Lazy Evaluation

CS 350

Dr. Joseph Eremondi

Last updated: August 12, 2024

Lazy Evaluation

Overview

- Objectives

Overview

- Objectives
  - To understand the design space of evaluation order

Overview

- Objectives
  - To understand the design space of evaluation order
- Key Concepts

Overview

- Objectives
  - To understand the design space of evaluation order
- Key Concepts
  - Lazy evaluation

Overview

- Objectives
  - To understand the design space of evaluation order
- Key Concepts
  - Lazy evaluation
  - Strict evaluation

Overview

- Objectives
  - To understand the design space of evaluation order
- Key Concepts
  - Lazy evaluation
  - Strict evaluation
  - Strictness points

Overview

- Objectives
  - To understand the design space of evaluation order
- Key Concepts
  - Lazy evaluation
  - Strict evaluation
  - Strictness points
  - Thunks

The Type of Substitution

- We could replace a variable with *any expression* in subsituttion

The Type of Substitution

- We could replace a variable with *any expression* in subsituttion
  - Doesn’t have to be a value

The Type of Substitution

- We could replace a variable with *any expression* in subsituttion
  - Doesn’t have to be a value

The Type of Substitution

- We could replace a variable with *any expression* in subsituttion
  - Doesn't have to be a value

```
(define (subst [toReplace : Symbol]
               [replacedBy : Expr]
               [replaceIn : Expr]) : Expr ....)
```

Substituting Expressions vs. Values

- Think *waaaaayyyy* back to our substitution-based Curly-Lambda interpreter

Substituting Expressions vs. Values

- Think *waaaaayyyy* back to our substitution-based Curly-Lambda interpreter

Substituting Expressions vs. Values

- Think *waaaaayyyy* back to our substitution-based Curly-Lambda interpreter

```
(define (interp expr)
  (type-case Expr interp
  ....
    [(Call funExpr argExpr)
      (let* ([argVal (interp argExpr)]
             [funVal (checkAndGetFun (interp funExpr))]
             [funParam (fst funVal)]
             [funBody (snd funVal)]))
        (interp (subst funParam
                       (value->expr argVal )
                       funBody))] ))
```

- We evaluate the argument before substituting it into the function body

Substituting Expressions vs. Values

- Think *waaaaayyyy* back to our substitution-based Curly-Lambda interpreter

```
(define (interp expr)
  (type-case Expr interp
  ....
    [(Call funExpr argExpr)
      (let* ([argVal (interp argExpr)]
             [funVal (checkAndGetFun (interp funExpr))]
             [funParam (fst funVal)]
             [funBody (snd funVal)]))
        (interp (subst funParam
                       (value->expr argVal )
                       funBody))] ))
```

- We evaluate the argument before substituting it into the function body
  - Why do we have to do this?

Substituting Expressions vs. Values

- Think *waaaaayyyy* back to our substitution-based Curly-Lambda interpreter

```
(define (interp expr)
  (type-case Expr interp
  ....
    [(Call funExpr argExpr)
      (let* ([argVal (interp argExpr)]
             [funVal (checkAndGetFun (interp funExpr))]
             [funParam (fst funVal)]
             [funBody (snd funVal)]))
        (interp (subst funParam
                       (value->expr argVal )
                       funBody))] ))
```

- We evaluate the argument before substituting it into the function body
  - Why do we have to do this?
  - We don't

Substitution-Based Lazy Evaluation

Substitution-Based Lazy Evaluation

```
(define (interp expr)
  (type-case Expr interp
  ....
    [(Call funExpr argExpr)
      (let* (;; Interpret the function to a value
             [funVal (checkAndGetFun (interp funExpr))]
             [funParam (fst funVal)]
             [funBody (snd funVal)]))
        ;; Substitute the parameter value into the function body
        (interp (subst funParam
                       argExpr ;; !! <-----------------------
                       funBody))] ))
```

- Don’t interpret argument to a value before substituting it

Substitution-Based Lazy Evaluation

```
(define (interp expr)
  (type-case Expr interp
  ....
    [(Call funExpr argExpr)
      (let* (;; Interpret the function to a value
             [funVal (checkAndGetFun (interp funExpr))]
             [funParam (fst funVal)]
             [funBody (snd funVal)]))
        ;; Substitute the parameter value into the function body
        (interp (subst funParam
                       argExpr ;; !! <-----------------------
                       funBody))] ))
```

