Programming Languages and Compilers (CS 421)

Elsa L Gunter

2112 SC, UIUC

http://www.cs.uiuc.edu/class/sp07/cs421/

Based in part on slides by Mattox Beckman, as updated by Vikram Adve and Gul Agha

Infinite Data and Evaluation Methods

- Handling infinite data calls for different evaluation methods
- Three methods discussed below:
  - *Call by Value*
  - *Call by Name*
  - *Call by Need*

Call by Value

- Remember *eager evaluation*?
- Lambda calculus:
  - $(\lambda x.x)((\lambda y.y)z) \rightarrow (\lambda x.x)(z) \rightarrow z$

- This is call by value. It's what you are probably used to.
- OCaml, C/C++, Java, etc…

Call by Value Example

- let f x = (x+1)*(x+1)
- f ((3*4)/6) → ?

Call by Value Example

- let f x = (x+1)*(x+1)
- f ((3*4)/6)
- → f (12/6)

Call by Value Example

- let f x = (x+1)*(x+1)
- f ((3*4)/6)
- → f (12/6)
- → f 2

Call by Value Example

- let f x = (x+1)*(x+1)
- f ((3*4)/6)
- → f (12/6)
- → f 2
- → (2+1)*(2+1)

Call by Value Example

- let f x = (x+1)*(x+1)
- f ((3*4)/6)
- → f (12/6)
- → f 2
- → (2+1)*(2+1)
- → 3*(2+1)

Call by Value Example

- let f x = (x+1)*(x+1)
- f ((3*4)/6)
- → f (12/6)
- → f 2
- → (2+1)*(2+1)
- → 3*(2+1)
- → 3*3

Call by Value Example

- let f x = (x+1)*(x+1)
- f ((3*4)/6)
- → f (12/6)
- → f 2
- → (2+1)*(2+1)
- → 3*(2+1)
- → 3*3
- → 9

Call by Value Example

- let f x = (x+1)*(x+1)
- f ((3*4)/6)
- → f (12/6)
- → f 2
- → (2+1)*(2+1)
- → 3*(2+1)
- → 3*3
- → 9

Argument evaluated b*efore* function call

Only *values* are passed…
Not expressions.

Using Call by Value in Ocaml

[This slide left intentionally blank]*

*Ocaml already uses *Call by Value* semantics, so you don't need to do anything

Call by Value pros/cons

- The Good
  - Efficiency
  - Easy to implement
  - When side-effects are present, easier to understand effects of function call
- The Bad
  - Sometimes wasteful
- The Ugly
  - Can cause otherwise avoidable nontermination

Call by Value nontermination

- let rec foo x = foo (x + 1);;
- let fTrue a b = a;;
- fTrue 5 (foo 10);;

- We've seen this kind of thing before
  - (Eager evaluation in λ-calculus)

Call by Name

- Remember *lazy evaluation*?
- Lambda calculus:
  - $(\lambda x.x)((\lambda y.y)z) \rightarrow (\lambda y.y)z \rightarrow z$
- This is *Call by Name*
- On function call, pass in whole argument, without evaluating

Call by Name Example

- let f x = (x+1)*(x+1)
- f ((3*4)/6) → ?

Call by Name Example

- let f x = (x+1)*(x+1)
- f ((3*4)/6)
- →(((3*4)/6)+1)*(((3*4)/6)+1)

Call by Name Example

- let f x = (x+1)*(x+1)
- f ((3*4)/6)
- →(((3*4)/6)+1)*(((3*4)/6)+1)
- →((12/6)+1)*(((3*4)/6)+1)

Call by Name Example

- let f x = (x+1)*(x+1)
- f ((3*4)/6)
- →(((3*4)/6)+1)*(((3*4)/6)+1)
- →((12/6)+1)*(((3*4)/6)+1)
- →(2+1)*(((3*4)/6)+1)

Call by Name Example

- let f x = (x+1)*(x+1)
- f ((3*4)/6)
- →(((3*4)/6)+1)*(((3*4)/6)+1)
- →((12/6)+1)*(((3*4)/6)+1)
- →(2+1)*(((3*4)/6)+1)
- →(3)*(((3*4)/6)+1)

