L10: Lazy Evaluation (Pre Lecture)

Dr. Aysu Betin Can

CSCI-400, Colorado School of Mines

Fall 2024

Introduction

Overview

- **Lazy Evaluation**: only evaluate expressions when we absolutely have to
- Function parameters passed *without evaluating them*
- Laziness is sometimes faster (and sometimes slower)
- Laziness lets us represent *infinite* data structures!

Learning Outcomes

- Know definitions of **eager** and **lazy** evaluation.
- Implement lazy evaluation using lambdas and references
- Understand and use **streams**: lazy, infinite-length lists

Streams: Infinite Sequences

Example

π

3 → 1 → 4 → 1 → 5 → 9 → 2 → · · ·

Pseudorandom numbers

42 → 11355432 → 2836018348 → 476557059 → · · ·

Simulations ($\sin \frac{i*\pi}{4}$)

0 → 0.7071 → 1.0 → 0.7071 → 0.0 → · · ·

How can we represent "infinite" data with finite memory?

Streams Representation Idea

Wrong Idea

Pseudocode

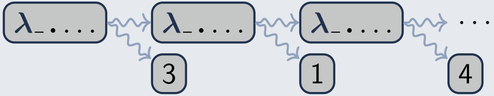
type α stream $\leftarrow$
| Stream **of** $\alpha \times \alpha$ stream

- No base case in the ADT
- Needs infinite memory!

Right Idea

Lazy Cons Cells

Evaluation



do not evaluate rest (tail) until needed

Outline

Call-by-Value

Call-by-Value

- **Call-by-Value**: Eagerly evaluate parameters before calling function
- Used in C, Java, Python, OCaml
- **Function Call w Call-by-Value**: Prints "`apple`" and "`banana`"

Example (Two functions)

var $a \leftarrow (\lambda x\,.\,\texttt{print}\ \text{"}\texttt{apple}\text{"};\ 1)\,;$
var $b \leftarrow (\lambda x\,.\,\texttt{print}\ \text{"}\texttt{banana}\text{"};\ 2)\,;$

Example (Function Call)

$(\lambda x\,.\,\underline{\hspace{6em}})\ (a())\ (b())$

Example: Calls vs. Conditionals

Example (Two functions)

var $a \leftarrow (\boldsymbol{\lambda} x\,.\,\texttt{print "apple"};\ 1)$;
var $b \leftarrow (\boldsymbol{\lambda} x\,.\,\texttt{print "banana"};\ 2)$;

Example (Conditional)

if true then $a()$ **else** $b()$

Example (Function Call)

$(\boldsymbol{\lambda} x\,.\,\boldsymbol{\lambda} y\,.\,\textbf{if true then } x \textbf{ else } y)\ (a())\ (b())$

Behavior

- **Conditional**: Prints "apple".
- **Function Call**:
  - *Eager*: Prints "apple" and "banana"
  - *Lazy*: Prints "apple"
- Conditionals are *lazy*
- Function calls, depends on the language

MINES

Conditionals are "Lazy"

If Expression

$$\textbf{if}\ \underbrace{e_1}_{\text{test}}\ \textbf{then}\ \underbrace{e_2}_{\text{then clause}}\ \textbf{else}\ \underbrace{e_3}_{\text{else clause}}$$

Evaluation

- When $e_1 \rightsquigarrow \textbf{true}$ and $e_2 \rightsquigarrow v_2$, $\textbf{if}\ e_1\ \textbf{then}\ e_2\ \textbf{else}\ e_3 \rightsquigarrow v_2$ (and don't evaluate e_3)
- When $e_1 \rightsquigarrow \textbf{false}$ and $e_3 \rightsquigarrow v_3$, $\textbf{if}\ e_1\ \textbf{then}\ e_2\ \textbf{else}\ e_3 \rightsquigarrow v_3$ (and don't evaluate e_2)
- See also: Boolean "short-circuiting"

Omit evaluation of the unused clause

Eager vs. Lazy Evaluation

Definition (Eager Evaluation)

Under **Eager Evaluation** (also called **Strict Evaluation**), arguments to functions are evaluated <u>before</u> the body of the function.

Definition (Lazy Evaluation)

Under **Lazy Evaluation**, arguments to functions are evaluated <u>on-demand</u>.

- Arguments are passed unevaluated to functions and only evaluated when (and if!) needed.
- Evaluated results are cached to avoid re-evaluations.

