Recursion

- Observe that EXP_1 — as currently defined — has no recursion:

Ex: Let `foo` be bound to $\lambda x.0$ in the environment $u[\texttt{foo} \mapsto \lambda x.0]$. Consider the evaluation of the following expression:

$$\mathit{evaluate}[\![\, \textbf{let fun } \texttt{foo(n: } \textbf{int} \texttt{) =}$$
$$\textbf{if } \texttt{n=0 } \textbf{then } \texttt{0 } \textbf{else } \texttt{n + foo(n−1)}$$
$$\textbf{in } \texttt{foo(3)} \,]\!]\, (u[\texttt{foo} \mapsto \lambda x.0])$$

$$= \mathit{evaluate}[\![\texttt{foo(3)}]\!]\, (u[\texttt{foo} \mapsto \lambda x.0,\ \texttt{foo} \mapsto f])$$

where

$$f = \lambda a.\mathit{evaluate}[\![\textbf{if } \texttt{n=0 } \textbf{then } \texttt{0 } \textbf{else } \texttt{n + foo(n−1)}]\!]$$
$$(u[\texttt{foo} \mapsto \lambda x.0,\ \texttt{n} \mapsto a])$$

$$= \lambda a.\textbf{if } a = 0 \textbf{ then } 0 \textbf{ else } a + (\lambda x.0)(a-1)$$

$$= \lambda a.\textbf{if } a = 0 \textbf{ then } 0 \textbf{ else } a$$

$$= \lambda a.a$$

Thus: $\mathit{evaluate}[\![\texttt{(foo(3)}]\!]\, (u[\texttt{foo} \mapsto \lambda x.0,\ \texttt{foo} \mapsto f])$

$$= \mathit{evaluate}[\![\texttt{(foo(3)}]\!]\, (u[\texttt{foo} \mapsto f])$$

$$= f(3) = (\lambda a.a)\ 3 = 3$$

- *First* `foo` is newly introduced symbol, defined in terms of *second* `foo` — which is a pre-existing symbol in environment with a *different* binding.
- Analogous to **let val** `x = x * 2` **in** $\cdots$ — not recursive!

Recursion (cont.)

- To obtain recursion, have to assure that *both* occurrences of `foo` are bound to the *same* (not previously defined & as yet unknown) *function*.
- Thus `foo` will be bound to $f*$ where:

$$f* = \lambda a.evaluate[\![\texttt{if n=0 then 0 else n + foo(n-1)}]\!](u[\texttt{foo} \mapsto f*, \texttt{n} \mapsto a])$$

$$= \lambda a.\textbf{if } a = 0 \textbf{ then } 0 \textbf{ else } a + f*(a-1)$$

- This last equation is is a *fixed point equation* of the form

$$f* = \tau\, f*$$

where τ is a *functional* given by

$$\tau = \lambda g.\lambda z.\textbf{if } z = 0 \textbf{ then } 0 \textbf{ else } z + g(z - 1)$$

- Note that the functional τ is a ''function transformer'':

$$\tau : (\texttt{int} \to \texttt{int}) \to (\texttt{int} \to \texttt{int})$$

- Scott: $f*$ is **defined** by the fixed point equation $f* = \tau\, f*$, where $\tau = \lambda g.\lambda z. \cdots$ is a *functional* derived from the body of the recursive definition.
- What is $f*$ for this example? What function $f*$ makes the equation $f* = \tau\, f*$ ''balance''?

$$f* = \begin{cases} \lambda n.\_\_\_\_\_\_\_\_ & \textbf{if } n \geq 0 \\ \lambda n.\bot & \textbf{if } n < 0 \end{cases}$$

- Now what is the value of the program (expression)?

$$evaluate[\![\texttt{foo(3)}]\!](u[\texttt{foo} \mapsto f*]) = f*(3)$$

$$= \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_$$

Recursive Definition

- ML:

```
- fun fact(n: int) = if n=0 then 1 else n*fact(n-1);
- fact(3);
```

- Scheme:

```
>> (define (fact n)
           (if (= n 0)  1  (* n (fact (-1+ n)))))
>> (fact 3)
```

  — Two kinds of `let` clause in Scheme: `(let ... )` for non-recursive definition and `(letrec ... )` for recursive. Top-level definitions (as above) are assumed to be recursive.

- Define $\mathrm{EXP}_2 \triangleq \mathrm{EXP}_1 +$ recursion + conditional expressions:

  — Add syntax

```
Declaration ::=...
          | recfun Identifier (Formal-Parameter)
               = Expression
```

  — example

```
let recfun fact(n:int ) =
          if n = 0 then 1 else n * fact(n – 1)
in fact(3)
```

- In each case, what is the *meaning* of the "body" or RHS `B` of the recursive definition?