- Don’t interpret argument to a value before substituting it
- Works just as well, *but we get different behaviour*

Strict vs. Lazy Semantics

- **Strict evaluation order** is when we evaluate function arguments to a value before substituting them into the function body

Strict vs. Lazy Semantics

- **Strict evaluation order** is when we evaluate function arguments to a value before substituting them into the function body
  - Also called **eager** semantics

Strict vs. Lazy Semantics

- **Strict evaluation order** is when we evaluate function arguments to a value before substituting them into the function body
  - Also called **eager** semantics
- **Lazy evaluation order** is when we don't evaluate function parameters before substituting them into function bodies

Strict vs. Lazy Semantics

- **Strict evaluation order** is when we evaluate function arguments to a value before substituting them into the function body
  - Also called **eager** semantics
- **Lazy evaluation order** is when we don't evaluate function parameters before substituting them into function bodies
- As long as a language has no mutable state, a program that runs successfully will produce *the exact same results* with strict and lazy semantics

Strict vs. Lazy Semantics

- **Strict evaluation order** is when we evaluate function arguments to a value before substituting them into the function body
  - Also called **eager** semantics
- **Lazy evaluation order** is when we don't evaluate function parameters before substituting them into function bodies
- As long as a language has no mutable state, a program that runs successfully will produce *the exact same results* with strict and lazy semantics
  - However, some programs might fail with strict but succeed with lazy

Strict vs. Lazy Semantics

- **Strict evaluation order** is when we evaluate function arguments to a value before substituting them into the function body
  - Also called **eager** semantics
- **Lazy evaluation order** is when we don't evaluate function parameters before substituting them into function bodies
- As long as a language has no mutable state, a program that runs successfully will produce *the exact same results* with strict and lazy semantics
  - However, some programs might fail with strict but succeed with lazy
  - Some programs might run forever with strict, but terminate with lazy

Strict vs. Lazy Semantics

- **Strict evaluation order** is when we evaluate function arguments to a value before substituting them into the function body
  - Also called **eager** semantics
- **Lazy evaluation order** is when we don't evaluate function parameters before substituting them into function bodies
- As long as a language has no mutable state, a program that runs successfully will produce *the exact same results* with strict and lazy semantics
  - However, some programs might fail with strict but succeed with lazy
  - Some programs might run forever with strict, but terminate with lazy
  - Opposite never true: if strict succeeds, then lazy does too

Example: Error

Example: Error

```
{letvar f {fun {x y} {ifo x y {* x 2}}}
        {f 3 undefinedVariable}}
```

- Strict evaluation:

Example: Error

```
{letvar f {fun {x y} {ifo x y {* x 2}}}
        {f 3 undefinedVariable}}
```

- Strict evaluation:
  - Evaluates `undefinedVariable` before substituting it in the function body

Example: Error

```
{letvar f {fun {x y} {ifo x y {* x 2}}}
        {f 3 undefinedVariable}}
```

- Strict evaluation:
  - Evaluates `undefinedVariable` before substituting it in the function body
  - Raises an error because it's undefined

Example: Error

```
{letvar f {fun {x y} {ifo x y {* x 2}}}
        {f 3 undefinedVariable}}
```

- Strict evaluation:
  - Evaluates `undefinedVariable` before substituting it in the function body
  - Raises an error because it's undefined
- Doesn't raise an error with lazy evaluation

Example: Error

```
{letvar f {fun {x y} {ifo x y {* x 2}}}
        {f 3 undefinedVariable}}
```

- Strict evaluation:
  - Evaluates `undefinedVariable` before substituting it in the function body
  - Raises an error because it's undefined
- Doesn't raise an error with lazy evaluation
  - We never evaluate `undefinedVariable`

Example: Error

```
{letvar f {fun {x y} {ifo x y {* x 2}}}
        {f 3 undefinedVariable}}
```

- Strict evaluation:
  - Evaluates `undefinedVariable` before substituting it in the function body
  - Raises an error because it's undefined
- Doesn't raise an error with lazy evaluation
  - We never evaluate `undefinedVariable`
  - When we substitute, it only ends up in the branch of the `ifo` that we don't take