Call by Name Example

- let f x = (x+1)*(x+1)
- f ((3*4)/6)
- →(((3*4)/6)+1)*(((3*4)/6)+1)
- →((12/6)+1)*(((3*4)/6)+1)
- →(2+1)*(((3*4)/6)+1)
- →(3)*(((3*4)/6)+1)
- →(3)*((12/6)+1)
- →(3)*(2+1) → 3*3 → 9

Using Call by Name in Ocaml

- Can *lambda lift* expressions to delay their evaluation (fun () -> *e*)
- Instead of

```
let f x = (x+1)*(x+1)
f ((3*4)/6)
```

- Write

```
let f x = ((x())+1)*((x())+1)
f (fun () -> ((3*4)/6))
```

Call by Name pros/cons

- The Good
  - If it is possible to terminate, it will.
- The Bad
  - Inefficient if you use argument more than once
  - Doesn't play nice with Side Effects

Side Effects

- Consider this C-like imperative code:

```
int x=5;
int square(int a)
{
    return a*a;
}
```

What does square(++x)return?

Side Effects

- What does square(++x)return?
- With *Call by Value*
  - Returns 6*6 = 36
- With *Call by Name*
  - Returns 6*7 = 42

But…

- The most useful property of *Call by Name* is its termination property.
  - If you have a non-terminating argument that **is not used**, then *Call by Name* will terminate and *Call by Value* won't.
  - Related: if you have an expensive argument that **is not used**, then *Call by Name* can be more efficient than *Call by Value*.

Call by Need

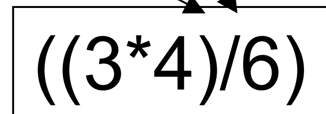
- The “Truly Lazy” form of Evaluation
  - *Call by Name* can end up doing more work than *Call by Value*, but *Call by Need* won’t.
- Postpone argument evaluation until value actually needed, *but* only evaluate once.
  - Have to use side-effects!

Call by Need Example

- let f x = (x+1)*(x+1)
- f ((3*4)/6) → ?

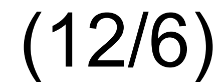
Call by Need Example

- let f x = (x+1)*(x+1)
- f ((3*4)/6)
- → (<thunk> + 1)*(<thunk> + 1)



Call by Need Example

- let f x = (x+1)*(x+1)
- f ((3*4)/6)
- → (<thunk> + 1)*(<thunk> + 1)



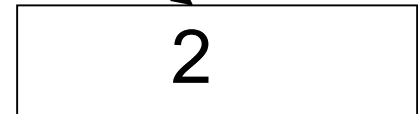
Call by Need Example

- let f x = (x+1)*(x+1)
- f ((3*4)/6)
- → (<thunk> + 1)*(<thunk> + 1)

2

Call by Need Example

- let f x = (x+1)*(x+1)
- f ((3*4)/6)
- → (<thunk> + 1)*(<thunk> + 1)
- →(2 + 1) * (<thunk> + 1)



Call by Need Example

- let f x = (x+1)*(x+1)
- f ((3*4)/6)
- → (<thunk> + 1)*(<thunk> + 1)
- →(2 + 1) * (<thunk> + 1)
- → 3 *(<thunk> + 1)

Call by Need Example

- let f x = (x+1)*(x+1)
- f ((3*4)/6)
- → (<thunk> + 1)*(<thunk> + 1)
- →(2 + 1) * (<thunk> + 1)
- → 3 *(<thunk> + 1)
- → 3 *(2 + 1)

2

Call by Need Example

- let f x = (x+1)*(x+1)
- f ((3*4)/6)
- → (<thunk> + 1)*(<thunk> + 1)
- →(2 + 1) * (<thunk> + 1)
- → 3 *(<thunk> + 1)
- → 3 *(2 + 1)
- → 3 * 3
- → 9

Argument is not evaluated until it is needed, but it is evaluated at most once

2

Side-Effects in Ocaml

- Ocaml has some imperative features

```
# let counter = ref 0;;
val counter : int ref = {contents = 0}
# counter := ! counter + 1;;
# !counter;;
- : int = 1
# counter := !counter + 1;;
# !counter;;
- : int = 2
```