  — a *functional* that transforms a function to a function

  — $\tau = \lambda f.\mathit{evaluate}[\![\mathtt{B}]\!](u[\mathtt{fact} \mapsto f])$
  $= \lambda f.\lambda x.\mathbf{if}\ x = 0\ \mathbf{then}\ 1\ \mathbf{else}\ x \cdot f(x - 1)$

Recursive Definition (cont.)

Ex:

$$\tau\,(\lambda x.x+1)$$
$$\begin{aligned} &= \lambda x.\ \textbf{if}\ x=0\ \ \textbf{then}\ 1\ \ \textbf{else}\ x\cdot(\lambda z.z+1)(x-1) \\ &= \lambda x.\ \textbf{if}\ x=0\ \ \textbf{then}\ 1\ \ \textbf{else}\ x^2 \\ &= (\lambda x.x^2)[0 \mapsto 1] \end{aligned}$$

$$\tau\,(\lambda x.x)$$
$$\begin{aligned} &= \lambda x.\ \textbf{if}\ x=0\ \ \textbf{then}\ 1\ \ \textbf{else}\ x\cdot(\lambda z.z)(x-1) \\ &= \lambda x.\ \textbf{if}\ x=0\ \ \textbf{then}\ 1\ \ \textbf{else}\ x\cdot(x-1) \\ &= (\lambda x.x^2-x)[0 \mapsto 1] \end{aligned}$$

$$\tau\,(\lambda x.1)$$
$$\begin{aligned} &= \lambda x.\ \textbf{if}\ x=0\ \ \textbf{then}\ 1\ \ \textbf{else}\ x\cdot(\lambda z.1)(x-1) \\ &= \lambda x.\ \textbf{if}\ x=0\ \ \textbf{then}\ 1\ \ \textbf{else}\ x\cdot 1 \\ &= (\lambda x.x)[0 \mapsto 1] \end{aligned}$$

$$\tau\,(\lambda x.x!)$$
$$\begin{aligned} &= \lambda x.\ \textbf{if}\ x=0\ \ \textbf{then}\ 1\ \ \textbf{else}\ x\cdot(\lambda z.z!)(x-1) \\ &= \lambda x.\ \textbf{if}\ x=0\ \ \textbf{then}\ 1\ \ \textbf{else}\ x\cdot(x-1)! \\ &= (\lambda x.x!) \end{aligned}$$

- Notice that $\lambda x.x!$ is a *fixed point* of the functional τ
- Meaning of `fact` in **`let recfun`** `fact(n) = B` **`in`** $\cdots$?
  - — Want *evaluate*$[\![$`fact`$]\!]$ = function $f*$ such that
    $f* =$ *evaluate*$[\![$`B`$]\!]$ $(u[$`fact` $\mapsto f*])$
  - — i.e., $f* = (\lambda f.\mathit{evaluate}[\![\texttt{B}]\!]\,(u[\texttt{fact} \mapsto f]))\ f*$
  - — i.e., $f* = \tau\ f*$
    $\therefore$ Want a function that is the fixed point of τ

Recursive Definition (cont.)

- Solution: $f* = \lambda z.z!$
  - Verify:
    $$\begin{aligned}\tau\, f* = \tau\,(\lambda z.z!) &= \lambda x.\ \textbf{if}\ x = 0\ \ \textbf{then}\ 1\ \ \textbf{else}\ x \cdot *(\lambda z.z!)(x - 1)\\ &= \lambda x.\ \textbf{if}\ x = 0\ \ \textbf{then}\ 1\ \ \textbf{else}\ x \cdot *(x - 1)!\\ &= \lambda x.\ \textbf{if}\ x = 0\ \ \textbf{then}\ 1\ \ \textbf{else}\ x!\\ &= \lambda x.\ x!\\ &= f*\end{aligned}$$
  - $\therefore\ f* = \lambda z.z!$ is a fixed point
- Questions Remain:
  - Is $\lambda z.z!$ the *right* fixed point ? (there might be several)
  - What is the connection between this fixed point and the function that is actually *computed* by recursion?

Fixed Points

- *Definition:* Let $\tau : D \rightarrow D$ be a mapping from a domain to itself. $x*$ is a *fixed point of D* $\Leftrightarrow$ $x* = \tau(x*)$

Examples from various domains:

- $D = \mathcal{R}$. To find a root of $x^3 - x^2 - x - 1 = 0$, divide through by x^2 to get
  $x = 1 + (1/x) + (1/x^2) = \tau(x)$
  The positive root $x* = 1.839 \cdots$ is found by iterating: $x_0 = 1$, $x_{n+1} = \tau(x_n)$

- D = Integer .
  - $\tau = \lambda x . x + 1$ has *no* fixed point (except ∞).
  - $\tau = \lambda x . x^2$ has *two* fixed points.
  - $\tau = \lambda x . x$ has infinitely many fixed points — any point in D.

