The Substitution Model

Concepts of Programming Languages
Lecture 15

CAS CS 320

Outline

Look formally at the **lambda calculus** and its semantics

Discuss **substitution** and the pitfalls to avoid

Demo an **implementation** of the lambda calculus

Recap

Recall: Small-Step Semantics

$$(S, p) \longrightarrow (S', p')$$

Recall: Small-Step Semantics

$$(S, p) \longrightarrow (S', p')$$

Small-step semantics formalizes a **"step by step"** computation which reduces a syntactic object until no reductions can be done

Recall: Small-Step Semantics

$$(S, p) \longrightarrow (S', p')$$

Small-step semantics formalizes a **"step by step"** computation which reduces a syntactic object until no reductions can be done

Notation. We write $e \longrightarrow e'$ to mean e reduces to e' in a single step

Recall: Small-Step Semantics

$$(S, p) \longrightarrow (S', p')$$

Small-step semantics formalizes a **"step by step"** computation which reduces a syntactic object until no reductions can be done

Notation. We write $e \longrightarrow e'$ to mean e reduces to e' in a single step

In general, we define small-step semantics on a **configuration,** which is a program together with some stateful information

Recall: Small-Step Semantics

$$(S, \underbrace{p}_{\text{program}}) \longrightarrow (S', \underbrace{p'}_{\text{transformed program}})$$

Small-step semantics formalizes a **"step by step"** computation which reduces a syntactic object until no reductions can be done

Notation. We write $e \longrightarrow e'$ to mean e reduces to e' in a single step

In general, we define small-step semantics on a **configuration**, which is a program together with some stateful information

Recall: Small-Step Semantics

$$(\underbrace{S}_{\text{state}}, \overbrace{p}^{\text{program}}) \longrightarrow (\underbrace{S'}_{\text{transformed state}}, \overbrace{p'}^{\text{transformed program}})$$

Small-step semantics formalizes a **"step by step"** computation which reduces a syntactic object until no reductions can be done

Notation. We write $e \longrightarrow e'$ to mean e reduces to e' in a single step

In general, we define small-step semantics on a **configuration,** which is a program together with some stateful information

Recall: Small-Step Semantics

$$\overbrace{(\underbrace{S}_{\text{state}}, \overbrace{p}^{\text{program}})}^{\text{configuration}} \longrightarrow \overbrace{(\underbrace{S'}_{\text{transformed state}}, \overbrace{p'}^{\text{transformed program}})}^{\text{transformed configuration}}$$

Small-step semantics formalizes a **"step by step"** computation which reduces a syntactic object until no reductions can be done

Notation. We write $e \longrightarrow e'$ to mean e reduces to e' in a single step

In general, we define small-step semantics on a **configuration,** which is a program together with some stateful information

Recall: Small-Step Semantics

$$\overbrace{\overbrace{(\underbrace{S}_{\text{state}}, \overbrace{p}^{\text{program}})}^{\text{configuration}} \longrightarrow \overbrace{(\underbrace{S'}_{\text{transformed state}}, \overbrace{p'}^{\text{transformed program}})}^{\text{transformed configuration}}}^{\text{single-step reduction}}$$

Small-step semantics formalizes a **"step by step"** computation which reduces a syntactic object until no reductions can be done

Notation. We write $e \longrightarrow e'$ to mean e reduces to e' in a single step

In general, we define small-step semantics on a **configuration,** which is a program together with some stateful information

Recall: High-Level

$$\underbrace{\overbrace{(S}_{\text{state}},\overbrace{p}^{\text{program}})} \longrightarrow (\underbrace{S'}_{\text{transformed state}},\overbrace{p'}^{\text{transformed program}})$$

Recall: High-Level

$$(\underbrace{S}_{\text{state}}, \overbrace{p}^{\text{program}}) \longrightarrow (\underbrace{S'}_{\text{transformed state}}, \overbrace{p'}^{\text{transformed program}})$$

When we define the small-step semantics of PL, we need to define three things:

Recall: High-Level

$$\underbrace{(S}_{\text{state}},\overbrace{p}^{\text{program}}) \longrightarrow (\underbrace{S'}_{\text{transformed state}},\overbrace{p'}^{\text{transformed program}})$$

When we define the small-step semantics of PL, we need to define three things:

» What kind of **state** are we manipulating?

Recall: High-Level

$$(\underset{\text{state}}{S}, \overset{\text{program}}{p}) \longrightarrow (\underset{\text{transformed state}}{S'}, \overset{\text{transformed program}}{p'})$$

When we define the small-step semantics of PL, we need to define three things:

» What kind of **state** are we manipulating?

» What **rules** describe how to transform configurations?

Recall: High-Level

$$\underbrace{(S}_{\text{state}}, \overbrace{p}^{\text{program}}) \longrightarrow (\underbrace{S'}_{\text{transformed state}}, \overbrace{p'}^{\text{transformed program}})$$

When we define the small-step semantics of PL, we need to define three things:

» What kind of **state** are we manipulating?

» What **rules** describe how to transform configurations?

» What are the **values** of our PL (i.e., when are we done reducing)?

Recall: Example

```
<expr> ::= ( <op> <expr> <expr> )
         | <bool> | <int>
<op>   ::= add | sub | eq
<bool> ::= true | false
<int>  ::= ...
```

Recall: Example

```
<expr> ::= ( <op> <expr> <expr> )
         | <bool> | <int>
<op>   ::= add | sub | eq
<bool> ::= true | false
<int>  ::= ...
```

State: $\varnothing$

Recall: Example

```
<expr> ::= ( <op> <expr> <expr> )
         | <bool> | <int>
<op>   ::= add | sub | eq
<bool> ::= true | false
<int>  ::= ...
```