Outline

Overview of Simulating Laziness

1. Delay the computation (using Lambda)
2. Cache the results (using References)

Example: Calls vs. Conditionals

Example (Two functions)

$\textbf{var}\ a \leftarrow (\boldsymbol{\lambda} x\,.\,\texttt{print}\ \texttt{"apple"};\ 1);$
$\textbf{var}\ b \leftarrow (\boldsymbol{\lambda} x\,.\,\texttt{print}\ \texttt{"banana"};\ 2);$

Example (Eager Arguments)

$(\boldsymbol{\lambda} x\,.\,\boldsymbol{\lambda} y\,.\,\textbf{if true then}\ x\ \textbf{else}\ y)\ \ (a())\ (b())$

Example (Delayed Arguments: Special Case)

$(\boldsymbol{\lambda} x\,.\,\boldsymbol{\lambda} y\,.\,\textbf{if true then}\ x()\ \textbf{else}\ y())\ \ a\ b$

Example (Delayed Arguments: General)

$(\boldsymbol{\lambda} x\,.\,\boldsymbol{\lambda} y\,.\,\textbf{if true then}\ x()\ \textbf{else}\ y())\ \ (\boldsymbol{\lambda} z\,.\,a())\ (\boldsymbol{\lambda} z\,.\,b())$

Using Lambda to Delay Evaluation

Thunks

Definition (Thunk)

A function, passed as an argument, used to defer evaluation of its body expression.

Thunk Creation

$$\boxed{e} \rightsquigarrow \boxed{(\lambda\_ . e)}$$

Example (Thunks)

- $1 + 2 \quad \rightsquigarrow \quad (\lambda\_ . 1 + 2)$
- **if** a **then** $b + c$ **else** d
  $\rightsquigarrow \quad (\lambda\_ . \textbf{if } a \textbf{ then } b + c \textbf{ else } d)$

Delaying Parameter Evaluation

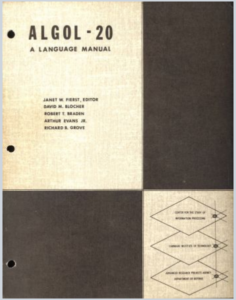
$$\boxed{(\lambda x . (\ldots x \ldots)) \; e}$$
$$\rightsquigarrow$$
$$\boxed{(\lambda x . (\ldots (x\ ()) \ldots)) \; (\lambda\_ . e)}$$

Description

- Given function $\lambda x . \alpha$, and parameter e
- Replace e with function definition $\lambda\_ . e$
- Replace x in α with function call $x\ ()$

Historical Interlude

ALGOL



- Family of languages
- Late 1950s – early 1970s
- Influenced many later languages

Thunks in ALGOL

- ALGOL-60 passed parameters as thunks
- Etymology: irregular past tense of "think": compiler analyzed parameter expression
- But thunks mean future computation
- Better called "will-think?"

Example 0: Delaying Evaluation

Example (Create the thunk)

$$(\lambda a \,.\, a + 1)\ 2$$

$$\Big\downarrow\ 2 \rightsquigarrow \lambda \_ \,.\, 2 \quad a \rightsquigarrow a\ ()$$

$$(\lambda a \,.\, a\ () + 1)\ \lambda \_ \,.\, 2$$

Example (Evaluation)

$$(\lambda a \,.\, a\ () + 1)\ \lambda \_ \,.\, 2$$

$$\Big\downarrow\ a \leftarrow \lambda \_ \,.\, 2$$

$$(\lambda \_ \,.\, 2)\ () + 1$$

$$\Big\downarrow\ \_ \leftarrow ()$$

$$2 + 1$$

$$\Big\downarrow\ +$$

$$3$$

Example 1: Delaying Evaluation

Example (Create the thunk)

$$\boxed{(\lambda b\,.\,\neg b)\ (\neg\mathbf{true})}$$

$$\Big\downarrow\ \neg\mathbf{true} \rightsquigarrow \lambda\_\,.\,\neg\mathbf{true}$$

$$\boxed{(\lambda b\,.\,\neg b\ ())\ \lambda\_\,.\,\neg\mathbf{true}}$$

Example (Evaluation)