Example: Loop

Example: Loop

```
{letvar f {fun {x y} {ifo x y {* x 2}}}
        {f 3
           {{fun {x} {x x}} {fun {x} {x x}}}
           }}
```

- `{{fun {x} {x x}} {fun {x} {x x}}}` runs forever

Example: Loop

```
{letvar f {fun {x y} {ifo x y {* x 2}}}
        {f 3
           {{fun {x} {x x}} {fun {x} {x x}}}
           }}
```

- `{{fun {x} {x x}} {fun {x} {x x}}}` runs forever
  - Midterm bonus

Example: Loop

```
{letvar f {fun {x y} {ifo x y {* x 2}}}
        {f 3
           {{fun {x} {x x}} {fun {x} {x x}}}
           }}
```

- `{{fun {x} {x x}} {fun {x} {x x}}}` runs forever
  - Midterm bonus
- Strict evaluation:

Example: Loop

```
{letvar f {fun {x y} {ifo x y {* x 2}}}
        {f 3
           {{fun {x} {x x}} {fun {x} {x x}}}
           }}
```

- `{{fun {x} {x x}} {fun {x} {x x}}` runs forever
  - Midterm bonus
- Strict evaluation:
  - Evaluates the application that runs forever, never gets to substituting

Example: Loop

```
{letvar f {fun {x y} {ifo x y {* x 2}}}
        {f 3
           {{fun {x} {x x}} {fun {x} {x x}}}
           }}
```

- `{{fun {x} {x x}} {fun {x} {x x}}` runs forever
  - Midterm bonus
- Strict evaluation:
  - Evaluates the application that runs forever, never gets to substituting
- Terminates with lazy evaluation

Example: Loop

```
{letvar f {fun {x y} {ifo x y {* x 2}}}
        {f 3
           {{fun {x} {x x}} {fun {x} {x x}}}
           }}
```

- `{{fun {x} {x x}} {fun {x} {x x}}` runs forever
  - Midterm bonus
- Strict evaluation:
  - Evaluates the application that runs forever, never gets to substituting
- Terminates with lazy evaluation
  - We never evaluate the function calling itself

Example: Loop

```
{letvar f {fun {x y} {ifo x y {* x 2}}}
        {f 3
           {{fun {x} {x x}} {fun {x} {x x}}}
           }}
```

- `{{fun {x} {x x}} {fun {x} {x x}}` runs forever
  - Midterm bonus
- Strict evaluation:
  - Evaluates the application that runs forever, never gets to substituting
- Terminates with lazy evaluation
  - We never evaluate the function calling itself
  - When we substitute, it only ends up in the branch of the `ifo` that we don't take

The General Rule

- For function calls

The General Rule

- For function calls
  - Lazy: substitute the argument *expression* into the body, then evaluate the body

The General Rule

- For function calls
  - Lazy: substitute the argument *expression* into the body, then evaluate the body
  - Strict: evaluate the argument to a value, then substitute into the body, then evaluate the body

Non-Function Recursion

- Recall: with recursion, we needed to have all recursive references in the body of a lambda

Non-Function Recursion

- Recall: with recursion, we needed to have all recursive references in the body of a lambda
- With lazy evaluation, we don’t need this, because we won’t try to evaluate the variable

Non-Function Recursion

- Recall: with recursion, we needed to have all recursive references in the body of a lambda
- With lazy evaluation, we don't need this, because we won't try to evaluate the variable
- Lets us define **infinite data structures** using recursion

Non-Function Recursion

- Recall: with recursion, we needed to have all recursive references in the body of a lambda
- With lazy evaluation, we don't need this, because we won't try to evaluate the variable
- Lets us define **infinite data structures** using recursion
  - As long as we only ever access a finite part of the structure, program will terminate

Non-Function Recursion

- Recall: with recursion, we needed to have all recursive references in the body of a lambda
- With lazy evaluation, we don't need this, because we won't try to evaluate the variable
- Lets us define **infinite data structures** using recursion
  - As long as we only ever access a finite part of the structure, program will terminate
  - e.g. "the list of all natural numbers"