State: $\varnothing$

Rules:

$$\frac{e_1 \longrightarrow e_1'}{(\texttt{add}\ e_1\ e_2) \longrightarrow (\texttt{add}\ e_1'\ e_2)}\ \texttt{add-left} \qquad \frac{n\ \texttt{is a number} \qquad e_2 \longrightarrow e_2'}{(\texttt{add}\ n\ e_2) \longrightarrow (\texttt{add}\ n\ e_2')}\ \texttt{add-right}$$

$$\frac{n_1\ \texttt{is a number} \qquad n_2\ \texttt{is a number}}{(\texttt{add}\ n_1\ n_2) \longrightarrow n_1 + n_2}\ \texttt{add-ok}$$

$$\frac{e_1 \longrightarrow e_1'}{(\texttt{sub}\ e_1\ e_2) \longrightarrow (\texttt{sub}\ e_1'\ e_2)}\ \texttt{sub-left} \qquad \frac{n\ \texttt{is a number} \qquad e_2 \longrightarrow e_2'}{(\texttt{sub}\ n\ e_2) \longrightarrow (\texttt{sub}\ n\ e_2')}\ \texttt{sub-right}$$

$$\frac{n_1\ \texttt{is a number} \qquad n_2\ \texttt{is a number}}{(\texttt{sub}\ n_1\ n_2) \longrightarrow n_1 - n_2}\ \texttt{sub-ok}$$

Recall: Example

```
<expr> ::= ( <op> <expr> <expr> )
         | <bool> | <int>
<op>   ::= add | sub | eq
<bool> ::= true | false
<int>  ::= ...
```

State: $\varnothing$

Rules:

$$\frac{e_1 \longrightarrow e_1'}{(\texttt{add}\ e_1\ e_2) \longrightarrow (\texttt{add}\ e_1'\ e_2)}\ \texttt{add-left} \qquad \frac{n \text{ is a number} \qquad e_2 \longrightarrow e_2'}{(\texttt{add}\ n\ e_2) \longrightarrow (\texttt{add}\ n\ e_2')}\ \texttt{add-right}$$

$$\frac{n_1 \text{ is a number} \qquad n_2 \text{ is a number}}{(\texttt{add}\ n_1\ n_2) \longrightarrow n_1 + n_2}\ \texttt{add-ok}$$

$$\frac{e_1 \longrightarrow e_1'}{(\texttt{sub}\ e_1\ e_2) \longrightarrow (\texttt{sub}\ e_1'\ e_2)}\ \texttt{sub-left} \qquad \frac{n \text{ is a number} \qquad e_2 \longrightarrow e_2'}{(\texttt{sub}\ n\ e_2) \longrightarrow (\texttt{sub}\ n\ e_2')}\ \texttt{sub-right}$$

$$\frac{n_1 \text{ is a number} \qquad n_2 \text{ is a number}}{(\texttt{sub}\ n_1\ n_2) \longrightarrow n_1 - n_2}\ \texttt{sub-ok}$$

Values: <int> (i.e., numbers)

Recall: Example

```
<expr> ::= ( <op> <expr> <expr> )
         | <bool> | <int>
<op>   ::= add | sub | eq
<bool> ::= true | false
<int>  ::= ...
```

$$\frac{n \texttt{ is a number}}{n \Downarrow n}\ \texttt{numEval}$$

$$\frac{e_1 \Downarrow v_1 \qquad e_2 \Downarrow v_2 \qquad v_1 \texttt{ is a number} \qquad v_2 \texttt{ is a number}}{(\texttt{add}\ e_1\ e_2) \Downarrow v_1 + v_2}\ \texttt{addEval}$$

$$\frac{e_1 \Downarrow v_1 \qquad e_2 \Downarrow v_2 \qquad v_1 \texttt{ is a number} \qquad v_2 \texttt{ is a number}}{(\texttt{sub}\ e_1\ e_2) \Downarrow v_1 - v_2}\ \texttt{subEval}$$

We can also give a big-step semantics to this system

```
<prog> ::= { <stmt> ; }
<stmt> ::= rot90 | refX | refY
```

Practice Problem

$$\frac{}{(s,\ \texttt{rot90;}\ P) \longrightarrow (s\ \texttt{rotated 90 deg. clockwise},\ P)}$$

$$\frac{}{(s,\ \texttt{refX;}\ P) \longrightarrow (s\ \texttt{reflected across x-axis},\ P)}$$

$$\frac{}{(s,\ \texttt{refY;}\ P) \longrightarrow (s\ \texttt{reflected across y-axis},\ P)}$$

What does (△, `rot90; refY; rot90; refX;`) *evaluate to?*
Give a sequence of single step reductions (you do not need to give the full multi-step derivation)

Answer

```
<prog> ::= { <stmt> ; }
<stmt> ::= rot90 | refX | refY
```

(△, rot90; refY; rot90; refX;)

The Lambda Calculus

High-Level View

```
(fun x -> x x)(fun x -> x x)
```

lambda term called Ω

High-Level View

```
(fun x -> x x)(fun x -> x x)
```

lambda term called Ω

The **lambda calculus** is the *simplest functional programming language*. It only has:

High-Level View

```
(fun x -> x x)(fun x -> x x)
```

lambda term called Ω

The **lambda calculus** is the *simplest functional programming language*. It only has:

» variables

High-Level View

```
(fun x -> x x)(fun x -> x x)
```

lambda term called Ω

The **lambda calculus** is the *simplest functional programming language*. It only has:

» variables
» anonymous functions

High-Level View

```
(fun x -> x x)(fun x -> x x)
```

lambda term called Ω

The **lambda calculus** is the *simplest functional programming language*. It only has:

» variables
» anonymous functions
» function application

High-Level View

```
(fun x -> x x)(fun x -> x x)
```

lambda term called Ω

The **lambda calculus** is the *simplest functional programming language*. It only has:

» variables
» anonymous functions
» function application

It's also **untyped**, so *anything* can be applied to *anything*

demo

(OCaml and Python)

History

History

The lambda calculus was introduced by **Alonzo Church** in the 1930s

History

The lambda calculus was introduced by **Alonzo Church** in the 1930s

Church was trying to give a *foundation of mathematics* (did not succeed) and extracted from that work the lambda calculus

History

The lambda calculus was introduced by **Alonzo Church** in the 1930s

Church was trying to give a *foundation of mathematics* (did not succeed) and extracted from that work the lambda calculus

The lambda calculus **as powerful** as every model of computation (Turing Machines, Register Machines, etc.)