$$\boxed{(\lambda b\,.\,\neg b\ ())\ \lambda\_\,.\,\neg\mathbf{true}}$$

$$\Big\downarrow\ b \leftarrow \lambda\_\,.\,\neg\mathbf{true}$$

$$\boxed{\neg(\lambda\_\,.\,\neg\mathbf{true})\ ()}$$

$$\Big\downarrow\ \_ \leftarrow ()$$

$$\boxed{\neg\neg\mathbf{true}}$$

$$\Big\downarrow\ \neg$$

$$\boxed{\neg\mathbf{false}}$$

$$\Big\downarrow\ \neg$$

$$\boxed{\mathbf{true}}$$

Exercise 1: Delaying Evaluation

1. $(\lambda a\,.\,a + 1)\ (2 + 3)$
2. $(\lambda a\,.\,\lambda b\,.\,\lambda c\,.\,a + \textbf{if}\ \neg b\ \textbf{then}\ c\ \textbf{else}\ 0)\ (1 + 2)\ (\neg\textbf{false})\ (3 + 4)$
3. $(\lambda a\,.\,a + a)\ (2 + 3)$

Exercise 1.a: Delaying Evaluation

Create the thunk

$(\lambda a \,.\, a + 1)\ (2 + 3)$

Evaluation

Exercise 1.b: Delaying Evaluation

Create the thunk

$(\lambda a\,.\,\lambda b\,.\,\lambda c\,.\,a + \textbf{if}\ \neg b\ \textbf{then}\ c\ \textbf{else}\ 0)\ \ (1+2)\ (\neg\textbf{false})\ (3+4)$

Evaluation

Exercise 1.e: Delaying Evaluation

Create the thunk

$(\lambda a \,.\, a + a)\ (2 + 3)$

Evaluation

Memoization

Definition (Memoization)

Caching the results of *pure* (side-effect free) function calls and returning the cached result when the function is called again with the same inputs.

Uses of Memoization

- General optimization technique
- Basic principle of dynamic programming
- Implementing lazy evaluation

Suspensions

Pseudocode

type α thunk $\leftarrow$ unit $\mapsto \alpha$

type α suspval $\leftarrow$
| Unevaluated **of** α thunk
| Evaluated **of** α

type α suspension $\leftarrow$ Ref α suspval

Description

- Tag:
  - thunk with Unevaluated
  - value with Evaluated
- Use/update a reference to:
  - Evaluate the thunk once
  - Replace with result for later reuse

Creating and Using Suspensions

Delay: create a suspension

Function delay(thunk)

    ref (Unevaluated thunk)

Force: use a suspension

Function force(susp)

match `deref` susp **with**
  case Evaluated $y \to y$
  case Unevaluated thunk $\to$
    let $y \leftarrow$ thunk () **in**
      susp $\Leftarrow$ Evaluated y;
      y

Example 0: Suspensions

Create and Evaluate

Example (Create the suspension)

$$(\lambda a\,.\,a+1)\ 2$$

$$\rightsquigarrow$$

$$(\lambda a\,.\,\texttt{force}\ a+1)\ (\texttt{delay}\ \lambda\_\,.\,2)$$

Example (Evaluation)

$$(\lambda a\,.\,\texttt{force}\ a+1)\ (\texttt{delay}\ \lambda\_\,.\,2)$$

$$\rightsquigarrow$$

$$(\lambda a\,.\,\texttt{force}\ a+1)\ (\texttt{ref}\ \bullet) \qquad \text{Unevaluated } \lambda\_\,.\,2$$

$$\rightsquigarrow$$

$$\texttt{force}\ (\texttt{ref}\ \bullet)+1 \qquad \text{Unevaluated } \lambda\_\,.\,2$$

$$\rightsquigarrow$$

$$2+1 \qquad \text{Evaluated } 2$$

$$\rightsquigarrow$$

$$3$$

Example 1: Suspensions

Create

Example (Create the suspension)

$$\boxed{(\lambda a\, b\,.\, a + \textbf{if}\ \neg b\ \textbf{then}\ a\ \textbf{else}\ 0)\ 2\ \textbf{false}}$$

$$\rightsquigarrow$$

$$\boxed{(\lambda a\, b\,.\, \texttt{force}\ a + \textbf{if}\ \neg\texttt{force}\ b\ \textbf{then}\ \texttt{force}\ a\ \textbf{else}\ 0)\ (\texttt{delay}\ \lambda\_.\,2)\ (\texttt{delay}\ \lambda\_.\,\textbf{false})}$$

