CS 421 Lecture 25: Lazy evaluation and lambda calculus

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

Announcements

- Practice homework posted
- Final review
  - Moved to Tuesday, August 4
- Final exam information
  - Posted by tomorrow
- Cumulative grades & statistics
  - Posted by Monday/Tuesday

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:

$$\frac{e_1 \Downarrow \text{fun } x \to e \quad e_2 \Downarrow v \quad e[v/x] \Downarrow v'}{e_1 e_2 \Downarrow v'}$$

to:

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

What difference does it make?

```
(fun x y -> if x=0 then x else y) 0 (3/0)
```

Lazy lists

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

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

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

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

- Could do the same for all data types, *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

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

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

$$\frac{\eta, e_1 \Downarrow \langle \text{fun } x \rightarrow e, \eta \rangle \qquad \eta[x \rightarrow \lhd e_2, \eta \rhd], e \Downarrow v}{\eta, e_1 e_2 \Downarrow v}$$

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

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.

Lambda-calculus: the power of h-o functions

- Just need abstraction, application, variables, let
- To show power, we will remove parts of Ocaml:
  - tuples, lists
  - integers
  - if-then-else
  - recursion
- Use β-reduction:

$$(\lambda x.e)\ e' \equiv e[e'/x]$$

and composition:

$$f \circ g = \lambda x. f\,(g\ x)$$

(OCaml defn: `compose f g = fun x -> f (g x)`)

Tuples

```
let pair x y = λf. f x y
let fst p = p (λx. λy. x)
let fst p = p (λx. λy. y)
```

- **Example:** `fst (pair 4 5)`

```
= (λp. p (λx. λy. x)) ((λx. λy. λf. f x y) 4 5)
≡β (λp. p (λx. λy. x)) (λf. f 4 5)
≡β (λf. f 4 5) (λx. λy. x)
≡β (λx. λy. x) 4 5
≡β (λy. 4) 5
≡β 4
```

Lists

```
let nil = λf. f 0 0 true
let cons x y = λf. f x y false
let hd lis = lis (λx. λy. λz. x)
let tl lis = lis (λx. λy. λz. y)
let isnull lis = lis (λx. λy. λz. z)
```

- **Example:** `isnull nil`

  `= (λlis. lis (λx. λy. λz. z)) (λf. f 0 0 true)`

  $\equiv_\beta$ `(λf. f 0 0 true) (λx. λy. λz. z)`

  $\equiv_\beta$ `(λx. λy. λz. z) 0 0 true` $\equiv_\beta$ `true`

- **Example:** `isnull (cons a b)`

  `= isnull (λf. f a b false)`

  $\equiv_\beta$ … $\equiv_\beta$ `(λx. λy. λz. z) a b true` $\equiv_\beta$ `false`

Natural numbers

- Church numerals
- Represent *n* by expression:

$$\lambda f.\lambda x.f\,(f\,(\ldots(fx)\ldots)) = \lambda f.f \circ f \circ \ldots \circ f = \lambda f.f^{\,n}$$

- Example:

```
0 = λf. λx. x
1 = λf. λx. f x ≡η λf. f
2 = λf. λx. f (f x) = λf. f ∘ f
3 = λf. f ∘ f ∘ f
```

Addition and multiplication

- $i + j = \lambda f.\ (i\ f) \circ (j\ f)$

  $1 + 2 = \lambda f.\ (1\ f) \circ (2\ f) = \lambda f.\ (f) \circ (f \circ f)$

  $= \lambda f.\ f \circ f \circ f = 3$

- $i * j = \lambda f.\ i \circ j$

  $2 * 3 = 2 \circ 3 = (\lambda f.\ f \circ f) \circ (\lambda f.\ f \circ f \circ f)$

  $\equiv \lambda g.\ ((\lambda f.\ f \circ f)\ ((\lambda f.\ f \circ f \circ f)\ g))$

  $\equiv \lambda g.\ ((\lambda f.\ f \circ f)\ (\lambda g.\ g \circ g \circ g))$

  $\equiv \lambda g.\ ((g \circ g \circ g) \circ (g \circ g \circ g))$

  $\equiv \lambda g.\ g^6 = 6$

Recursion in lambda-calculus

- Define "paradoxical combinator"

$$\mathrm{Y} = \lambda f.(\lambda x.f\ (x\ x))\,(\lambda x.f\ (x\ x))$$

- For any f:

$$\mathrm{Y}\ f = f\,(\mathrm{Y}\ f)$$ (apply β-reduction twice)

- Consider OCaml definition:

```
let rec sum x = if x = 0 then 0 else x+sum(x-1)
```

then consider this definition:

```
let Sum = Y(λsum. λx. if x=0 then 0 else x+Sum(x-1))
```

- Note that definition of Y is not recursive.

Recursion

```
let Sum = Y(λsum. λx. if x=0 then 0 else x+sum(x-1))
```

- Evaluate `Sum 2:`

```
 (Y s) 2 = s (Y s) 2
= (λx. if x=0 then 0 else x+(Y s)(x-1)) 2
= if 2=0 then 0 else x + (Y s)(2-1)
= 2 + (Y s) 1
= 2 + s (Y s) 1
= 2 + (λx. if x=0 …) 1
= 2 + 1 + (Y s) 0
= 2 + 1 + s (Y s) = … 2 + 1 + 0 = 3
```

- Note: need lazy evaluation!

Lambda-calculus

- Similarly, can get rid of:
  - if-then-else
  - booleans
  - …
- To express *any* sequential functional program, all we need is:
  - Variables
  - Abstraction (λ-expressions)
  - Application (using β-reduction)