Syntax

```
<expr> ::= fun <var> -> <expr>
         | <var>
         | <expr> <expr>
         | ( <expr> )
<var>  ::= a | b | ... | y | z
```

Syntax

```
<expr> ::= fun <var> -> <expr>
         | <var>
         | <expr> <expr>
         | ( <expr> )
<var>  ::= a | b | ... | y | z
```

This presentation is technically ambiguous (why?)

Syntax

```
<expr> ::= fun <var> -> <expr>
         | <var>
         | <expr> <expr>
         | ( <expr> )
<var>  ::= a | b | ... | y | z
```

This presentation is technically ambiguous (why?)

We assume application is **left-associative** and has higher precedence than the anonymous function syntax

Syntax (Unambiguous)

```
<expr>  ::= fun <var> -> <expr>
          | <expr2> { <expr2> }
<expr2> ::= <var> | ( <expr> )
<var>   ::= a | b | ... | y | z
```

In this grammar we can only use variables or functions in parentheses in applications

Syntax (Mathematical)

```
<expr> ::= λ<var>.<expr>
         | <var>
         | <expr><expr>
```

In mathematical settings, we use more compact syntax

Parentheses, precedence, and variables are often left implicit

Warning: Get used to λ

$$\lambda x\,.\,e \qquad \equiv \qquad \texttt{fun}\ x\ \texttt{->}\ e$$

We will use these syntaxes interchangeably starting now

These are the same thing, get used to it

Values

```
<val> ::= λ<var>.<expr>
```

Values

<val> ::= λ<var>.<expr>

In arithmetic, values are **numbers**

Values

$$\texttt{<val> ::= }\lambda\texttt{<var>.<expr>}$$

In arithmetic, values are **numbers**

In the lambda calculus, values are **functions**

Values

`<val> ::= λ<var>.<expr>`

In arithmetic, values are **numbers**

In the lambda calculus, values are **functions**

We often use BNF syntax to specify our values

Small-Step Semantics

(1) $$\frac{e_1 \longrightarrow e_1'}{e_1 e_2 \longrightarrow e_1' e_2}$$

(2) $$\frac{e_2 \longrightarrow e_2'}{(\lambda x \,.\, e_1) e_2 \longrightarrow (\lambda x \,.\, e_1) e_2'}$$

(3) $$\frac{}{(\lambda x \,.\, e)(\lambda y \,.\, e') \longrightarrow [(\lambda y \,.\, e')/x]e}$$

Small-Step Semantics

$$(1)\ \frac{e_1 \longrightarrow e_1'}{e_1 e_2 \longrightarrow e_1' e_2}$$

$$(2)\ \frac{e_2 \longrightarrow e_2'}{(\lambda x\,.\,e_1)e_2 \longrightarrow (\lambda x\,.\,e_1)e_2'}$$

$$(3)\ \frac{}{(\lambda x\,.\,e)(\lambda y\,.\,e') \longrightarrow [(\lambda y\,.\,e')/x]e}$$

1. We can reduce the LHS of an application

Small-Step Semantics

$$(1)\ \frac{e_1 \longrightarrow e_1'}{e_1 e_2 \longrightarrow e_1' e_2}$$

$$(2)\ \frac{e_2 \longrightarrow e_2'}{(\lambda x\,.\,e_1)e_2 \longrightarrow (\lambda x\,.\,e_1)e_2'}$$

$$(3)\ \frac{}{(\lambda x\,.\,e)(\lambda y\,.\,e') \longrightarrow [(\lambda y\,.\,e')/x]e}$$

1. We can reduce the LHS of an application
2. We can reduce the RHS of an application *if the LHS is already a function*

Small-Step Semantics

(1) $$\frac{e_1 \longrightarrow e_1'}{e_1 e_2 \longrightarrow e_1' e_2}$$

(2) $$\frac{e_2 \longrightarrow e_2'}{(\lambda x \,.\, e_1) e_2 \longrightarrow (\lambda x \,.\, e_1) e_2'}$$

(3) $$\frac{}{(\lambda x \,.\, e)(\lambda y \,.\, e') \longrightarrow [(\lambda y \,.\, e')/x]e}$$

1. We can reduce the LHS of an application
2. We can reduce the RHS of an application *if the LHS is already a function*
3. We can apply a function to another function by **substitution.** This is also called β**–reduction**

Example

$$(1)\ \frac{e_1 \longrightarrow e_1'}{e_1 e_2 \longrightarrow e_1' e_2} \qquad (2)\ \frac{e_2 \longrightarrow e_2'}{(\lambda x\,.\,e_1)e_2 \longrightarrow (\lambda x\,.\,e_1)e_2'}$$

$$(3)\ \frac{}{(\lambda x\,.\,e)(\lambda y\,.\,e') \longrightarrow [(\lambda y\,.\,e')/x]e}$$

$$(\lambda f\,.\,\lambda x\,.\,fx)(\lambda y\,.\,y)$$

Small-Step Semantics (Another Form)

$$(1)\ \frac{e_1 \longrightarrow e_1'}{e_1 e_2 \longrightarrow e_1' e_2}$$

$$(2)\ \frac{}{(\lambda x\,.\,e)e' \longrightarrow [e'/x]e}$$

Small-Step Semantics (Another Form)

$$(1)\ \frac{e_1 \longrightarrow e_1'}{e_1 e_2 \longrightarrow e_1' e_2}$$

$$(2)\ \frac{}{(\lambda x\,.\,e)e' \longrightarrow [e'/x]e}$$

1. We can reduce the LHS of an application

Small-Step Semantics (Another Form)

$$(1)\ \frac{e_1 \longrightarrow e_1'}{e_1 e_2 \longrightarrow e_1' e_2}$$

$$(2)\ \frac{}{(\lambda x\,.\,e)e' \longrightarrow [e'/x]e}$$

1. We can reduce the LHS of an application
2. We can apply a function to *any* expression via substitution

Example

$$(1)\ \frac{e_1 \longrightarrow e_1'}{e_1 e_2 \longrightarrow e_1' e_2}$$

$$(2)\ \frac{}{(\lambda x \,.\, e)e' \longrightarrow [e'/x]e}$$

$$(\lambda x \,.\, y)((\lambda z \,.\, z)(\lambda w \,.\, w))$$

Recall: Values

When evaluating, there are **three** "end" cases to evaluation:

