CS 456

Programming Languages
Fall 2024

Week 2

Lambda-Calculus

A Digression …

- Property-Based Testing
  - Test a "property" of a program
  - Properties hold for class of inputs
    - Don't need to write tests one-by-one
  - Randomly generate testcases to check a property

Examples

- **Idempotence**: Applying a function twice same as applying it once
- **Equivalence**: Optimized function mirrors reference version
- **Well-formedness**:
  - A tree is a BST
  - A list is sorted
- **Relationships Between Functions**:
  - An inserted value is a member of a BST
  - Deleteing an value from a BST means it is no longer a member
  - **Inverse**: One function "undoes" another function

Property-Based Testing

Quickcheck: A library for PBT of OCaml programs

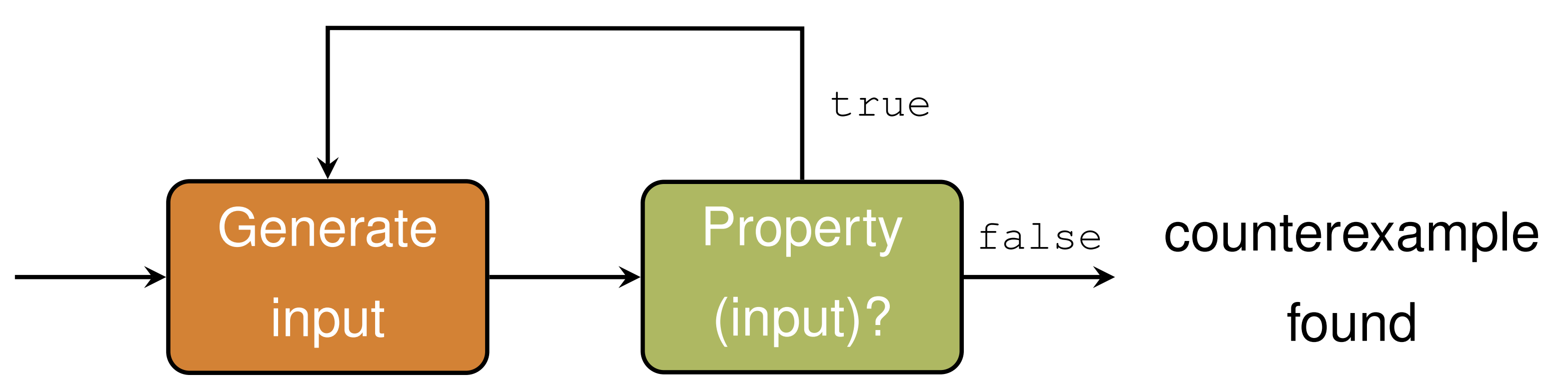
Basic Idea:

- Write a random input generator using Quickcheck provided functions
- Write properties of the program you would like to test

* Quickcheck will generate random inputs from the provided generator and check that the provided properties hold over those inputs
* When an input fails, Quickcheck will "shrink" it to find a minimal failing test case

Tests are described by

- a generator (delivering random input)
- a property (Boolean-valued function)



Property-Based Testing

- Test that list reverse is *involutive*: List.rev (List.rev l) == l for any list l.

create a generator

```
let test =
  QCheck.Test.make ~count:1000 ~name:"list_rev_is_involutive"
   QCheck.(list small_nat)
   (fun l -> List.rev (List.rev l) = l);;
```

generate random lists of small numbers

check the property holds for each such generated list

```
QCheck.Test.check_exn test;;

QCheck_runner.run_tests [test];;
```

Different mechanisms to run tests

Property-Based Testing


```
type tree = Leaf of int | Node of tree * tree

let leaf x = Leaf x
let node x y = Node (x,y)

let tree_gen = QCheck.Gen.(sized @@ fix
  (fun self n -> match n with
    | 0 -> map leaf nat
    | n ->
      frequency
        [1, map leaf nat;
         2, map2 node (self (n/2)) (self (n/2))]
    ));;
```

Generates a size and applies it to the generator returned by fix

Generate a natural number and supply it as an argument to leaf

Generate two subtrees and supply them as arguments to node

Defining a Language

A “recipe” for defining a language:

1. Syntax:
   - What are the valid expressions?
2. Semantics (Dynamic Semantics):
   - What is the meaning of valid expressions?
3. Sanity Checks (Static Semantics):
   - What expressions have meaningful evaluations?

