CMSC 330: Organization of Programming Languages

Lazy Evaluation and Streams

Call-by-Value

- In call-by-value (*cbv*), arguments to functions are fully evaluated before the function is invoked
    - Also in OCaml, in let x = e1 in e2, the expression e1 is fully evaluated before e2 is evaluated
- C, C++, and Java also use call-by-value

```
int r = 0;

int add(int x, int y) { return r + x + y; }

int set_r(void) {
  r = 3;
  return 1;
}

add(set_r(), 2);
```

Call-by-Value in Imperative Languages

- In C, C++, and Java, call-by-value has another feature
  - What does this program print?

0

```
void f(int x) {
  x = 3;
}

int main() {
  int x = 0;
  f(x);
  printf("%d\n", x);
}
```

  - Cbv protects function arguments against modifications

Call-by-Name

- Call-by-name (cbn)
  - First described in description of Algol (1960)
  - Generalization of Lambda expressions
  - Idea: In a function:

    Let add x y = x+y
    add (a*b) (c*d)

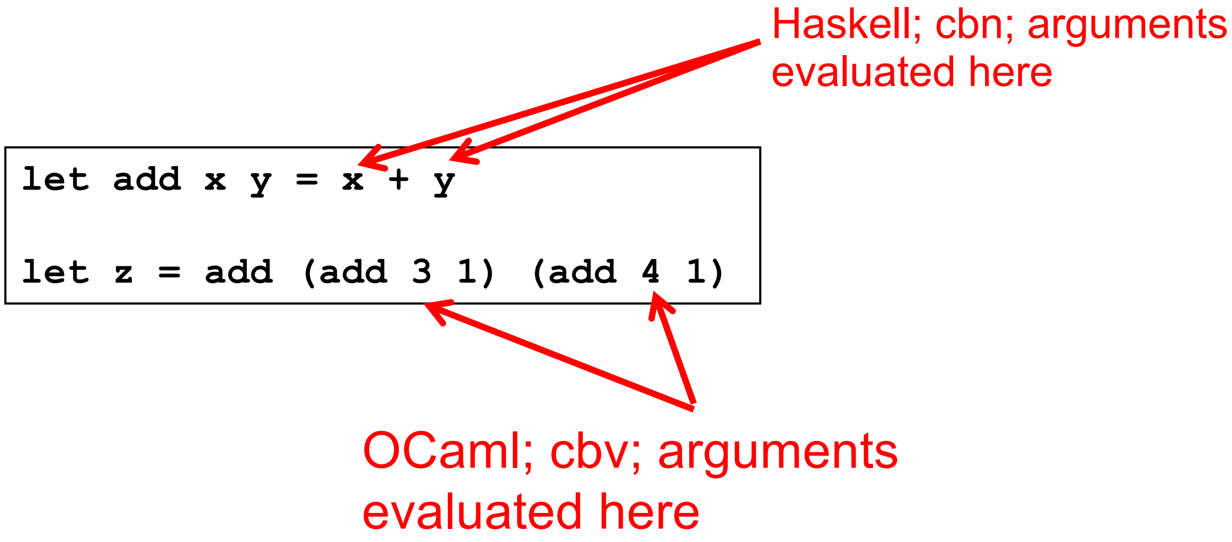
Example:
add (a*b) (c*d) =
(a*b) + (c*d) ← executed function

Then each use of x and y in the function definition is just a literal substitution of the actual arguments, (a*b) and (c*d), respectively

  - Implementation: Highly complex, inefficient, and provides little improvement over other mechanisms

Call-by-Name (cont.)

- In call-by-name (*cbn*), arguments to functions are evaluated at the last possible moment, just before they're needed



Haskell; cbn; arguments evaluated here

```
let add x y = x + y

let z = add (add 3 1) (add 4 1)
```

OCaml; cbv; arguments evaluated here

Call-by-Name (cont.)

- What would be an example where this difference matters?

```
let cond p x y = if p then x else y
let rec loop n = loop n
let z = cond true 42 (loop 0)
```

OCaml; cbv; infinite recursion at call

```
cond p x y = if p then x else y
loop n = loop n
z = cond True 42 (loop 0)
```

Haskell; cbn; never evaluated because parameter is never used

Two Cool Things to Do with CBN

- CBN is also called lazy evaluation
- CBV is also known as eager evaluation

- Build control structures with functions

```
let cond p x y = if p then x else y
```

- Build “infinite” data structures

```
integers n = n::(integers (n+1))
take 10 (integers 0)  (* infinite loop in cbv *)
```

Simulate CBN

Lazy Module: delays computation

```
module Lazy :
  sig
    type 'a t = 'a lazy_t
    val force : 'a t -> 'a
end
```

A value of type 'a Lazy.t is a value of type 'a whose computation has been delayed.