» **value:** we reach the end of our computation and the value of our program

» **stuck:** we reach an expression that cannot be reduced, but that is not a value

» **diverge:** the computation never reaches a point where the expression is not reducible

Stuck Terms

`<val> ::= λ<var>.<expr>`

$$\frac{e_1 \longrightarrow e_1'}{e_1 e_2 \longrightarrow e_1' e_2} \qquad \frac{}{(\lambda x \,.\, e)e' \longrightarrow [e'/x]e}$$

$$(\lambda x \,.\, yx)(\lambda x \,.\, x)$$

Based on our operational semantics, it's possible for the above expression to reduce to a value

Non-Termination

$$\frac{e_1 \longrightarrow e_1'}{e_1 e_2 \longrightarrow e_1' e_2} \qquad \frac{}{(\lambda x\,.\,e)e' \longrightarrow [e'/x]e}$$

$$(\lambda x\,.\,xx)(\lambda x\,.\,xx)$$

And unlike with arithmetic, it's now possible to define expressions which **do not terminate.** These expression do not have values, but also don't get stuck

Big-Step Semantics

$$(1)\ \frac{}{\lambda x\,.\,e \Downarrow \lambda x\,.\,e}$$

$$(2)\ \frac{e_1 \Downarrow \lambda x\,.\,e \qquad e_2 \Downarrow v_2 \qquad [v_2/x]e \Downarrow v}{e_1 e_2 \Downarrow v}$$

Big-Step Semantics

$$\text{(1)}\ \frac{}{\lambda x\,.\,e \Downarrow \lambda x\,.\,e}$$

$$\text{(2)}\ \frac{e_1 \Downarrow \lambda x\,.\,e \qquad e_2 \Downarrow v_2 \qquad [v_2/x]e \Downarrow v}{e_1 e_2 \Downarrow v}$$

1. A function evaluates to a function value (itself)

Big-Step Semantics

$$(1)\ \frac{}{\lambda x\,.\,e \Downarrow \lambda x\,.\,e}$$

$$(2)\ \frac{e_1 \Downarrow \lambda x\,.\,e \qquad e_2 \Downarrow v_2 \qquad [v_2/x]e \Downarrow v}{e_1 e_2 \Downarrow v}$$

1. A function evaluates to a function value (itself)

2. If e_1 evaluates to the function $\lambda x\,.\,e$ and e_2 evaluates to the value v_2 and e with v_2 substituted for x evaluates to v, then the application $e_1 e_2$ evaluates to v

Big-Step Semantics (Another Form)

$$\frac{}{\lambda x\,.\,e \Downarrow \lambda x\,.\,e}$$

$$\frac{e_1 \Downarrow \lambda x\,.\,e \qquad [e_2/x]e \Downarrow v}{e_1 e_2 \Downarrow v}$$

These are the same rules as before except we're not required to evaluate e_2 first

Big-Step Semantics (Another Form)

$$\frac{}{\lambda x \,.\, e \Downarrow \lambda x \,.\, e}$$

$$\frac{e_1 \Downarrow \lambda x \,.\, e \qquad [e_2/x]e \Downarrow v}{e_1 e_2 \Downarrow v}$$

These are the same rules as before except we're not required to evaluate e_2 first

Practice Problem

$$(\lambda x . \lambda y . y)((\lambda z . z)(\lambda q . q)) \Downarrow \lambda y . y$$

Give a derivation of the above judgment in both versions of the big-step semantics

$$\frac{}{\lambda x . e \Downarrow \lambda x . e}$$

$$\frac{e_1 \Downarrow \lambda x . e \qquad e_2 \Downarrow v_2 \qquad [v_2/x]e \Downarrow v}{e_1 e_2 \Downarrow v}$$

$$\frac{}{\lambda x . e \Downarrow \lambda x . e}$$

$$\frac{e_1 \Downarrow \lambda x . e \qquad [e_2/x]e \Downarrow v}{e_1 e_2 \Downarrow v}$$

Answer

$$(\lambda x \,.\, \lambda y \,.\, y)((\lambda z \,.\, z)(\lambda q \,.\, q)) \Downarrow \lambda y \,.\, y$$

Call-by-Value vs. Call-by-Name

$$\frac{e_1 \Downarrow \lambda x\,.\,e_1' \qquad e_2 \Downarrow v_2 \qquad [v_2/x]e_1' \Downarrow v}{e_1 e_2 \Downarrow v}\ \texttt{(CBV)}$$

$$\frac{e_1 \Downarrow \lambda x\,.\,e_1' \qquad [e_2/x]e_1' \Downarrow v}{e_1 e_2 \Downarrow v}\ \texttt{(CBN)}$$

Call-by-Value vs. Call-by-Name

$$\frac{e_1 \Downarrow \lambda x \,.\, e_1' \qquad e_2 \Downarrow v_2 \qquad [v_2/x]e_1' \Downarrow v}{e_1 e_2 \Downarrow v}\ \texttt{(CBV)}$$

$$\frac{e_1 \Downarrow \lambda x \,.\, e_1' \qquad [e_2/x]e_1' \Downarrow v}{e_1 e_2 \Downarrow v}\ \texttt{(CBN)}$$

The two versions of semantics we've given correspond to **call-by-value** (CBV) and **call-by-name** (CBN). These are **evaluation strategies** for functional languages

Call-by-Value vs. Call-by-Name

$$\frac{e_1 \Downarrow \lambda x \, . \, e_1' \qquad e_2 \Downarrow v_2 \qquad [v_2/x]e_1' \Downarrow v}{e_1 e_2 \Downarrow v} \text{(CBV)}$$

$$\frac{e_1 \Downarrow \lambda x \, . \, e_1' \qquad [e_2/x]e_1' \Downarrow v}{e_1 e_2 \Downarrow v} \text{(CBN)}$$

The two versions of semantics we've given correspond to **call-by-value** (CBV) and **call-by-name** (CBN). These are **evaluation strategies** for functional languages

