

Lecture 21 — Lazy evaluation

- Lazy evaluation, a.k.a. “call-by-name”
 - Evaluation method used in Haskell and lambda-calculus
- Simplifying the basic language
 - Implementing tuples and lists
 - Implement if
 - Implementing integers
 - Implementing recursion

MiniOCaml with lazy evaluation

- In this lecture, we explore the impact of making one small change in the rules for evaluation via substitution:

(Const) $\text{Const } x \Downarrow_{\ell} \text{Const } x$

(Fun) $\text{Fun}(a, e) \Downarrow_{\ell} \text{Fun}(a, e)$

(Rec) $\text{Rec}(f, \text{Fun}(a, e)) \Downarrow_{\ell} \text{Fun}(a, e[\text{Rec}(f, \text{Fun}(a, e))/f])$

(δ) $e \text{ op } e' \Downarrow_{\ell} v \text{ OP } v'$
 $e \Downarrow_{\ell} v$
 $e' \Downarrow_{\ell} v'$

(δ) $\text{op } e \Downarrow_{\ell} \text{OP } v$
 $e \Downarrow_{\ell} v$

(If) $\text{If}(e_1, e_2, e_3) \Downarrow_{\ell} v$
 $e_1 \Downarrow_{\ell} \text{True}$
 $e_2 \Downarrow_{\ell} v$

(If) $\text{If}(e_1, e_2, e_3) \Downarrow_{\ell} v$
 $e_1 \Downarrow_{\ell} \text{False}$
 $e_3 \Downarrow_{\ell} v$

(List) $[e_1, \dots, e_n] \Downarrow_{\ell} [v_1, \dots, v_n]$
 $e_1 \Downarrow_{\ell} v_1$
 $\vdots$
 $e_n \Downarrow_{\ell} v_n$

(App) $e \text{ } e' \Downarrow_{\ell} v$
 $e \Downarrow_{\ell} \text{Fun}(a, e'')$
 $e''[e'/a] \Downarrow_{\ell} v$

let and list

- Since `let` is really just syntactic sugar for function application ($\text{let } x=e \text{ in } e' \equiv (\text{fun } x \rightarrow e') e$), we change the `let` rule accordingly:

$$\begin{array}{c} \text{(Let)} \text{ Let}(a,e,e') \Downarrow_{\ell} v' \\ e'[e/a] \Downarrow_{\ell} v' \end{array}$$

- We will see below that it is also best to change the `List` rule, and, in a couple of cases, the δ rule.
- The set of values stays the same — `consts`, lists of values, and functions — although we will again make a small change later.

Lazy evaluation

- $\Downarrow_\ell$ is called *lazy evaluation* because the evaluation of arguments to functions is delayed until the last moment:

```
(fun x -> x+1) (3+4)  $\Downarrow$  8
  fun x -> x+1  $\Downarrow$  fun x -> x+1
    3+4  $\Downarrow$  7
      3  $\Downarrow$  3
      4  $\Downarrow$  4
    7+1  $\Downarrow$  8
      7  $\Downarrow$  7
      1  $\Downarrow$  1
```

```
(fun x -> x+1) (3+4)  $\Downarrow_\ell$  8
  fun x -> x+1  $\Downarrow_\ell$  fun x -> x+1
    (3+4)+1  $\Downarrow_\ell$  8
      3+4  $\Downarrow_\ell$  7
        3  $\Downarrow_\ell$  3
        4  $\Downarrow_\ell$  4
      1  $\Downarrow_\ell$  1
```

Lazy evaluation (cont.)

- Since all expressions being evaluated are closed, they have the same value regardless of *when* they are evaluated — whether before the function call or within the function call. For this reason, $\Downarrow$ and $\Downarrow_\ell$ almost always produce the same result. But there are exceptions:

`(fun x -> 3) (4/0)  $\Downarrow$`

`(fun x -> 3) (4/0)  $\Downarrow_\ell$`

Lazy evaluation (cont.)

- **Another exception.** This is the abstract syntax version of “let rec f x = f x in (fun x -> 3)(f 0)”:

`let f = Rec(f, fun x -> f x) in (fun x->3)(f 0) ↓`

`let f = Rec(f, fun x -> f x) in (fun x->3)(f 0) ↓ℓ`

Ocaml, Haskell, and lambda calculus

- OCaml uses non-lazy, or “eager,” evaluation ($\Downarrow$)
- Haskell is a popular functional language that uses lazy evaluation. Lazy evaluation provides some extra programming power over eager evaluation (though it also has some drawbacks...).
- λ -calculus is a mathematical system invented to study programming languages, marked by the extreme simplicity of its evaluation rules. It uses only the (lazy versions of) the App and Fun rules, yet is just as powerful (in a theoretical sense) as OCaml.

Ocaml, Haskell, and lambda calculus

- In this lecture, we will show how λ -calculus works, by gradually eliminating all the evaluation rules except App and Fun. All other language features — lists, numbers, if, and recursion — will be shown to be theoretically superfluous (“syntactic sugar”); they can be defined using only Fun. (We have already done this for `let`.)
- Along the way, we will see many examples of higher-order functions.
- (We note that much, but not all, of what we will do could also be done with eager evaluation. Also, some of what we will do only works in a *dynamically-typed* language; that is, we will write some functions that work but would not type-check in OCaml (or Haskell).