Defining a Programming Language

1. Syntax

atexp	::=	*scon*	special constant
		⟨`op`⟩*longvid*	value identifier
		`{` ⟨*exprow*⟩ `}`	record
		`let` *dec* `in` *exp* `end`	local declaration
		`(` *exp* `)`	
exprow	::=	*lab* `=` *exp* ⟨ `,` *exprow*⟩	expression row
exp	::=	*atexp*	atomic
		exp atexp	application (L)
		exp_1 *vid* exp_2	infixed application
		exp `:` *ty*	typed (L)
		exp `handle` *match*	handle exception
		`raise` *exp*	raise exception
		`fn` *match*	function
match	::=	*mrule* ⟨ `|` *match*⟩	
mrule	::=	*pat* `=>` *exp*	
dec	::=	`val` *tyvarseq valbind*	value declaration
		`type` *typbind*	type declaration
		`datatype` *datbind*	datatype declaratic
		`datatype` *tycon* `-=-` `datatype` *longtycon*	datatype replicatio:
		`abstype` *datbind* `with` *dec* `end`	abstype declaratior
		`exception` *exbind*	exception declarati
		`local` dec_1 `in` dec_2 `end`	local declaration
		`open` $longstrid_1 \cdots longstrid_n$	open declaration (r
			empty declaration
		dec_1 ⟨;⟩ dec_2	sequential declarati
		`infix` ⟨*d*⟩ $vid_1 \cdots vid_n$	infix (L) directive
		`infixr` ⟨*d*⟩ $vid_1 \cdots vid_n$	infix (R) directive
		`nonfix` $vid_1 \cdots vid_n$	nonfix directive
valbind	::=	*pat* `=` *exp* ⟨`and` *valbind*⟩	
		`rec` *valbind*	
typbind	::=	*tyvarseq tycon* `=` *ty* ⟨`and` *typbind*⟩	
datbind	::=	*tyvarseq tycon* `=` *conbind* ⟨`and` *datbind*⟩	
conbind	::=	⟨`op`⟩*vid* ⟨`of` *ty*⟩ ⟨ `|` *conbind*⟩	
exbind	::=	⟨`op`⟩*vid* ⟨`of` *ty*⟩ ⟨`and` *exbind*⟩	
		⟨`op`⟩*vid* `=` ⟨`op`⟩*longvid* ⟨`and` *exbind*⟩	

Figure 4: Grammar: Expressions, Matches, Declarations and Bindings

2. Semantics

$$\frac{E \vdash atexp \Rightarrow v}{E \vdash atexp \Rightarrow v} \quad (96)$$

$$\frac{E \vdash exp \Rightarrow vid \qquad vid \neq \texttt{ref} \qquad E \vdash atexp \Rightarrow v}{E \vdash exp\ atexp \Rightarrow (vid, v)} \quad (97)$$

$$\frac{E \vdash exp \Rightarrow en \qquad E \vdash atexp \Rightarrow v}{E \vdash exp\ atexp \Rightarrow (en, v)} \quad (98)$$

$$\frac{s, E \vdash exp \Rightarrow \texttt{ref}\ , s' \qquad s', E \vdash atexp \Rightarrow v, s'' \qquad a \notin \mathrm{Dom}(mem \text{ of } s'')}{s, E \vdash exp\ atexp \Rightarrow a,\ s'' + \{a \mapsto v\}} \quad (99)$$

$$\frac{s, E \vdash exp \Rightarrow\ \texttt{:=}\ , s' \qquad s', E \vdash atexp \Rightarrow \{1 \mapsto a,\ 2 \mapsto v\}, s''}{s, E \vdash exp\ atexp \Rightarrow \{\} \text{ in Val},\ s'' + \{a \mapsto v\}} \quad (100)$$

$$\frac{E \vdash exp \Rightarrow b \qquad E \vdash atexp \Rightarrow v \qquad \mathrm{APPLY}(b, v) = v'/p}{E \vdash exp\ atexp \Rightarrow v'/p} \quad (101)$$

$$\frac{\begin{array}{c} E \vdash exp \Rightarrow (match, E', VE) \qquad E \vdash atexp \Rightarrow v \\ E' + \mathrm{Rec}\,VE,\ v \vdash match \Rightarrow v' \end{array}}{E \vdash exp\ atexp \Rightarrow v'} \quad (102)$$