<u>CBV:</u> evaluate the argument of a function *before* substituting it in the function

Call-by-Value vs. Call-by-Name

$$\frac{e_1 \Downarrow \lambda x \,.\, e_1' \qquad e_2 \Downarrow v_2 \qquad [v_2/x]e_1' \Downarrow v}{e_1 e_2 \Downarrow v}\ (\mathtt{CBV})$$

$$\frac{e_1 \Downarrow \lambda x \,.\, e_1' \qquad [e_2/x]e_1' \Downarrow v}{e_1 e_2 \Downarrow v}\ (\mathtt{CBN})$$

The two versions of semantics we've given correspond to **call-by-value** (CBV) and **call-by-name** (CBN). These are **evaluation strategies** for functional languages

<u>CBV:</u> evaluate the argument of a function *before* substituting it in the function

<u>CBN:</u> substitute the expression *directly* into the function

$$\frac{}{\lambda x\,.\,e \Downarrow \lambda x\,.\,e} \qquad \frac{e_1 \Downarrow \lambda x\,.\,e_1' \qquad e_2 \Downarrow v_2 \qquad [v_2/x]e_1' \Downarrow v}{e_1 e_2 \Downarrow v}$$

Benefits of CBV

$$(\lambda x\,.\,x + x + x + x)e$$

Benefits of CBV

$$\frac{}{\lambda x\,.\,e \Downarrow \lambda x\,.\,e} \qquad \frac{e_1 \Downarrow \lambda x\,.\,e_1' \qquad e_2 \Downarrow v_2 \qquad [v_2/x]e_1' \Downarrow v}{e_1 e_2 \Downarrow v}$$

$$(\lambda x\,.\,x + x + x + x)e$$

If we compute the value of an argument before substituting it into the expression, we only have to compute the expression *once*

Benefits of CBV

$$\frac{}{\lambda x\,.\,e \Downarrow \lambda x\,.\,e} \qquad \frac{e_1 \Downarrow \lambda x\,.\,e_1' \qquad e_2 \Downarrow v_2 \qquad [v_2/x]e_1' \Downarrow v}{e_1 e_2 \Downarrow v}$$

$$(\lambda x\,.\,x + x + x + x)e$$

If we compute the value of an argument before substituting it into the expression, we only have to compute the expression *once*

This is good if the variable appears several times in the body of our function

Benefits of CBV

$$\frac{}{\lambda x\,.\,e \Downarrow \lambda x\,.\,e} \qquad \frac{e_1 \Downarrow \lambda x\,.\,e_1' \qquad e_2 \Downarrow v_2 \qquad [v_2/x]e_1' \Downarrow v}{e_1 e_2 \Downarrow v}$$

$$(\lambda x\,.\,x + x + x + x)e$$

If we compute the value of an argument before substituting it into the expression, we only have to compute the expression *once*

This is good if the variable appears several times in the body of our function

This is also called **eager**, or **applicative**, or **strict** evaluation (and is what OCaml does)

Benefits of CBN

$$\frac{}{\lambda x \,.\, e \Downarrow \lambda x \,.\, e} \qquad \frac{e_1 \Downarrow \lambda x \,.\, e_1' \qquad [e_2/x]e_1' \Downarrow v}{e_1 e_2 \Downarrow v}$$

$$(\lambda x \,.\, \lambda y \,.\, x)e_1 e_2$$

Benefits of CBN

$$\frac{}{\lambda x\,.\,e \Downarrow \lambda x\,.\,e} \qquad \frac{e_1 \Downarrow \lambda x\,.\,e_1' \qquad [e_2/x]e_1' \Downarrow v}{e_1 e_2 \Downarrow v}$$

$$(\lambda x\,.\,\lambda y\,.\,x)e_1 e_2$$

If a variables doesn't appear in our function, then the argument is *not evaluated at all*

Benefits of CBN

$$\frac{}{\lambda x\,.\,e \Downarrow \lambda x\,.\,e} \qquad \frac{e_1 \Downarrow \lambda x\,.\,e_1' \qquad [e_2/x]e_1' \Downarrow v}{e_1 e_2 \Downarrow v}$$

$$(\lambda x\,.\,\lambda y\,.\,x)e_1 e_2$$

If a variables doesn't appear in our function, then the argument is *not evaluated at all*

If an argument is only seldomly used, it will only be computed when it is used (e.g, if its computed in a branch of an if-expression that is almost never reached)

Benefits of CBN

$$\frac{}{\lambda x\,.\,e \Downarrow \lambda x\,.\,e} \qquad \frac{e_1 \Downarrow \lambda x\,.\,e_1' \qquad [e_2/x]e_1' \Downarrow v}{e_1 e_2 \Downarrow v}$$

$$(\lambda x\,.\,\lambda y\,.\,x)e_1 e_2$$

If a variables doesn't appear in our function, then the argument is *not evaluated at all*

If an argument is only seldomly used, it will only be computed when it is used (e.g, if its computed in a branch of an if-expression that is almost never reached)

Aside. It's possible to simulate CBN in CBV. Think about it for a bit, ask me after if you're interested

Side Effects

```
let f x = x + x in
let y =
  let _ = print_int 2 in
  2
in f y
```

Side Effects

```
let f x = x + x in
let y =
  let _ = print_int 2 in
  2
in f y
```

What does this program print? ***It depends on if we're using CBN or CBV evaluation***

Side Effects

```
let f x = x + x in
let y =
  let _ = print_int 2 in
  2
in f y
```

What does this program print? ***It depends on if we're using CBN or CBV evaluation***

Definition. *(informal)* A **side effect** refers to something that happens during the evaluation of a program that is not a part of the formal semantics, e.g., printing to the console

Side Effects

```
let f x = x + x in
let y =
  let _ = print_int 2 in
  2
in f y
```

What does this program print? ***It depends on if we're using CBN or CBV evaluation***

Definition. *(informal)* A **side effect** refers to something that happens during the evaluation of a program that is not a part of the formal semantics, e.g., printing to the console

Different evaluation strategies yield different side-effectful behavior!