- D = (Integer $\rightarrow$Integer).
  - $\tau = \lambda f . \lambda x . f(x)$ has any function in D as fixed point
  - $\tau = \lambda f . \lambda x .$ **if** $x = 0$ **then** 0 **else** $x + f(x - 1)$ has fixed point $f* = \lambda x . x(x + 1)/2$

— $\tau = \lambda f.\lambda x.x + f(x-1)$ has the fixed points $f_c* = \lambda x.x(x+1)/2 + c$, one for each c in D. Note that $f_\perp* = \lambda x.\perp = \Omega$.

— $\tau = \lambda f.\lambda x.$ **if** $f(x) = 0$ **then** 1 **else** 0 has the fixed point $f* = \lambda x.\perp = \Omega$.

— $\tau = \lambda f.\lambda x.$ **if** $x = 0$ **then** a **else** $f(x)$ has as fixed point $f*$ *any f* such that $f(0) = a$.

- $D = ($Integer $\times$Integer $\rightarrow$Integer $)$.

— Consider the fixed point equation

$$\begin{aligned} f(m,n) &= \tau(f)(m,n) \\ &= \textbf{if } m = 0 \textbf{ then } n \textbf{ else } f(m-1, n+1) \end{aligned}$$

— $g(m,n) = m + n$ is a fixed point. Verification:

$$\begin{aligned} \tau(g)(m,n) &= \textbf{if } m = 0 \textbf{ then } n \textbf{ else } g(m-1, n+1) \\ &= \textbf{if } m = 0 \textbf{ then } n \textbf{ else } (m-1) + (n+1) \\ &= \textbf{if } m = 0 \textbf{ then } n \textbf{ else } m + n \\ &= m + n \\ &= g(m,n) \end{aligned}$$

— Fact: If a fixed point is defined for every element of the source domain, then it is the unique fixed point (McCarthy’s Recursion Induction Principle).

- D = (Integer $\rightarrow$Integer).
  - Let $\tau = \lambda f.\lambda n.f(n+1)$. Now for every integer a, $g_a = \lambda n.a$ is a fixed point.
  - Which one is ''correct''?
  - What do we get by computing the recursion?

    $$f(n) \rightarrow f(n+1) \rightarrow f(n+2) \rightarrow \cdots$$

  - So the fixed point actually computed is $g_\perp = \lambda n.\perp$. This is the *minimal fixed point* of τ in D = (Integer $\rightarrow$Integer), i.e., that fixed point of τ that contains the least amount of information.

Semantics of Recursion

- Principle: The function defined by the recursive definition
  $f = \tau(f)$
  is the fixed point $f*$ of τ that is minimal in information ordering among all fixed points of τ
- Key Properties:
  - — *Uniqueness*: There is only one such minimal $f*$ for τ.
  - — *Existence*: $f*$ always exists: any τ constructible by a syntactic definition in any programming language is monotone and continuous, and hence has such a minimal fixed point.
  - — *Correctness*: For every input n, $f*(n)$ agrees with the value (possibly $\perp$) that is computed by ''unwinding the recursion'' in the usual way:
    $f(n) \rightarrow \tau(f(n)) \rightarrow \tau(\tau(f(n))) \rightarrow$
  - — *Realized by Successive Approximation*. The sequence of functions $f_0 = \Omega;\ f_{n+1} = \tau(f_n)$ forms a monotone chain (nondecreasing sequence) in D $f_0 \sqsubseteq f_1 \sqsubseteq f_2 \sqsubseteq \cdots$ This chain converges to a limit identical to the minimal fixed point: $f* = \cup_i f_i$

Semantics of Recursion (cont.)

- Main result of fixed point semantics: the notion of "function defined by recursion" has a semantic meaning independent of what is obtained by formal computation, but agreeing with it in all respects.

- **Ex:** $\tau = \lambda f.\lambda x.\textbf{if } x = 0 \textbf{ then } 1 \textbf{ else } f(x-2)$

  — Fixed points are
  $g_n = \lambda x.\textbf{if } (x \geq 0) \wedge \mathit{even}(x) \textbf{ then } 1 \textbf{ else } n$

  — Minimal fixed point is
  $g_\perp = \lambda x.\textbf{if } (x \geq 0) \wedge \mathit{even}(x) \textbf{ then } 1 \textbf{ else } \perp$
  because $g_\perp \sqsubseteq g_n$ for all n in D.

