Semantics: Recursion

Recursion

What is recursion?

J.H. Morris, *Lambda-Calculus Models of Programming Languages*, PhD dissertation, pp. 12, 1968:

> *"We tend to understand these subjects pragmatically. When a programner thinks of recursion, he thinks of push-down stacks and other aspects of how recursion "works". Similarly, types and type declarations are often described as communications to a compiler to aid it in allocating storage, etc.*
> *The thesis of this dissertation, then, is that these aspects of programming languages can be given an intuitively reasonable semantic interpretation."*

Recursion

```
fun fact(n) =
  if n=0 then 1 else n*fact(n-1)

(fun fact(n) =
  if n=0 then 1 else n*fact(n-1))  3 →
```

* Idea and formulation quoted from Robert Harper, It Is What It Is (And Nothing Else), https://existentialtype.wordpress.com/2016/02/22/it-is-what-it-is-and-nothing-else/.

Recursion

```
fun fact(n) =
  if n=0 then 1 else n*fact(n-1)

(fun fact(n) =
  if n=0 then 1 else n*fact(n-1))  3 →

if 3=0 then 1 else
3*( fun fact(n) =
      if n=0 then 1 else n*fact(n-1) )(3-1) →
```

Recursion

```
fun fact(n) =
  if n=0 then 1 else n*fact(n-1)

(fun fact(n) =
  if n=0 then 1 else n*fact(n-1))  3 →

3*( fun fact(n) =
      if n=0 then 1 else n*fact(n-1) )(3-1) →
```

Recursion

```
fun fact(n) =
  if n=0 then 1 else n*fact(n-1)

(fun fact(n) =
  if n=0 then 1 else n*fact(n-1))  3 →

3*( fun fact(n) =
      if n=0 then 1 else n*fact(n-1) )(2) →
```

Recursion

```
fun fact(n) =
  if n=0 then 1 else n*fact(n-1)

(fun fact(n) =
  if n=0 then 1 else n*fact(n-1))  3 →

3*( fun fact(n) =
      if n=0 then 1 else n*fact(n-1) )(2) →

3*(if 2=0 then 1 else
   2*( fun fact(n) =
         if n=0 then 1 else n*fact(n-1) )(2-1)) →
```

Recursion

```
fun fact(n) =
  if n=0 then 1 else n*fact(n-1)

(fun fact(n) =
  if n=0 then 1 else n*fact(n-1))  3 →

3*( fun fact(n) =
      if n=0 then 1 else n*fact(n-1) )(2) →

3*(2*( fun fact(n) =
         if n=0 then 1 else n*fact(n-1) )(2-1)) →
```

Recursion

```
fun fact(n) =
  if n=0 then 1 else n*fact(n-1)

(fun fact(n) =
  if n=0 then 1 else n*fact(n-1))  3 →

3*( fun fact(n) =
      if n=0 then 1 else n*fact(n-1) )(2) →

3*(2*( fun fact(n) =
         if n=0 then 1 else n*fact(n-1) )(1)) →
```

Recursion

```
fun fact(n) =
  if n=0 then 1 else n*fact(n-1)

(fun fact(n) =
  if n=0 then 1 else n*fact(n-1))  3 →

3*( fun fact(n) =
      if n=0 then 1 else n*fact(n-1) )(2) →

3*(2*( fun fact(n) =
         if n=0 then 1 else n*fact(n-1) )(1)) →

3*(2*(if 1=0 then 1 else
      1*( fun fact(n) =
          if n=0 then 1 else n*fact(n-1) )(1-1))) →
```

Recursion

```
fun fact(n) =
  if n=0 then 1 else n*fact(n-1)

(fun fact(n) =
  if n=0 then 1 else n*fact(n-1))  3 →

3*( fun fact(n) =
      if n=0 then 1 else n*fact(n-1) )(2) →

3*(2*( fun fact(n) =
         if n=0 then 1 else n*fact(n-1) )(1)) →

3*(2*(1*( fun fact(n) =
            if n=0 then 1 else n*fact(n-1) )(1-1)))
                                          →  ...
```

Recursion

```
fun fact(n) =
  if n=0 then 1 else n*fact(n-1)

(fun fact(n) =
  if n=0 then 1 else n*fact(n-1))  3 →

3*( fun fact(n) =
      if n=0 then 1 else n*fact(n-1) )(2) →

3*(2*( fun fact(n) =
         if n=0 then 1 else n*fact(n-1) )(1)) →

3*(2*(1*( fun fact(n) =
            if n=0 then 1 else n*fact(n-1) )(1-1)))
                                       →  ...
```

What is recurison?