Aside: Evaluation Strategies

Aside: Evaluation Strategies

There are *a lot* more evaluation strategies, all of which optimize *something*

Aside: Evaluation Strategies

There are *a lot* more evaluation strategies, all of which optimize *something*

Haskell uses **lazy** evaluation also called **call-by-need**

Aside: Evaluation Strategies

There are *a lot* more evaluation strategies, all of which optimize *something*

Haskell uses **lazy** evaluation also called **call-by-need**

In languages with pointers we also often have the option to use **call-by-reference** evaluation or **call-by-sharing**

Aside: Evaluation Strategies

There are *a lot* more evaluation strategies, all of which optimize *something*

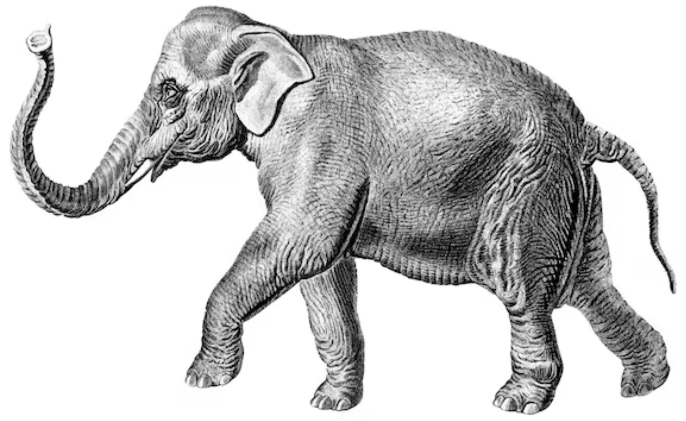
Haskell uses **lazy** evaluation also called **call-by-need**

In languages with pointers we also often have the option to use **call-by-reference** evaluation or **call-by-sharing**

*We will exclusively implement **call-by-value** (because, again, this is what OCaml does)*

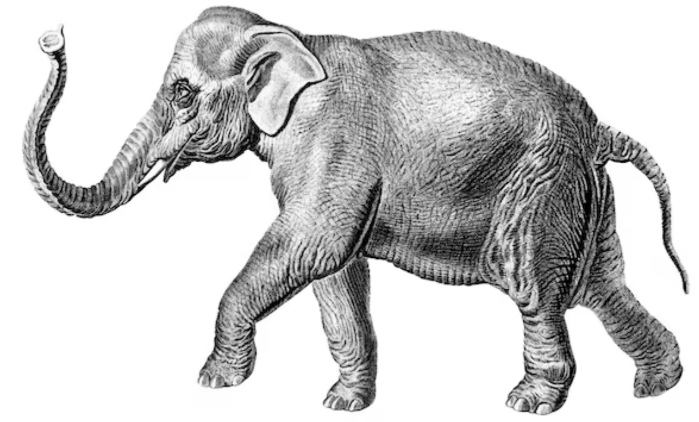
Substitution

Substitution



$$\frac{e_1 \Downarrow \lambda x\,.\,e \qquad e_2 \Downarrow v_2 \qquad [v_2/x]e \Downarrow v}{e_1 e_2 \Downarrow v}$$

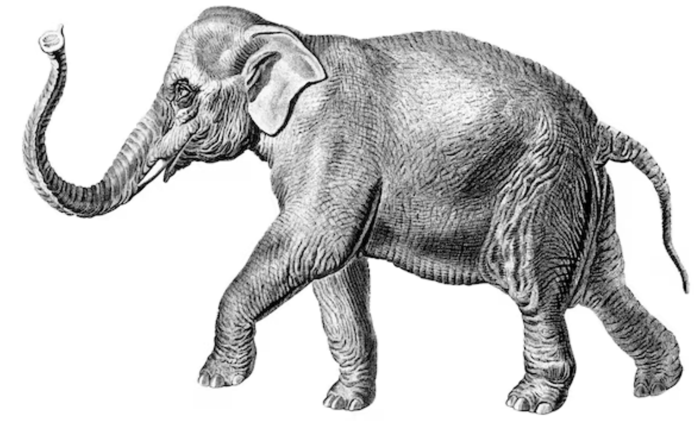
Substitution



$$\frac{e_1 \Downarrow \lambda x\,.\,e \qquad e_2 \Downarrow v_2 \qquad [v_2/x]e \Downarrow v}{e_1 e_2 \Downarrow v}$$

It's time to get more formal about **substitution**

Substitution

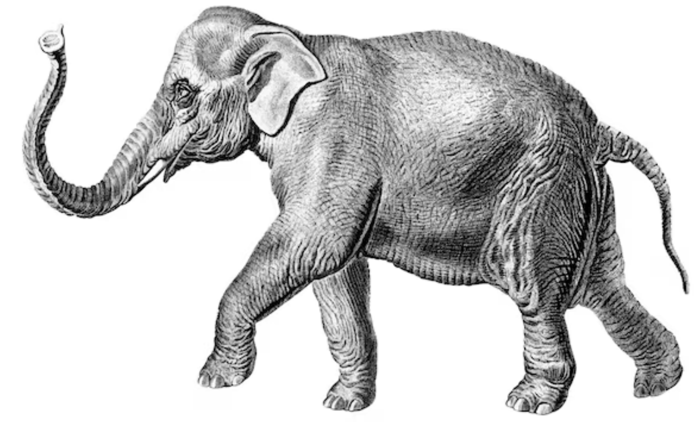


$$\frac{e_1 \Downarrow \lambda x \,.\, e \qquad e_2 \Downarrow v_2 \qquad [v_2/x]e \Downarrow v}{e_1 e_2 \Downarrow v}$$

It's time to get more formal about **substitution**

We've been able to get by on our intuitions for a while, but our intuitions won't help us *implement* substitution (which is *difficult*)

Substitution



$$\frac{e_1 \Downarrow \lambda x \,.\, e \qquad e_2 \Downarrow v_2 \qquad [v_2/x]e \Downarrow v}{e_1 e_2 \Downarrow v}$$

It's time to get more formal about **substitution**

We've been able to get by on our intuitions for a while, but our intuitions won't help us *implement* substitution (which is *difficult*)

We need to understand why...