Implementing Thunks

- We need three things:
  - A *thunk* or *suspension* to hold the data
  - A function *delay* to suspend an expression
  - A function *force* to give us the value of the suspended expression

```
# type 'a thunk_type =
  Value of 'a
 | Susp of (unit -> 'a);;
```

Implementing Thunks

```
# let delay f =
    let thunk = ref (Susp f) in
    fun () -> match (!thunk) with
    | Value a -> a
    | Susp f ->
      let result = f () in
      (thunk := (Value result);
       result );;
```

Implementing Thunks

```
# let force f = f ();;
```

- Instead of

```
let f x = (x+1)*(x+1)
f ((3*4)/6)
```

- Write

```
let f x = ((force x)+1)*((force x)+1)
f (delay (fun () -> ((3*4)/6)))
```

The Lazy Module

- Actually, this has been implemented for you already in Ocaml's "Lazy" module
- lazy creates the suspension (thunk)
- Lazy.force gets the value

```
let f x = ((Lazy.force x)+1)
        *((Lazy.force x)+1)
f (lazy ((3*4)/6))
```

Infinite Data

- You saw this already:

```
# let rec ones = 1::ones;;

val ones : int list =
  [1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1;
   1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1;
   1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1;
   1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1;
   1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1;
   1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1;
   1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1;
   1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1;
   1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1;
   1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1;
   1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1;
   1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1;
   ...]
```

Infinite Data

- let rec ones = 1::ones
- But how would you operate on that data?
  - For example, what if you want to do map on ones?

```
# let twos=List.map ((+) 1) ones;;
Stack overflow during evaluation (looping
  recursion?).
```

Lazy Lists

```
type 'a llist =
  Cons of 'a * 'a llist Lazy.t
  | Nil;;

let rec ftake n llst =
  match n,llst with
  | _,Nil -> []
  | 0,_ -> []
  | _,(Cons(x,xs)) -> x :: ftake (n-1) (Lazy.force xs);;
```

Infinite Data the Lazy way

```
# let rec ones = Cons(1, lazy ones);;
val ones : int llist = Cons (1, <lazy>)

# ftake 30 ones;;
- : int list = [1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1;
   1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1]
```

Map on an Infinite List

```
let rec lmap f llst =
  match llst with
  | Cons (x,xs) -> Cons (f x,
    lazy(lmap f (Lazy.force xs)))
  | Nil -> Nil
```

Map on an Infinite List

```
# let twos = lmap ((+) 1) ones;;
val twos : int llist = Cons (2, <lazy>)
# ftake 30 twos;;
- : int list =
[2; 2; 2; 2; 2; 2; 2; 2; 2; 2; 2; 2; 2; 2; 2; 2;
 2; 2; 2; 2; 2; 2; 2; 2; 2; 2; 2; 2; 2; 2]
```

The List of Natural Numbers

let rec nats = ?

Hint: You’re going to use `lmap`

The List of Natural Numbers

```
# let rec nats = Cons(1, lazy (lmap ((+) 1)
   nats));;
val nats : int llist = Cons (1, <lazy>)
# ftake 30 nats;;
- : int list =
[1; 2; 3; 4; 5; 6; 7; 8; 9; 10; 11; 12; 13; 14; 15;
   16; 17; 18; 19; 20; 21; 22; 23; 24; 25; 26; 27;
   28; 29; 30]
```

Question

```
# let plus a b = print_string "Plus "; a+b;;
val plus : int -> int -> int = <fun>
# let rec nats = Cons(1, lazy (lmap (plus 1) nats));;
val nats : int llist = Cons (1, <lazy>)
# ftake 3 nats;;
Plus Plus Plus- : int list = [1; 2; 3]
# ftake 5 nats;;
```

How many times is Plus printed here?

Answer

```
# let plus a b = print_string "Plus "; a+b;;
val plus : int -> int -> int = <fun>
# let rec nats = Cons(1, lazy (lmap (plus 1) nats));;
val nats : int llist = Cons (1, <lazy>)
# ftake 3 nats;;
Plus Plus Plus- : int list = [1; 2; 3]
# ftake 5 nats;;
Plus Plus- : int list = [1; 2; 3; 4; 5]
```