Lecture 27: Lazy evaluation and lambda calculus

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

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?

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

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

to:

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

What difference does it make?

Lazy lists

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

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

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

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

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

Using lazy lists

- Consider this OCaml definition:

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

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

“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:

```
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

- 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)
```

Implementation of lazy eval.

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

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

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

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, $e_1 e_2$
  - Operational semantics:
    - Values: (closed) abstractions
    - Computation rule: Apply principle of **β-equivalence** - (λx.e)e' $\equiv$ e[e'/x] - anywhere; repeat until value is obtained, if ever. (When used in forward direction - (λx.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:
- Theorem (Church-Rosser) For any expression e, if two sequences of β-reductions lead to a value, then they lead to the same value.
- Theorem 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.