Lecture 26: 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 thunk: $\triangleleft e, \eta \triangleright$ - like a closure, but *e* does not have to be an abstraction.

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

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

Lambda-calculus

- Historically, “fun x->e” was written “λx.e”
- Original “functional language” was proposed by Alonzo Church in 1941:
  - Exprs: var’s, λx.e, e_1e_2
  - Operational semantics:
    - Values: (closed) abstractions
    - Computation rule: Apply β-reductions anywhere in expression; repeat until value is obtained, if ever. (β-reduction means replacing any subexpression of the form (λx.e)e’ by e[e’/x].)
- Computation rule corresponds to lazy evaluation.

Lambda-calculus (cont.)

- In a given expression, there may be many choices of which β-reductions to perform in which order. Some 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.