Example 1: Suspensions

Evaluate 1/3

Example (Evaluation)

$$(\lambda a\, b\, .\, \texttt{force}\ a + \textbf{if}\ \neg\texttt{force}\ b\ \textbf{then}\ \texttt{force}\ a\ \textbf{else}\ 0)\ \ (\texttt{delay}\ \lambda\_.\,2)\ (\texttt{delay}\ \lambda\_.\,\textbf{false})$$

$$(\lambda a\, b\, .\, \texttt{force}\ a + \textbf{if}\ \neg\texttt{force}\ b\ \textbf{then}\ \texttt{force}\ a\ \textbf{else}\ 0)\ \ (\texttt{ref}\ \bullet)\ (\texttt{ref}\ \bullet)$$

Unevaluated $\lambda\_.\,2$

Unevaluated $\lambda\_.\,\textbf{false}$

$$\texttt{force}\ (\texttt{ref}\ \bullet) + \textbf{if}\ \neg\texttt{force}\ (\texttt{ref}\ \bullet)\ \textbf{then}\ \texttt{force}\ (\texttt{ref}\ \bullet)\ \textbf{else}\ 0$$

Unevaluated $\lambda\_.\,2$

Unevaluated $\lambda\_.\,\textbf{false}$

Both lazy a's share a ref cell

Example 1: Suspensions

Evaluate 2/3

Example (Evaluation)

$\texttt{force (ref} \bullet\texttt{)} + \textbf{if } \neg\texttt{force (ref} \bullet\texttt{)} \textbf{ then } \texttt{force (ref} \bullet\texttt{)} \textbf{ else } 0$

Unevaluated $\lambda\_ .\, 2$

Unevaluated $\lambda\_ .\, \textbf{false}$

$\rightsquigarrow$

$2 + \textbf{if } \neg\texttt{force (ref} \bullet\texttt{)} \textbf{ then } \texttt{force (ref} \bullet\texttt{)} \textbf{ else } 0$

Evaluated(2)

Unevaluated $\lambda\_ .\, \textbf{false}$

$\rightsquigarrow$

$2 + \textbf{if } \neg\textbf{false} \textbf{ then } \texttt{force (ref} \bullet\texttt{)} \textbf{ else } 0$

Evaluated 2

Evaluated **false**

Example 1: Suspensions

Evaluate 3/3

Example (Evaluation)

$2 + \textbf{if } \neg\textbf{false then } \texttt{force (ref } \bullet\texttt{)} \textbf{ else } 0$ Evaluated 2

$\leadsto$

$2 + \textbf{if true then } \texttt{force (ref } \bullet\texttt{)} \textbf{ else } 0$ Evaluated 2

$\leadsto$

$2 + \texttt{force (ref } \bullet\texttt{)}$ Evaluated 2

$\leadsto$

$2 + 2 \leadsto 4$

Exercise 2: Suspensions

1. $(\boldsymbol{\lambda} a \,.\, a + 1)\ (2 + 3)$
2. $(\boldsymbol{\lambda} a \,.\, a + a)\ (2 + 3)$

Exercise 2.a: Suspensions

Create

Create the suspension

$$\boxed{(\boldsymbol{\lambda} a \,.\, a + 1)\ (2 + 3)}$$

Exercise 2.a: Suspensions

Evaluate

Evaluation

Exercise 2.b: Suspensions

Create

Create the suspension

$$\boxed{(\lambda a \,.\, a + a)\ (2 + 3)}$$

Exercise 2.b: Suspensions

Evaluate

Evaluation

Calling Conventions

Definition (Call-by-Value)

Evaluate arguments before calling the function, and pass the resulting argument value to the function.

Used in: C, OCaml

Definition (Call-by-Name)

Pass arguments to functions as a thunks, which are evaluated when (and each time) the argument is used.

Used in: ALGOL 60

Definition (Call-by-Reference)

Evaluate arguments before calling the function, and pass a reference to the argument result to the function.

Used in: Fortran

Definition (Call-by-Need/Lazy Evaluation)

Pass arguments to functions as a thunks, which are evaluated once when the argument is used and memoized for later reuse.

Used in: Haskell

Applicative vs Normal Order Evalution

Evaluation order concepts in Lambda calculus.