  It is the "most partial" of all the fixed points; i.e., contains the bare minimum of information needed to satisfy the equation $f = \tau(f)$.

Semantics of Recursion (cont.)

— Pick values and compute by unwinding the recursion

$f(n) \;=\; \tau(f)(n) \;=\; \textbf{if } n = 0 \textbf{ then } 1 \textbf{ else } f(n-2)$

$f(3) \rightarrow f(1) \rightarrow f(-1) \rightarrow \cdots$ (diverges)
$f(4) \rightarrow f(2) \rightarrow f(0) \rightarrow 1$ (converges)
and in general $f(n)$ diverges for n odd or negative and converges to 1 for n even and non-negative.

— Start with ‘‘zero-information’’ approximation Ω, and form a chain by successive application of τ:

$$
\begin{aligned}
g_0 &= \Omega \\
g_1 &= \tau(g_0) \\
&= \lambda n.\textbf{if } n = 0 \textbf{ then } 1 \textbf{ else } \Omega(n-2) \\
&= \lambda n.\textbf{if } n = 0 \textbf{ then } 1 \textbf{ else } \bot \\
g_2 &= \tau(g_1) \\
&= \lambda n.\textbf{if } n = 0 \textbf{ then } 1 \\
&\quad \textbf{else } (\textbf{if } (n-2) = 0 \textbf{ then } 1 \textbf{ else } \bot) \\
&= \lambda n.\textbf{if } (n = 0) \vee (n = 2) \textbf{ then } 1 \textbf{ else } \bot \\
g_3 &= \tau(g_2) \\
&= \lambda n.\textbf{if } n = 0 \textbf{ then } 1 \\
&\quad \textbf{else } (\textbf{if } (n-2) = 0 \vee (n-2) = 2 \textbf{ then } 1 \\
&\quad \textbf{else } \bot) \\
&= \lambda n.\textbf{if } (n = 0) \vee (n = 2) \vee (n = 4) \textbf{ then } 1 \\
&\quad \textbf{else } \bot
\end{aligned}
$$

$$g_4 \quad = \tau(g_3)$$
$$\ldots$$

It is clear that these functions form a chain, each an extension of its predecessor containing more information (being more defined) than its predecessor. It is also evident that the chain converges to the limit function $g_\perp = \lambda n.\mathbf{if}\ (n \geq 0) \wedge even(n)\ \ \mathbf{then}\ 1\ \ \mathbf{else}\ \perp$.

EXP_2: EXP With Recursive Function Definition

($EXP_2 \triangleq EXP_1$ + recursion + conditional expressions)

- Extend Syntax:

```
Declaration  ::=...
      |     recfun Identifier (Formal-Parameter)
                    = Expression
Expression::=...
      |     Expression = Expression
      |     if Expression then Expression
                    else Expression
```

- Extend Semantics:

New semantic rule for recursive function *definition*:

— constuct a functional *abstraction* τ that
  - binds formal parm to λ-variable x
  - binds function name to λ-variable f
  - evaluates body in definition *env* overlain by these bindings
  - constructs τ from this body by lambda abstraction

— bind *fixed point* of τ to name I

EXP_2 (cont.)

$$
\begin{array}{l}
\mathit{elaborate}[\![\texttt{recfun}\ I\,\texttt{(}\,FP\,\texttt{)} = E]\!]\ \mathit{env} = \\
\qquad \textbf{let}\ \tau = \lambda f\,.\,\lambda x\,.\,\mathit{evaluate}[\![E]\!]\,(\mathit{env}[I \mapsto f, FP \mapsto x]) \\
\qquad \textbf{in} \\
\qquad \textbf{let}\ \mathit{func} = \tau\ \mathit{func} \qquad\qquad \text{— fixed point} \\
\qquad \textbf{in} \\
\qquad \mathit{bind}(I, \mathit{function}\ \mathit{func})
\end{array}
$$

— If I does not occur in E, then this reduces to
$\mathit{func} = \tau\ \mathit{func} = \lambda x\,.\,\mathit{evaluate}[\![E]\!]\,(\mathit{env}[FP \mapsto x])$
which reduces to the rule for ordinary functions:
$\mathit{elaborate}[\![\texttt{fun}\ I\,\texttt{(}\,FP\,\texttt{)} = E]\!]\ \mathit{env} = \cdots$

- Add semantics for **`if`**, relational operators, etc.
- All other semantics (e.g., function calls) stays the same