data values? It consists of the flat domain of integers $\mathbb{Z}_\perp$ augmented by lazy lists over \texttt{JamVal}s:

\begin{align*}
\text{JamVal} & = \mathbb{Z}_\perp + \text{JamList} \\
\text{JamList} & = \text{JamEmpty} + \text{cons}(\text{JamVal}, \text{JamList})
\end{align*}

where \texttt{cons} is lazy (non-strict) in both arguments. Can we write a form of $Y$ for recursively defining infinite non-empty \texttt{JamList}s? Yes!
Call-by-value $Y$ for lazy lists

Recall the trick we used to define $Y$ for call-by-value function definitions. We had to $\eta$-convert the application $F(x \ x)$ inside $Y$ to make it a value. In this case, we similarly need to syntactically convert $F(x \ x)$ to an equivalent expression that is a value. In this case $F(x \ x)$ denotes a potentially infinite non-empty JamList. Hence, we need to convert $F(x \ x)$ to

$$\text{cons}(\text{first}(F(x \ x)), \text{rest}(F(x \ x)))$$

which is a value (corresponding to $\text{cons}(\perp, \perp)$). Hence, $Y_{\text{cons}}$ is $\lambda F. G(G)$ where

$$G = \lambda x. \text{cons}(\text{first}(F(x \ x)), \text{rest}(F(x \ x)))$$

Try it in the Reference Interpreter for Assignment 3.
Sample Assignment 3 Evaluations

let Ycons := map F to
   let g := map x to cons(first(F(x(x))),rest(F(x(x))));
   in g(g);
in first(Ycons(map zeros to cons(0, zeros)))
⇒ 0

let Ycons := map F to
   let g := map x to cons(first(F(x(x))),rest(F(x(x))));
   in g(g);
Yv := map F to
   let g := map x to map y to (F(x(x)))(y);
   in g(g);
in let
   sum := Yv(map s to map l to map k to
              if k = 0 then 0
              else first(l) + (s(rest(l))(k-1));
   countup := Yv(map cu to map k to cons(k, cu(k+1));
   in (sum(countup(1)))(10)
⇒ 55