Non-Function Recursion

- Recall: with recursion, we needed to have all recursive references in the body of a lambda
- With lazy evaluation, we don't need this, because we won't try to evaluate the variable
- Lets us define **infinite data structures** using recursion
  - As long as we only ever access a finite part of the structure, program will terminate
  - e.g. "the list of all natural numbers"
- See Racket examples in `lazy.rkt`

Strictness Points

- Some features *had* to evaluate their arguments before using them

Strictness Points

- Some features *had* to evaluate their arguments before using them
  - e.g. the number we're testing in `if0`

Strictness Points

- Some features *had* to evaluate their arguments before using them
  - e.g. the number we're testing in `if0`
    - Can't progress until it's a value

Strictness Points

- Some features *had* to evaluate their arguments before using them
  - e.g. the number we're testing in `if0`
    - Can't progress until it's a value
  - These are *strictness points*

Strictness Points

- Some features *had* to evaluate their arguments before using them
  - e.g. the number we're testing in `if0`
    - Can't progress until it's a value
  - These are *strictness points*
    - Places an un-evaluated argument will be evaluated

Strictness Points

- Some features *had* to evaluate their arguments before using them
  - e.g. the number we're testing in `if0`
    - Can't progress until it's a value
  - These are *strictness points*
    - Places an un-evaluated argument will be evaluated
- Include

Strictness Points

- Some features *had* to evaluate their arguments before using them
  - e.g. the number we're testing in `if0`
    - Can't progress until it's a value
  - These are *strictness points*
    - Places an un-evaluated argument will be evaluated
- Include
  - Number tested in ~if0-

Strictness Points

- Some features *had* to evaluate their arguments before using them
  - e.g. the number we're testing in `if0`
    - Can't progress until it's a value
  - These are *strictness points*
    - Places an un-evaluated argument will be evaluated
- Include
  - Number tested in ~if0-
  - Arguments to plus or times

Strictness Points

- Some features *had* to evaluate their arguments before using them
  - e.g. the number we're testing in `if0`
    - Can't progress until it's a value
  - These are *strictness points*
    - Places an un-evaluated argument will be evaluated
- Include
  - Number tested in ~if0-
  - Arguments to plus or times
  - The *function* in a function call

Strictness Points

- Some features *had* to evaluate their arguments before using them
  - e.g. the number we're testing in `if0`
    - Can't progress until it's a value
  - These are *strictness points*
    - Places an un-evaluated argument will be evaluated
- Include
  - Number tested in ~if0~
  - Arguments to plus or times
  - The *function* in a function call
    - not the argument

Laziness in real languages

- Python and JavaScript have *generators*

Laziness in real languages

- Python and JavaScript have *generators*
  - A structure we can iterate from, that doesn't generate the entire list we're going through

Laziness in real languages

- Python and JavaScript have *generators*
  - A structure we can iterate from, that doesn't generate the entire list we're going through
  - This is basically just a lazy list, implemented similar way

Laziness in real languages

- Python and JavaScript have *generators*
  - A structure we can iterate from, that doesn't generate the entire list we're going through
  - This is basically just a lazy list, implemented similar way
- Can probably do something similar with iterators in C++

Laziness via lambda

- We can simulate laziness in an eager language using just functions

Laziness via lambda

- We can simulate laziness in an eager language using just functions
- A function that ignores its argument acts like a lazy value

Laziness via lambda

- We can simulate laziness in an eager language using just functions
- A function that ignores its argument acts like a lazy value
  - To force it back to a value, you call it with any value

Laziness via lambda example

Laziness via lambda example

```
{letvar f {fun {x y} {ifo x {y 0} {* x 2}}}
        {f 3 {fun {x} undefinedVariable}}}
```

- Undefined variable behind a lambda, so no error

Laziness via lambda example

```
{letvar f {fun {x y} {ifo x {y 0} {* x 2}}}
        {f 3 {fun {x} undefinedVariable}}}
```

- Undefined variable behind a lambda, so no error
  - Closures store *expressions*, not values

Laziness via lambda example

```
{letvar f {fun {x y} {ifo x {y 0} {* x 2}}}
        {f 3 {fun {x} undefinedVariable}}}
```

- Undefined variable behind a lambda, so no error
  - Closures store *expressions*, not values
- In a strict language, this has the same behavior as the lazy version