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?

05simp In OS_{subst}, change application rule from:

$$\frac{e_1 \Downarrow \operatorname{fun} x -> e \quad e_2 \Downarrow v \quad e[v/x] \Downarrow v'}{e_1 e_2 \Downarrow v'}$$

to:

to:

$$\underbrace{e_{1} \text{ fun } x \rightarrow e}_{e_{1}e_{2}} \underbrace{v}_{v}$$

$$\underbrace{f_{un} \times \rightarrow O}_{(3+4)} \underbrace{(3+4)}_{(3+4)} \underbrace{f_{un} \times \rightarrow \times +1}_{(3+4)} \underbrace{(3+4)}_{(3+4)} +1$$
What difference does it make?

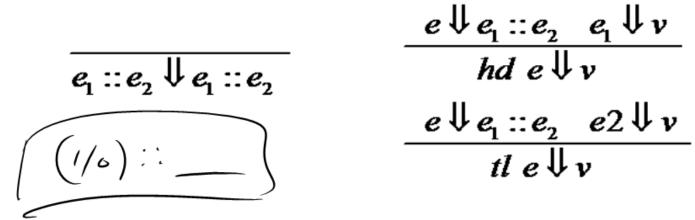
(fun x if x=0 then

(Jun x if for y = y e, = o there o ele y + e,) e =

if e, l o

o - e2 never eval'd ez never eval'd (fun x -> fun y -> y x=0 then b elu y) a (L/a)

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



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

Using lazy lists (0+1+1)

· Consider this OCaml definition:

```
let rec ints = fun i -> i :: ints (i+1)

let ints0 = ints 0 \Rightarrow 0: at_{\bullet}(i)

hd (tl (tl ints0)) \Rightarrow 0:: 1: at_{\bullet}(2)
```

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

happen in lazy OCaml?

$$tl = 0 \text{ intr}(0+1)$$
 $tl = 0 \text{ intr}(0+1)$
 $tl = 0 \text{ intr}(0+1)$

"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 a₀ is arbitrary, and a_{i+1} = (a_i +x/a_i)/2. Then choose first a_i s.t. |a_i-a_{i-1}|<ε.
- 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

oCaml: (int list) list -> (int list) list -> bool

Hatten list = flatten list

```
OCaml:
```

```
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;;
```

sameints flathe ([1,2,37,[47,557)

Lazy OCaml:

```
> 1:: flatter ([[2,37,[Y][[7]]
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
|([],_) -> false
        | (a::as, b::bs) -> (a=b) and equal as bs
sameints lis1 lis2 = equal (flatten lis1) (flatten lis2)
```

Implementation of lazy eval.

- Use closure model, modified.
- Introduce new value, called a thunk:
 <a>de,η▷ like a closure, but e does not have to be an abstraction.

Lambda-calculus

- λ-calculus the "original functional language," proposed by Alonzo Church in 1941
- Church wrote "λx.e" instead of "fun x->e".
 - Exprs: var's, λx.e, e₁e₂
 - Operational semantics:
 - Values: (closed) abstractions
 - Computation rule: Apply principle of β-equivalence (λx.e)e' ≡ e[e'/x] anywhere; repeat until value is
 obtained, if ever. (When used in forward direction (λx.e)e' ⇒ 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:
- Theorem (Church-Rosser) For any expression e, if two sequences of βreductions lead to a value, then they lead to the same value.
- Theorem Lambda-calculus is a Turingcomplete 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.