Call-by-name
Call-by-value
and Lazy Evaluation

COS 326
David Walker
Princeton University

Last Time

OCaml includes *lazy* computations:

- computations that are executed only when *forced*
- computed only once -- their results are *memoized*

While it is generally a bad idea to use laziness in combination with effects such as printing, printing helps us understand when computations happen:

```
let x = lazy (print_string "hi") in
let y = lazy (print_string "bi") in
Lazy.force y;
Lazy.force x
```

→ "bihi"

```
let x = lazy (print_string "hi") in
let y = lazy (print_string "bi") in
Lazy.force y;
Lazy.force y
```

→ "bi"

Call-by-value Evaluation

Ignoring lazy expressions, OCaml is *call-by-value (CBV)*

Also called *strict* or *eager*.

Left-to-right CBV evaluation of a function application e1 e2:

1) e1 is evaluated to a value v1, which should be a function (fun x -> e)
2) e2 is evaluated to a value v2
3) evaluation continues by substituting v1 for x in the body of the expression e

```
    (fun x -> x + x) (2+3)
--> (fun x -> x + x) 5
--> 5 + 5
--> 10
```

Note that OCaml doesn't specify whether it is left-to-right CBV or right-to-left CBV.

Right-to-left CBV evaluation of a function application:

1) e2 is evaluated to a value v2
2) e1 is evaluated to a value v1, which should be a function (fun x -> e)
3) evaluation continues by substituting v1 for x in the body of the expression e

Call-by-value Evaluation

Notice that the following expression evaluates the same way regardless of whether we use left-to-right or right-to-left CBV

left-to-right CBV:

```
    (fun x -> x + x) (2+3)
--> (fun x -> x + x) 5
--> 5 + 5
--> 10
```

right-to-left CBV:

```
    (fun x -> x + x) (2+3)
--> (fun x -> x + x) 5
--> 5 + 5
--> 10
```

Call-by-value Evaluation

The following expression is evaluated in a slightly different order under left-to-right or right-to-left CBV:

left-to-right CBV:

```
    (fun x -> fun y -> x + y) 2) (3+5)
--> (fun y -> 2 + y) (3+5)
--> (fun y -> 2 + y) 8
--> 2 + 8
--> 10
```

right-to-left CBV:

```
    (fun x -> fun y -> x + y) 2) (3+5)
--> (fun x -> fun y -> x + y) 2) 8
--> (fun y -> 2 + y) 8
--> 2 + 8
--> 10
```

But notice that they compute the same value in the end.

Left-to-right and right-to-left CBV evaluation in pure languages (with effects) always gives the same answer.

Specifying Evaluation Orders

There are many more ways that one might evaluate a functional program! (We saw one: lazy evaluation)

If we want to specify how a language evaluates precisely, we can use an *operational semantics*.

We typically specify operational semantics using inference rules.

Recall:

$$\frac{\text{premiss1} \qquad \text{premiss2} \qquad \ldots \qquad \text{premissn}}{\text{conclusion}}$$

"if premiss1 and premiss2 ... and premiss3 are all valid then the conclusion is valid"

valid means "can be proven by finitely many other inference rules"

λ-calculus

The pure λ-calculus is a language that contains nothing but variables, functions, and function application:

```
x        -- just a variable
λx.e     -- a function with parameter x and body e (i.e., fun x -> e)
e1 e2    -- one expression applied to another (function application)
```

The only lambda calculus *values* are functions (λx.e).

When you see the letter v in what follows, assume I am referring to a value. When you see the letter e, assume I am referring to a general expression.

λ-calculus operational semantics

CBV evaluation rules:

$$\frac{}{(\lambda x.\ e)\ v \ \mapsto\ e[v/x]}$$ (β-reduction)

$$\frac{e1 \ \mapsto\ e1'}{e1\ e2 \ \mapsto\ e1'\ e2}$$

$$\frac{e2 \ \mapsto\ e2'}{e1\ e2 \ \mapsto\ e1\ e2'}$$

Examples:

(λx. x x) (λy.y)
--> (λy.y) (λy.y)

((λx. x x) (λy.y)) ((λx. x x) (λy.y))
--> ((λy.y) (λy.y)) ((λx. x x) (λy.y))

((λx. x x) (λy.y)) ((λx. x x) (λy.y))
--> ((λx. x x) (λy.y)) ((λy.y) (λy.y))

λ-calculus operational semantics

Left-to-right CBV evaluation rules:

$$\frac{}{(\lambda\, x\, .\ e)\ v \ \mapsto\ e[v/x]} \quad (\beta\text{-reduction})$$

$$\frac{e1 \ \mapsto\ e1'}{e1\ e2 \ \mapsto\ e1'\ e2}$$

$$\frac{e2 \ \mapsto\ e2'}{v\ e2 \ \mapsto\ v\ e2'}$$

Examples:

(λx. x x) (λy.y)
--> (λy.y) (λy.y)

((λx. x x) (λy.y)) ((λx. x x) (λy.y))
--> ((λy.y) (λy.y)) ((λx. x x) (λy.y))

Doesn't apply because green is not a value:

((λx. x x) (λy.y)) ((λx. x x) (λy.y))
--> ((λx. x x) (λy.y)) ((λy.y) (λy.y))

λ-calculus operational semantics

Call-by-Name (CBN) evaluation rules:

$$\frac{}{(\lambda x.\ e)\ e2 \ \mapsto\ e[e2/x]} \quad (\beta\text{-reduction})$$

$$\frac{e1 \ \mapsto\ e1'}{e1\ e2 \ \mapsto\ e1'\ e2}$$

~~$$\frac{e2 \ \mapsto\ e2'}{e1\ e2' \ \mapsto\ e1\ e2'}$$~~

Examples:

(λx. x x) ((λy.y) (λy.y))
--> ((λy.y) (λy.y)) ((λy.y) (λy.y))

((λx. x x) (λy.y)) ((λx. x x) (λy.y))
--> ((λy.y) (λy.y)) ((λx. x x) (λy.y))

Don't evaluate expressions until you have to.
Just substitute them in for parameters of functions

Pragmatic CBN Examples

```
    (fun x -> fun y -> x + y + y) 2) (3+5)
--> (fun y -> 2 + y + y) (3+5)
--> 2 + (3+5) + (3+5)
--> 2 + 8 + (3+5)
--> 10 + (3+5)
--> 10 + 8
--> 18
```

I decided to evaluate operators left-to-right

	Printed So Far
`(fun x -> x; x ) (print_string "hi")`	
`--> (print_string "hi"); (print_string "hi")`	
`--> (); print_string "hi"`	hi
`--> print_string "hi"`	hi
`--> ()`	hihi

Non-terminating Computations

Consider the following computation:

$(\lambda x.\ x\ x)\ (\lambda y.y\ y)$

What does it evaluate to using left-to-right CBV evaluation?

$(\lambda y.\ y\ y)\ (\lambda y.y\ y)$

That is the same thing (modulo variable renaming)!

That thing is not a value ... we can keep computing ... forever

We also get the same result if we use right-to-left CBV or CBN!

Do we always get the same answer?

Consider the following computation:

(λx. λy.y) (loop)

where loop is (λy. y y) (λy.y y)

What does it evaluate to using CBV evaluation in 1 step?

(λx. λy.y) (loop)

where loop is (λy. y y) (λy.y y)

What does it evaluate to using CBN evaluation in 1 step?

λy.y

Sometimes call-by-name terminates when call-by-value doesn't!

Is CBN always better than CBV?

Consider the following computation:

```
(λx. x x) (big)

where big is (((λy. y) (λy.y)) (λy. y)) (λy.y)
```

CBV evaluates "big" once.

CBN evaluates "big" twice:

```
(λx. x x) (big)
--> (big) (big)
```

Any time a parameter is used more than once in a function body, CBN is going to repeat evaluation of the argument. Not good!

CBN vs CBV vs Lazy

Sometimes CBN terminates when CBV does not terminate

Sometimes CBN avoids computing an argument when CBV does.

Sometimes CBN computes something 2 (or 3 or 4 ...) times when CBV computes it once.

Laziness:

- can be specified using evaluation rules, but it requires some extra mechanisms, so I won't do it now
- always terminates when CBN terminates
- always avoids computing an argument when CBN does
- computes an argument at most once

Is lazy evaluation the way to go?

Laziness

- Creating a lazy computation is a lot like creating a closure for a function with type unit -> t
- So it takes some work, and it requires some space
  - these constant factors can make a difference
- But a bigger difference is the difficulty reasoning about space:

```
let xs = [1;2;3;4; ... big list ... ] in

let n = lazy (fold (+) 0 xs) in
...
n forced and xs is not used
```

- xs takes up a lot of space
- n is just one integer
- xs can't be collected because n used

exam question:
show how to evaluate expression e step by step using the substitution model with

- left-to-right CBV
- right-to-left CBV, or
- CBN order!

Summary

- CBV is the evaluation strategy used by most languages
  - OCaml, Java, C, ...
- CBN is used by no languages
  - too expensive in practice
    - repeated execution of the same computation
  - but is most likely to terminate and you can write programs that are asymptotically faster than CBV
- Lazy is used by Haskell
  - also language extension in OCaml
  - can also be simulated in other languages using functions with type unit -> t and references