J.H. Morris, *Lambda-Calculus Models of Programming Languages*, PhD dissertation, pp. 12, 1968:

> *"We tend to understand these subjects pragmatically. When a programner thinks of recursion, he thinks of push-down stacks and other aspects of how recursion "works". Similarly, types and type declarations are often described as communications to a compiler to aid it in allocating storage, etc.*
> *The thesis of this dissertation, then, is that these aspects of programming languages can be given an intuitively reasonable semantic interpretation."*

Recursion

Robert Harper, It Is What It Is (And Nothing Else), https://existentialtype.wordpress.com/2016/02/22/it-is-what-it-is-and-nothing-else/.

```
fun fact(n) =
  if n=0 then 1 else n*fact(n-1)
```

$$(\text{fun}\, f(x) = e)(v) \rightarrow$$
$$e[v/x][\,(\text{fun}\, f(x) = e) \,/\, f\,]$$

Tools

If $e \to^* n$ then $\mathrm{sgn}(n) \in \alpha[\![e, \cdot]\!]$.

$$\mathrm{AVal} = \{+, -, 0\}; \quad \rho : \mathrm{Var} \to \mathcal{P}(\mathrm{AVal})$$

$$\alpha[\![n, \rho]\!] = \{\mathrm{sgn}(n)\}$$

$$\alpha[\![x, \rho]\!] = \rho(x)$$

$$\alpha[\![(\mathrm{let}\, x = e_1 \text{ in } e_2), \rho]\!] = \alpha[\![e_2, (\rho[x := \alpha[\![e_1, \rho]\!]])]\!]$$

$$\alpha[\![(\text{if } e \text{ then } e_1 \text{ else } e_2), \rho]\!] = \alpha[\![e_1, \rho]\!] \cup \alpha[\![e_2, \rho]\!]$$

$$\alpha[\![e_1 \times e_2]\!] =$$

$$\begin{aligned} &(\quad \alpha[\![e_2, \rho]\!] \quad \text{if} \quad + \in \alpha[\![e_1, \rho]\!]) \\ \cup &(-\alpha[\![e_2, \rho]\!] \quad \text{if} \quad - \in \alpha[\![e_1, \rho]\!]) \\ \cup &(\{0\} \quad \text{if} \quad 0 \in \alpha[\![e_1, \rho]\!]) \end{aligned}$$

$$\alpha[\![e_1 + e_2]\!] = \ldots$$

Type Systems

Abstraction

```
module type SWITCH =
sig  val toggle:unit->unit  val get:unit->bool  end

module SwitchBool : SWITCH = struct
  let is_on      = ref false
  let toggle () = (is_on := not (!is_on))
  let get ()     = !is_on
end

module SwitchLog : SWITCH = struct
  let is_on      = ref 0
  let toggle () = (is_on := (!is_on) + 1)
  let get ()     = (!is_on) mod 2
end
```

(Example modified from Andrew M. Pitts, Existential types: Logical relations and operational equivalence, In *ICALP 1998*, pp 309-326.)

Abstraction

```
module SwitchBool =        module SwitchLog =
 let toggle () =            let toggle () =
  is_on := not(!is_on)       is_on := (!is_on) + 1
 let get ()    =            let get ()    =
  !is_on                     (!is_on) mod 2
```

$$\sim \; := \; \{(\text{false}, 2k) \mid k \in \mathbb{N}\} \cup \{(\text{true}, 2k+1) \mid k \in \mathbb{N}\}$$

$$\begin{array}{ccc} (\text{toggle}_{\text{Bool}}(), \cdot[l := \text{false}]) & R^{()} & (\text{toggle}_{\text{Log}}(), \cdot[l := 2k]) \\ \downarrow_* & & \downarrow_* \\ ((), \cdot[l := \text{true}]) & R^{()} & ((), \cdot[l := 2k+1]) \end{array}$$

* This is a completely informal depiction. It should actually be the logical relation instantiated using $\sim$.

Abstraction

```
module SwitchBool =
 let toggle () =
  is_on := not(!is_on)
 let get ()    =
  !is_on
```

```
module SwitchLog =
 let toggle () =
  is_on := (!is_on) + 1
 let get ()    =
  (!is_on) mod 2
```

$$\sim \;:=\; \{(\mathrm{false}, 2k) \mid k \in \mathbb{N}\} \cup \{(\mathrm{true}, 2k+1) \mid k \in \mathbb{N}\}$$

If the initial expressions (using different modules) and the initial stores are related, the final values and the final stores will also be related in a similar manner.

Refinement Types

$$e : \tau$$

$$e : \{x : \tau \mid P\}$$

$$e : \{x : \mathbb{N} \mid 0 \leq x \leq 100\}$$

Type soundness now says if $e \to^* v$ then $0 \leq v \leq 100$.

Check out Liquid Haskell developed by UCSD Programming Systems group.

- https://ucsd-progsys.github.io/liquidhaskell-blog/
- http://goto.ucsd.edu:8090/index.html#?demo=SimpleRefinements.hs

Mac Preview has issues with links containing #s. Please copy the above URL.