Removing features: lists

- **Need to define a representation (type `list = something`) and operations:**

- `nil : list = ...`
- `cons (h:value) (t:list) : list = ...`
- `isempty (l:list) : bool = ...`
- `hd (l:list) : value = ...`
- `tl (l:list) : list = ...`

These must behave like lists, e.g. `hd (cons 3 ..) = 3`.

- **Here is a representation using only functions (where `value` can be any value, including `bool` — we're assuming dynamic type-checking):**

- `type list = (value -> list -> value -> value) -> value`
- `nil : list = fun f -> f 0 0 true`
- `cons (h:value) (t:list) : list = fun f -> f h t false`
- `isempty (l:list) : bool = l (fun h t n -> n)`
- `hd (l:list) : value = l (fun h t n -> h)`
- `tl (l:list) : list = l (fun h t n -> t)`

Removing features: lists (cont.)

`hd (cons 3 nil) ↓ 3`

Lazy lists

- Our definitions of list operations are slightly different from the MiniOCaml ones, in this way: `cons` is non-strict. Since it is a function, and only copies its arguments into another function, it never evaluates them. This is a good thing!
- For example, it allows us to build infinite lists:

```
let rec ints = fun i -> i :: ints (i+1) in hd (tl (tl (ints 0)))
```

- Infinite lists are not just a curiosity. They allow some computations to be written in a more modular way.

Lazy lists (cont.)

- E.g Newton's method:

To find $\text{sqrt}(x)$, generate sequence: $\langle a_i \rangle$, where a_0 is arbitrary, and $a_{i+1} = \frac{a_i + x/a_i}{2}$. Then choose first a_i s.t. $|a_i - a_{i-1}| < \epsilon$.

- This can be programmed elegantly by creating the *infinite* list $\langle a_i \rangle$, and then iterating over it:

```
let next x a = (a+x/a)/2
let rec repeat f a = a :: repeat f (f a)
let candidates x = repeat (next x) (x/2);;
let find test (a1::a2::as) = if test a1 a2 then a2
                             else find test (a2::as)
let withineps eps = fun a b -> (abs (a-b)) < eps
let sqrt x eps = find (withineps eps) candidates
```

If, and boolean values

- With lazy evaluation, can define if. This is impossible in an eager evaluation language like OCaml because all functions defined using fun are strict.

```
let true = fun x y -> x
let false = fun x y -> y
let if a b c = a b c
let lessthan i j = if i < j then true else false
let and b b' = if b then b' else true      (or just, b b' true)
```

```
if (lessthan 3 2) 5 10  $\Downarrow_\ell$  10
```

Where we are...

- We have shown that `let`, `lists`, and `if`, as well as boolean values, are syntactic sugar.
- This eliminates the need for boolean constants and lists, and the δ rules for them.
- Strings can be regarded as (indeed, are) lists of integers, so we will eliminate those as syntactic sugar.
- We are left with integer constants and associated δ rules, plus `Fun`, `App`, and `Rec` rules.

Church numerals

- Alonzo Church invented λ -calculus. On the way to proving that it is Turing-complete, he produced a representation of integers in terms of functions, which has been given the name “Church numerals.”
- As usual, we need a *representation* of integers (type `intgr = something`), and then definitions of constants and functions:
 - `let zero : intgr = ...`
 - `let one : intgr = ...`
 - `let plus (i1:intgr) (i2:intgr) : intgr = ...`
 - `let lessthan (i1:intgr) (i2:intgr) : bool = ...`
 - ... and so on

Church numerals

- These have to act in “integer-like” ways, e.g. `lessthan i (plus i one) must be true`, `equals (mult three four) (mult two six) must be true`, etc.
- Church numerals represent numbers by lambda terms:

```
type intgr = ('a -> 'a) -> ('a -> 'a)
zero  = fun f -> fun x -> x
one   = fun f -> fun x -> f x
two   = fun f -> fun x -> f (f x)
three = fun f -> fun x -> f (f (f x))
```

Church numerals (cont.)

- **Plus and times are easy to define:**

```
let plus m n = fun f -> fun x -> (m f (n f x))  
let mult m n = fun f -> fun x -> (m (n f)) x
```

Removing features: `let rec`

- Implementing recursion without `let rec` is the trickiest part. It uses the so-called “paradoxical combinator,” a.k.a. the “Y combinator”:

```
let W = fun F -> fun f -> F (f f)
let Y = fun F -> (W F)(W F)
```

- For recursive function `let rec f = fun x -> ...`, instead write `let f = Y (fun f -> fun x -> ...)`.

```
let fac = Y (fun fac -> fun x -> if x=0 then 1 else x * fac (x-1))
```

- We leave it as an exercise to show that, e.g., `fac 3` evaluates to 6. Note that there is no explicit recursion used here: each name defined (`W`, `Y`, and `fac`) refer only to names defined previously, never to themselves.