Notation

$$[y/x](\lambda x\ .\ y)$$

Notation

$$[y/x](\lambda x\,.\,y)$$

We write $[v/x]e$ to mean e with v substituted in for x

Notation

$$[y/x](\lambda x\,.\,y)$$

We write $[v/x]e$ to mean e with v substituted in for x

Informally. *Replace every instance of* x *with* v

Notation

$$[y/x](\lambda x\,.\,y)$$

We write $[v/x]e$ to mean e with v substituted in for x

Informally. *Replace every instance of x with v*

Already things start to break down with this informal definition, e.g., consider the above substitution...

The Idea

$$[y/x](\lambda x\ .\ y)$$

The Idea

$$[y/x](\lambda x\,.\,y)$$

However we define substitution shouldn't *change the underlying behavior of a function*

The Idea

$$[y/x](\lambda x \,.\, y)$$

However we define substitution shouldn't *change the underlying behavior of a function*

The Key Point: A function does not depend on our choice of variable names

α-Equivalence

```
let x = 2 in x + 1
```

$=_\alpha$

```
let z = 2 in z + 1
```

OCaml

$$\lambda x\,.\,\lambda y\,.\,x =_\alpha \lambda v\,.\,\lambda w\,.\,v$$

λ-calculus

α-Equivalence

```
let x = 2 in x + 1
```

$=_\alpha$

```
let z = 2 in z + 1
```

OCaml

$$\lambda x.\lambda y.x =_\alpha \lambda v.\lambda w.v$$

λ-calculus

The **principle of name irrelevance** says that any two programs that are the same up to "renaming of variables" should behave exactly the same way (they are α**-equivalent**)

α-Equivalence

```
let x = 2 in x + 1
```

$=_\alpha$

```
let z = 2 in z + 1
```

OCaml

$$\lambda x\,.\,\lambda y\,.\,x =_\alpha \lambda v\,.\,\lambda w\,.\,v$$

λ-calculus

The **principle of name irrelevance** says that any two programs that are the same up to "renaming of variables" should behave exactly the same way (they are **α-equivalent**)

We say that a variable x is **bound** is an expression if it appears in the expression as $(\ldots\lambda x\,.\,e\ldots)$

α-Equivalence

```
let x = 2 in x + 1
```

$=_\alpha$

```
let z = 2 in z + 1
```

OCaml

$$\lambda x . \lambda y . x =_\alpha \lambda v . \lambda w . v$$

λ-calculus

The **principle of name irrelevance** says that any two programs that are the same up to "renaming of variables" should behave exactly the same way (they are **α-equivalent**)

We say that a variable x is **bound** is an expression if it appears in the expression as $(\ldots\lambda x . e \ldots)$

Substitution should preserve this

Definition (First Attempt)

$$[v/y]x = \begin{cases} v & x = y \\ x & \texttt{else} \end{cases} \tag{1}$$

$$[v/y](\lambda x \,.\, e) = \lambda x \,.\, [v/y]e \tag{2}$$

$$[v/y](e_1 e_2) = ([v/y]e_1)([v/y]e_2) \tag{3}$$

Definition (First Attempt)

$$[v/y]x = \begin{cases} v & x = y \\ x & \texttt{else} \end{cases} \tag{1}$$

$$[v/y](\lambda x\,.\,e) = \lambda x\,.\,[v/y]e \tag{2}$$

$$[v/y](e_1 e_2) = ([v/y]e_1)([v/y]e_2) \tag{3}$$

1. Replace every y with v, leave other variables

Definition (First Attempt)

$$[v/y]x = \begin{cases} v & x = y \\ x & \texttt{else} \end{cases} \tag{1}$$

$$[v/y](\lambda x \,.\, e) = \lambda x \,.\, [v/y]e \tag{2}$$

$$[v/y](e_1 e_2) = ([v/y]e_1)([v/y]e_2) \tag{3}$$

1. Replace every y with v, leave other variables
2. Replace y with v in the body of a function

Definition (First Attempt)

$$[v/y]x = \begin{cases} v & x = y \\ x & \texttt{else} \end{cases} \tag{1}$$

$$[v/y](\lambda x\,.\,e) = \lambda x\,.\,[v/y]e \tag{2}$$

$$[v/y](e_1 e_2) = ([v/y]e_1)([v/y]e_2) \tag{3}$$

1. Replace every y with v, leave other variables

2. Replace y with v in the body of a function

3. Replace y with v in both subexpressions of an application

(This is an example of an *inductive definition*)

$$[v/y]x = \begin{cases} v & x = y \\ x & \texttt{else} \end{cases}$$

$$[v/y](\lambda x \,.\, e) = \lambda x \,.\, [v/y]e$$

$$[v/y](e_1 e_2) = ([v/y]e_1)([v/y]e_2)$$

Problem Case I

$$[y/x](\lambda x \,.\, x)$$

We shouldn't be allowed to substitute x if it's the argument of a function

This may *change the behavior* of a function

Definition (Second Attempt)

$$[v/y]x = \begin{cases} v & x = y \\ x & \texttt{else} \end{cases}$$

$$[v/y](\lambda x\,.\,e) = \boxed{\begin{cases} \lambda x\,.\,e & x = y \\ \lambda x\,.\,[v/y]e & \texttt{else} \end{cases}}$$

$$[v/y](e_1 e_2) = ([v/y]e_1)([v/y]e_2)$$

We can handle the problem case directly in our definition. *Check the bound variable before we substitute in the body of a function*

Is there still a problem?

$$[v/y]x = \begin{cases} v & x = y \\ x & \texttt{else} \end{cases}$$

$$[v/y](\lambda x \,.\, e) = \begin{cases} \lambda x \,.\, e & x = y \\ \lambda x \,.\, [v/y]e & \texttt{else} \end{cases}$$

$$[v/y](e_1 e_2) = ([v/y]e_1)([v/y]e_2)$$

Problem Case II

$$[y/x](\lambda y \,.\, x)$$

We're not replacing a bound variable, but we *are* substituting an expression that has variables which *became* bound