Lazy Module

```
let add x y = x + y;;
val add : int -> int -> int
# let g = lazy (add 10 20);;
val g : int lazy_t = <lazy>
```

Need the value? Force computation

```
# Lazy.force g;;
- : int = 30
```

Using Lazy Module

```
let add x y = x + y;;
let e1 = lazy (add 10 20);;
let e2 = lazy (add 1 2);;
let foo p e1  e2 =
   Lazy.force (if p then e1 else e2);;
```

Type of foo:

```
bool -> 'a Lazy.t -> 'a Lazy.t -> 'a
```

```
# foo true e1 e2;;   (* will not eval e2 *)
- : int = 30
```

Using Lazy Module

```
let rec foo n  = foo n;;
foo 1 (* infinite loop)
let e1 = lazy (foo 1);; (* foo 1 is delayed *)
let e2 = lazy (add 1 2);;

let foo p e1  e2 =
   Lazy.force (if p then e1 else e2);;

# foo false e1 e2;;  (* will not eval e1 *)
- : int = 3
```

Thunk

- Lazy evaluation is implemented using thunks. A thunk is a function of the form

```
fun () -> ....
```

- Body of a function is not evaluated when the function is defined, but only when it is applied. Thus function bodies are evaluated lazily.

```
# List.hd [];; (* eager evaluation *)
Exception: Failure "hd".

let f = fun () -> List.hd [];; (* computation delayed *)
val f : unit -> 'a = <fun> #
f ();;
Exception: Failure "hd".
```

Streams

A stream is an infinite list. Sometimes these are also called sequences, delayed lists, or lazy lists.

```
type stream = Nil | Cons of int * stream lazy_t;;

# let rec ones = Cons(1, lazy ones);;
- val ones : stream = Cons (1, <cycle>)    (* 1,1,1,1,… *)

# let rec from  n  =  Cons(n, lazy (from (n+1)));;
- val from : int -> stream

# let nats = from 0;;
- val nats : stream = Cons (0, <lazy>) (* 0,1,2,3,4,5,… *)
```

Quiz 1

Length of nats

A. 0
B. 10
C. infinite
D. 2

Quiz 1

Length of nats

A. 0
B. 10
C. infinite
D. 2

Quiz 2

To evaluate a lazy expression e, we call

```
A. Lazy.eval e
B. Eval e
C. Force e
D. Lazy.force e
```

Quiz 2

To evaluate a lazy expression e, we call

A. `Lazy.eval e`
B. `Eval e`
C. `Force e`
D. `Lazy.force e`

Streams cont.

```
type stream = Nil | Cons of int * stream lazy_t;;

let hd (s : stream) : int =
  match s with
    Nil -> failwith "hd"
  | Cons (x, _) -> x

let tl (s : stream) :  stream =
  match s with
    Nil -> failwith "tl"
  | Cons (_, g) -> Lazy.force g (* get the tail by evaluating the thunk *)
```

Streams cont.

```
#let rec ones = Cons(1, lazy ones);;
 -val ones : stream = Cons (1, <cycle>)

(* take first n items from the stream *)
let rec take (s : stream) (n : int) : int list =
  if n <= 0 then [] else
  match s with
    Nil -> []
  | _ -> hd s :: take (tl s) (n - 1)

#let t = take nats 10;;
-val t : int list = [0; 1; 2; 3; 4; 5; 6; 7; 8; 9]
```

Streams cont.

```
let rec map (f : int -> int) (s : stream) :  stream =
  match s with Nil -> Nil
  | _ -> Cons (f (hd s), lazy (map f (tl s)))


let rec filter (f : int -> bool) (s : stream) : stream =
  match s with Nil -> Nil
  | Cons (x, g) ->
      if f x then Cons (x, lazy (filter f (Lazy.force g)))
      else filter f (Lazy.force g)
```

Streams: natural numbers

```
let square n = n * n;;

take (map square nats) 10;
-  : int list = [0; 1; 4; 9; 16; 25; 36; 49; 64; 81]

let even = fun n -> n mod 2 = 0;;

take (filter even nats) 10;; (* stream of even numbers *)
- : int list = [0; 2; 4; 6; 8; 10; 12; 14; 16; 18]
```

Streams: Fibonacci

```
let fib1 : stream =
  let rec fibgen (a : int) (b : int) : stream =
    Cons(a, lazy (fibgen b (a + b)))
  in fibgen 1 1

take fib1 10;;
- : int list = [1; 1; 2; 3; 5; 8; 13; 21; 34; 55]

let rec fib2 : stream =
   let add = map2 (+) in
  Cons (1, lazy (Cons (1, lazy (add fib2 (tl fib2)))))

take fib2 10;;
- : int list = [1; 1; 2; 3; 5; 8; 13; 21; 34; 55]
```

Streams: Primes

(* delete multiples of p from a stream *)

```
let sift (p : int) : stream -> stream =
    filter (fun n -> n mod p <> 0)

# take (sift 2 nats) 10;;
- : int list = [1; 3; 5; 7; 9; 11; 13; 15; 17; 19]
```

(* sieve of Eratosthenes *)

```
let rec sieve (s : stream) : stream =
  match s with Nil -> Nil
  | Cons (p, g) -> Cons (p, lazy (sieve (sift p (Lazy.force g ))))
```

(* primes *)

```
let primes = sieve (from 2)

# take primes 20;;
- : int list = [2; 3; 5; 7; 11; 13; 17; 19; 23; 29]
```