Lecture 27: Lazy evaluation and lambda calculus

- What lazy evaluation is
- Why it's useful
- Implementing lazy evaluation
- Lambda calculus

- Haskell

What is lazy evaluation?

- A slightly different evaluation mechanism for functional programs that provide additional power.
- Used in popular functional language Haskell
- Basic idea: Do not evaluate expressions until it is really necessary to do so.

What is lazy evaluation?

OS_{simp}

- In OS_{subst}, change application rule from:

$$\frac{e_1 \Downarrow \textbf{fun}\ x \text{ -> } e \qquad e_2 \Downarrow v \qquad e[v/x] \Downarrow v'}{e_1 e_2 \Downarrow v'}$$

to:

$$\frac{e_1 \Downarrow \textbf{fun}\ x \text{ -> } e \qquad e[e_2/x] \Downarrow v}{e_1 e_2 \Downarrow v}$$

$(\text{fun } x \to 0)\ (3+4)$

$\Downarrow$

0

$(\text{fun } x \to x+1)\ (3+4)$

$(\text{fun } x \to x+1)\ 7 \to 8$

$(3+4)+1 \to 8$

What difference does it make?

$$((\text{fun } x \to (\text{fun } y \to \text{if } x=0 \text{ then } 0 \text{ else } y+x))\, e_1)\, e_2$$

$$\to (\text{fun } y \to \text{if } e_1 = 0 \text{ then } 0 \text{ else } y + e_1)\, e_2$$

if $e_1 \Downarrow 0$

$\to 0$ - e_2 never eval'd

$$(\text{fun } x \to \text{fun } y \to \text{if } x=0 \text{ then } 0 \text{ else } y)\ a\ (b/a)$$

Usual: (Ocaml, etc)
$$\frac{e_1 \Downarrow v_1 \quad e_2 \Downarrow v_2}{e_1 :: e_2 \Downarrow v_1 :: v_2}$$

Lazy lists

- Laziness principle can apply to cons operation.
- Values = constants | fun x -> e | e1 :: e2

$$\frac{}{e_1 :: e_2 \Downarrow e_1 :: e_2}$$

$$\frac{e \Downarrow e_1 :: e_2 \quad e_1 \Downarrow v}{hd\ e \Downarrow v}$$

$$\frac{e \Downarrow e_1 :: e_2 \quad e2 \Downarrow v}{tl\ e \Downarrow v}$$

(1/0) :: ___

- Could do the same for all data type, i.e. make all constructors lazy.

let rec l = ~~~ :: l ;

Using lazy lists

$0+1 :: ints(0+1+1)$

- Consider this OCaml definition:

```
let rec ints = fun i -> i :: ints (i+1)
let ints0 = ints 0
hd (tl (tl ints0))
```

$\Rightarrow 0 :: ints(1)$

$\Rightarrow 0 :: 1 :: ints(2)$

$\Rightarrow$ - - - -

- What happens in OCaml? What would happen in lazy OCaml?

$ints\ 0 \Downarrow 0 :: ints(0+1)$

$tl\ (0 :: ints(0+1))$

$\Downarrow 1 :: ints(0+1+1)$

“Generate and test” paradigm

- Many computations have the form “generate a list of candidates and choose the first successful one.”
- Using lazy evaluation, can separate candidate generation from selection:
  - Generate list of candidates – even if infinite
  - Search list for successful candidate
- With lazy evaluation, only candidates that are tested are ever generated.

Example: square roots

- Newton-Raphson method: To find sqrt(x), generate sequence: $<a_i>$, where a_0 is arbitrary, and $a_{i+1} = (a_i + x/a_i)/2$. Then choose first a_i s.t. $|a_i - a_{i-1}| < \varepsilon$.
- ```
  let next x a = (a+x/a)/2
  let rec repeat f a = a :: repeat f (f a)
  let rec withineps (a1::a2::as) =
        if abs(a2-a1) < eps then a2
                 else withinips eps (a2::as)
  let sqrt x eps = withineps eps (repeat (next x) (x/2)
  ```
  ```

sameints

- sameints: (int list) list -> (int list) list -> bool
- OCaml:

or $\underbrace{\text{flatten lis1 = flatten lis2}}_{\text{inefficient}}$

```
sameints lis1 lis2 = match (lis1,lis2) with
        ([], []) -> true
      | (_,[]) -> false
      | ([],_) -> false
      | ([]::xs,[]::ys)-> sameints xs ys
      | ([]::xs,ys) -> sameints xs ys
      | (_::xs,[]::ys)-> sameints xs ys
      | (a::as,b::bs) -> (a=b) and sameints as bs;;
```

[[1; 2]; [3]; [4; 5, 6]]   [[1]; [2; 3; 4]; [5]; [6]]

sameints

- Lazy OCaml:

```
flatten lis = match lis with
            [] -> []
          | []::lis' -> flatten lis'
          | (a::as)::lis' -> a :: flatten (as::lis')
equal lis1 lis2 = match (lis1,lis2) with
            ([],[]) -> true
          | (_,[]) -> false
          | ([],_) -> false
          | (a::as, b::bs) -> (a=b) and equal as bs
sameints lis1 lis2 = equal (flatten lis1) (flatten lis2)
```

flatten [[1,2,3],[4],[5]]

→ 1 :: flatten([[2,3],[4],[5]])

flatten [[1,5,6],[4]]

→ 1 :: flatten([[5,6],[4]])

Implementation of lazy eval.

- Use closure model, modified.
- Introduce new value, called a <u>thunk</u>: $\triangleleft e,\eta \triangleright$ - like a closure, but *e* does not have to be an abstraction.

$$\frac{\eta, e_1 \Downarrow \text{<fun } x \text{ -> } e,\eta\text{>} \qquad \eta[x \to \triangleleft e_2,\eta \triangleright],\ e \Downarrow v}{\eta, e_1 e_2 \Downarrow v'}$$

$$\frac{\eta, e \Downarrow v}{\eta', x \Downarrow v} \text{ if } \eta'(x) = \triangleleft e,\eta \triangleright$$

↑ now replace $\triangleleft e,\eta \triangleright$ in η' by v.

Lambda-calculus

- λ-calculus the "original functional language," proposed by Alonzo Church in 1941
- Church wrote "$\lambda x.e$" instead of "fun x->e".
  - Exprs: var's, $\lambda x.e$, $e_1 e_2$
  - Operational semantics:
    - Values: (closed) abstractions
    - Computation rule: Apply principle of **β-equivalence** - $(\lambda x.e)e' \equiv e[e'/x]$ - anywhere; repeat until value is obtained, if ever. (When used in forward direction - $(\lambda x.e)e' \Rightarrow e[e'/x]$ – this is called **β-reduction.**)
- Computation rule corresponds to lazy evaluation, i.e. leads to same results.

Lambda-calculus (cont.)

- In a given expression, there may be many places where β-reduction is applicable. Some choices may never lead to a value, while others do, but:
- <u>Theorem</u> (Church-Rosser) For any expression e, if two sequences of β-reductions lead to a value, then they lead to the same value.
- <u>Theorem</u> Lambda-calculus is a Turing-complete language.

Representing lists in λ-calculus

- We have seen how to represent pairs in OCaml without having them built in:
  - pair a b = λx. λy. λf. f x y
    fst p = p (λx. λy. x)
    snd p = p (λx. λy. y)
- Turns out, you can represent lists (in a similar way), numbers, truth values, etc.
- Using lazy evaluation with this definition of lists corresponds to “lazy cons” shown earlier.