$$\frac{\begin{array}{c} E \vdash exp \Rightarrow (match, E', VE) \qquad E \vdash atexp \Rightarrow v \\ E' + \mathrm{Rec}\,VE,\ v \vdash match \Rightarrow \mathrm{FAIL} \end{array}}{E \vdash exp\ atexp \Rightarrow [\texttt{Match}]} \quad (103)$$

$$\frac{E \vdash exp \Rightarrow v}{E \vdash exp\ \texttt{handle}\ match \Rightarrow v} \quad (104)$$

$$\frac{E \vdash exp \Rightarrow [e] \qquad E, e \vdash match \Rightarrow v}{E \vdash exp\ \texttt{handle}\ match \Rightarrow v} \quad (105)$$

$$\frac{E \vdash exp \Rightarrow [e] \qquad E, e \vdash match \Rightarrow \mathrm{FAIL}}{E \vdash exp\ \texttt{handle}\ match \Rightarrow [e]} \quad (106)$$

$$\frac{E \vdash exp \Rightarrow e}{E \vdash \texttt{raise}\ exp \Rightarrow [e]} \quad (107)$$

$$\frac{}{E \vdash \texttt{fn}\ match \Rightarrow (match, E, \{\})} \quad (108)$$

Lambda Calculus

- Lambda calculus was developed by Alonzo Church in the 30s
  - A core language in which *everything* is a function
- Syntax of Lambda terms:

```
t ::= x
    | λx.t
    | t t
```

Variable

Lambda abstraction

Application

Lambda Calculus


```
t ::= x
    | λx.t
    | t t

x ∈ Var
```

Identity function:

λx.x

Lambda Calculus

PLUS NUMBERS


```
t ::= x
    | λx.t
    | t t
    | n
    | t + t
x ∈ Var
n ∈ N
```

Identity function:

`λx.x`

Double function:

`λx.x + x`

Applying a function:

`(λx.x) 42`

Lambda Calculus

PLUS NUMBERS


```
t ::= x
    | λx.t
    | t t
    | n
    | t + t
x ∈ Var
n ∈ N
```

Identity function:

λx.x

Double function:

λx.x + x

Applying a function:

(λx.x) (λx.x)

Lambda Calculus

Plus Numbers


```
t ::= x
    | λx.t
    | t t
    | n
    | t + t
x ∈ Var
n ∈ N
```

Identity function:

`λx.x`

Double function:

`λx.x + x`

Applying a function:

`(λx.λy.x) (λx.x)`

Lambda Calculus

PLUS NUMBERS


```
t ::= x
    | λx.t
    | t t
    | n
    | t + t
x ∈ Var
n ∈ N
```

Identity function:

```
fun x -> x
```

Double function:

```
fun x -> x + x
```

Applying a function:

```
(fun x -> x) 42
```

Conventions


```
t ::= x
    | λx.t
    | t t
    | n
    | t + t
x ∈ Var
n ∈ N
```

- Application associates to the left:

  `s t u ≡ (s t) u`

- Group sequences of lambda abstractions:

  `λx y. x ≡ λx. λy. x`

- Bodies of abstraction extend as far to the right as possible:

  `λx y. x y x ≡`
  `λx.(λy.((x y) x))`

Variable Scopes


```
t ::= x
    | λx.t
    | t t
    | n
    | t + t
x ∈ Var
n ∈ N
```

1. A variable x is **bound** when it occurs in the body t of a lambda abstraction `λx.t`:

2. A variable x is **free** if it is not bound by an enclosing lambda expression:

3. A **closed** term has no free variables

Concept Check

What the **free** and **bound** variables in these terms?

- `λx.λy.y x z`
- `(λx.λy.y x) (5+2) λx.x+1`
- (λx.x) (λx.x y) (λz.(λy.y) z)

α-Equivalence


```
t ::= x
    | λx.t
    | t t
    | n
    | t + t
x ∈ Var
n ∈ N
```

1. Variables are bound to the closest enclosing lambda:
2. The name of bound variables is not important:
3. Expressions t_1 and t_2 that differ only in bound variable names are called **α-equivalent**

Concept Check

Which of these terms are **α-equivalent?**

```
(λx.x) ((λw.w) ((λz.(λy.y) z)) ≡α (λx.x) ((λx.x) ((λx.(λx.x) x))

(λx.λy.y x) (5+2) λx.x+1 ≡α (λq.λy.y q) (5+2) (λy.y+1)

(λx.λy.y x)(5+2)λx.x+1 ≡α ((λq.λy.y q)(5+2)) (λx.x+1)

(λx.λy.y x) (5+2) λx.x+1 ≡α(λx.λy.y x) 7 λx.x+1

(λx.λy.y x z)≡α (λa.λb.b c z)

(λy.λx.x y q)≡α (λx.λy.y x z)
```

Inference Rules

To describe the meaning of lambda-calculus expressions, we will use a notation called *inference (or reduction) rules*.