Applicative Order

- Function arguments are evaluated before the function is applied
- **Call-by-Value**
- All arguments are reduced to their simplest form before the function is invoked

Example

- $(\lambda x.\ x^2(\lambda x.\ (x+1)2)))$
  $\rightsquigarrow (\lambda x.x^2(2+1))$
  $\rightsquigarrow (\lambda x.x^2(3)) \rightsquigarrow (3)^2 \rightsquigarrow 9$
- $(\lambda x.\lambda y.\ y)((\lambda z.z\ z)(\lambda z.z\ z))$
  $\rightsquigarrow (\lambda x.\lambda y.\ y)((\lambda z.z\ z)(\lambda z.z\ z))$
  $\rightsquigarrow \ldots$

Applicative vs Normal Order Evalution

Evaluation order concepts in Lambda calculus.

Normal Order

- Evaluates the leftmost/outermost expression first,
- ...without evaluating its arguments until they are needed
- Ensures that the function arguments are only evaluated when required
- **Call-by-Name**

Example

- $(\lambda x.\ x^2(\lambda x.\ (x+1)\ 2)))$
  $\leadsto (\lambda x.\ (x+1)\ 2)^2$
  $\leadsto (2+1)^2 \leadsto 9$
- First reach to the *normal form* by reducing the outermost expression first
- Fully expands the expression and then reduces it
- $(\lambda x.\lambda y.\ y)((\lambda z.z\ z)(\lambda z.z\ z))$
  $\leadsto \lambda y.\ y)$

Outline

Streams Data Type

Description

- Need two mutually recursive types:
  1. Cells
  2. Streams
- A stream is a lazily evaluated cell
- A Stream cell holds:
  1. a single value
  2. the rest of the stream (lazily)

Pseudocode

type α stream-cell $\leftarrow$
  | StreamCons **of** $\alpha \times \alpha$ stream
and α stream $\leftarrow$
  | (α stream-cell) suspension

Evaluating Streams

Head

Function stream-head(σ)

match `force` σ **with**
- **case** StreamCons(h, $\_$) $\rightarrow$
  - h

Tail

Function stream-tail(σ)

match `force` σ **with**
- **case** StreamCons($\_$, τ) $\rightarrow$
  - τ

MINES

Example: Streams of infinite zeros

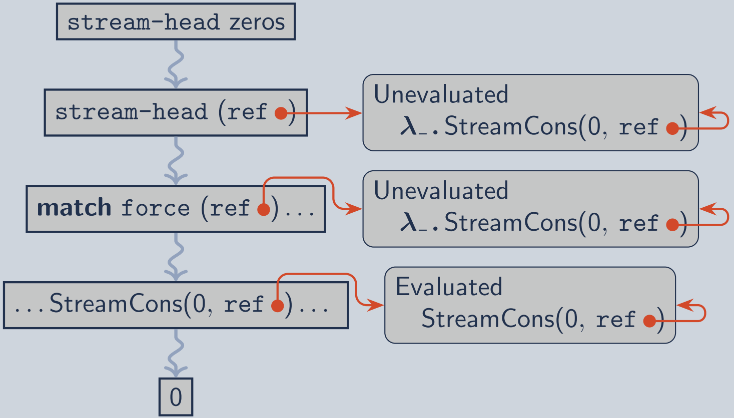
Pseudocode

var rec zeros $\leftarrow$
  delay zeros-thunk
and zeros-thunk $\leftarrow$
  λ_ . StreamCons(0, zeros)

Head

defun stream-head(σ) $\rightarrow$
  match force σ **with**
    case StreamCons(h, _) $\rightarrow$
      h

Example (Evaluation)



MINES

Higher Order Functions on Streams

Lazy `map`

Description

- Apply of a function over a stream
- Create a new lazy value that:
    1. Applies function to the stream head
    2. Recursively maps over stream tail

Pseudocode

Function stream-map(σ, f)

```
flet thunk _ →
    let
        h ← stream-head σ
        τ ← stream-tail σ
    in
        let ω ← stream-map τ f
        in StreamCons(f h, ω)
in delay thunk
```

Summary

Lazy Evaluation

- **Lazy Evaluation:** Delay evaluation until needed and memoize results
- In **eager** languages, we can simulate laziness using first-order functions and references

Eager Evaluation Benefits

- Sometimes faster: no overhead to delay/memoize evaluation
- Analysis: easier for (worst-case) running times

Lazy Evaluation Benefits

- Sometimes faster: saves computation of unused arguments
- Expressivity: infinite data structures
- Analysis: amortized running times