The variable y is said to be **captured** in this (incorrect) substitution

Free and Bound Variables

$$FV(x) = \{x\} \tag{1}$$

$$FV(\lambda x . e) = FV(e) \backslash \{x\} \tag{2}$$

$$FV(e_1 e_2) = FV(e_1) \cup FV(e_2) \tag{3}$$

Free and Bound Variables

$$FV(x) = \{x\} \tag{1}$$

$$FV(\lambda x\,.\,e) = FV(e) \setminus \{x\} \tag{2}$$

$$FV(e_1 e_2) = FV(e_1) \cup FV(e_2) \tag{3}$$

Definition. A variable x is **free** in e if it does not appear **bound** by a λ.
Formally:

Free and Bound Variables

$$FV(x) = \{x\} \tag{1}$$

$$FV(\lambda x\,.\,e) = FV(e) \setminus \{x\} \tag{2}$$

$$FV(e_1 e_2) = FV(e_1) \cup FV(e_2) \tag{3}$$

Definition. A variable x is **free** in e if it does not appear **bound** by a λ. Formally:

1. x is free in x

Free and Bound Variables

$$FV(x) = \{x\} \tag{1}$$

$$FV(\lambda x\,.\,e) = FV(e) \backslash \{x\} \tag{2}$$

$$FV(e_1 e_2) = FV(e_1) \cup FV(e_2) \tag{3}$$

Definition. A variable x is **free** in e if it does not appear **bound** by a λ. Formally:

1. x is free in x

2. x is free in $\lambda y\,.\,e$ if it is free in e and $x \neq y$

Free and Bound Variables

$$FV(x) = \{x\} \quad (1)$$

$$FV(\lambda x\,.\,e) = FV(e) \backslash \{x\} \quad (2)$$

$$FV(e_1 e_2) = FV(e_1) \cup FV(e_2) \quad (3)$$

Definition. A variable x is **free** in e if it does not appear **bound** by a λ. Formally:

1. x is free in x

2. x is free in $\lambda y\,.\,e$ if it is free in e and $x \neq y$

3. x is free in $e_1 e_2$ if x is free in e_1 or e_2

Free and Bound Variables

$$FV(x) = \{x\} \tag{1}$$

$$FV(\lambda x\,.\,e) = FV(e) \backslash \{x\} \tag{2}$$

$$FV(e_1 e_2) = FV(e_1) \cup FV(e_2) \tag{3}$$

<u>Definition.</u> A variable x is **free** in e if it does not appear **bound** by a λ. Formally:

1. x is free in x

2. x is free in $\lambda y\,.\,e$ if it is free in e and $x \neq y$

3. x is free in $e_1 e_2$ if x is free in e_1 or e_2

<u>Definition.</u> A variable x is **free** in e if $x \in FV(e)$ as above

Definition (Third Attempt)

$$[v/y]x = \begin{cases} v & x = y \\ x & \texttt{else} \end{cases}$$

$$[v/y](\lambda x . e) = \begin{cases} \lambda x . e & x = y \\ \lambda z . [v/y][z/x]e & x \in FV(v) \\ \lambda x . [v/y]e & \texttt{else} \end{cases}$$

$$[v/y](e_1 e_2) = ([v/y]e_1)([v/y]e_2)$$

Since we're interested in α-equivalence, we can first *replace* the bound variable and *substitute* it in the body of the function. This is called **α-renaming**

Is there still a problem?

Problem Case III

$$FV(x) = \{x\}$$
$$FV(\lambda x\,.\,e) = FV(e) \backslash \{x\}$$
$$FV(e_1 e_2) = FV(e_1) \cup FV(e_2)$$

$$[v/y]x = \begin{cases} v & x = y \\ x & \texttt{else} \end{cases}$$
$$[v/y](\lambda x\,.\,e) = \begin{cases} \lambda x\,.\,e & x = y \\ \lambda z\,.\,[w/z][z/x]e & x \in FV(v) \\ \lambda x\,.\,[v/y]e & \texttt{else} \end{cases}$$
$$[v/y](e_1 e_2) = ([v/y]e_1)([v/y]e_2)$$

$$[x/y](\lambda x\,.\,xyz)$$

This isn't exactly a problem, but *we have to be careful about which variable to replace the bound variable x with*

If we choose z, then we capture a *different* variable!

"Correct" Definition

$$[v/y]x = \begin{cases} v & x = y \\ x & \texttt{else} \end{cases}$$

$$[v/y](\lambda x \,.\, e) = \begin{cases} \lambda x \,.\, e & x = y \\ \lambda z \,.\, [v/y][z/x]e & x \in FV(v), z \notin FV(e) \\ \lambda x \,.\, [v/y]e & \texttt{else} \end{cases}$$

$$[v/y](e_1 e_2) = ([v/y]e_1)([v/y]e_2)$$

Finally a definition, that works. Sort of...

The only problem with this definition is that it now poses an implementation issue. **How do we come up with z?**

In mathematics, we can say it's **always possible** to come up with a variable z, but when we're implementing a programming language, we need an *actual* procedure

Well-Scopedness and Closedness

open

$$\lambda x\,.\,y$$

closed

$$\lambda x\,.\,\lambda y\,.\,y$$

Definition. *(informal)* An expression e is **well-scoped** if every free variable in e is "in scope" (more on that on Thursday)

Definition. An expression e is **closed** if it has no free variables

Every closed term is well-scoped

One Solution: Well-Scopedness Check

$$[v/y]x = \begin{cases} v & x = y \\ x & \texttt{else} \end{cases}$$

$$[v/y](\lambda x\,.\,e) = \begin{cases} \lambda x\,.\,e & x = y \\ \lambda z\,.\,[v/y][z/x]e & x \in FV(v), z \notin FV(e) \\ \lambda x\,.\,[v/y]e & \texttt{else} \end{cases}$$

$$[v/y](e_1 e_2) = ([v/y]e_1)([v/y]e_2)$$

If we only work with closed (well-scoped) expressions, then we don't need to worry about captured variables

The condition requiring α-renaming never holds!

The Takeaway: In mini-project 1, you should check if the expression has a free variable *before* you evaluate it

demo

(lambda calculus)