Informally, a rule of the form:

$$\frac{A_1, A_2, \ldots, A_n}{\mathtt{t}_1 \to \mathtt{t}_2}$$

reads:

Expression $\mathtt{t}_1$ evaluates to (or "reduces" to) $\mathtt{t}_2$
if the constraints defined by $A_1, A_2, \ldots, A_n$ hold

We'll delve into a more formal characterization of what these rules signify later in the course …

Semantics

REDUCTION RULES

$$\frac{\mathtt{t_1} \longrightarrow \mathtt{t_1'}}{\mathtt{t_1\ t_2} \longrightarrow \mathtt{t_1'\ t_2}}$$

$$\frac{\mathtt{value\ t_1} \qquad \mathtt{t_2} \longrightarrow \mathtt{t_2'}}{\mathtt{t_1\ t_2} \longrightarrow \mathtt{t_1\ t_2'}}$$

$$\frac{\mathtt{value\ t_2}}{\mathtt{(\lambda x.t_1)\ t_2} \longrightarrow \mathtt{[x{\coloneqq}t_2]t_1}}$$

This rule is called the **beta reduction rule**

Read $\mathtt{[x{\coloneqq}t_2]t_1}$ as "replace all free occurrences of $\mathtt{x}$ in $\mathtt{t_1}$ with $\mathtt{t_2}$"

VALUE RULES

$$\frac{}{\mathtt{value\ (\lambda x.t)}}$$

Semantics

PLUS NUMBERS

REDUCTION RULES

$$\frac{\texttt{value}\ t_1 \qquad t_2 \longrightarrow t_2'}{t_1\ t_2 \longrightarrow t_1\ t_2'}$$

$$\frac{t_2 \longrightarrow t_2'}{t_1 + t_2 \longrightarrow t_1 + t_2}$$

$$\frac{t_1 \longrightarrow t_1'}{t_1\ t_2 \longrightarrow t_1'\ t_2}$$

$$\frac{t_1 \longrightarrow t_1'}{t_1 + t_2 \longrightarrow t_1' + t_2}$$

$$\frac{\texttt{value}\ t_2}{(\lambda x.t_1)\ t_2 \longrightarrow [x{:=}t_2]t_1}$$

$$\frac{n \in \mathbb{Z} \qquad m \in \mathbb{Z}}{n + m \longrightarrow n +_{\mathbb{Z}} m}$$

VALUE RULES

$$\frac{}{\texttt{value}\ (\lambda x.t)}$$

$$\frac{n \in \mathbb{Z}}{\texttt{value}\ n}$$

Concept Review

$$(1 + 2) + (5 * 4) \longrightarrow$$

$$\frac{n \in \mathbb{Z} \qquad m \in \mathbb{Z}}{n+m \longrightarrow n +_{\mathbb{Z}} m} \text{ EADDCONST}$$

$$\frac{n \in \mathbb{Z} \qquad m \in \mathbb{Z}}{n*m \longrightarrow n *_{\mathbb{Z}} m} \text{ EMULCONST}$$

$$\frac{e_n \longrightarrow e_o}{e_n+e_m \longrightarrow e_o + e_m} \text{ EADDL}$$

$$\frac{e_n \longrightarrow e_o}{e_n*e_m \longrightarrow e_o*e_m} \text{ EMULL}$$

$$\frac{e_m \longrightarrow e_o}{e_n+e_m \longrightarrow e_n + e_o} \text{ EADDR}$$

$$\frac{e_m \longrightarrow e_o}{e_n*e_m \longrightarrow e_n*e_o} \text{ EMULR}$$

Substitution

Need to ensure that we don't inadvertently bind free variables!

$$[x{:=}s]x \equiv s$$

$$[x{:=}s]y \equiv y \quad \text{if } x \neq y$$

$$[x{:=}s]\lambda x.t \equiv \lambda x.t$$

$$[x{:=}s]\lambda y.t \equiv \lambda y.[x{:=}s]t \quad \text{where } x \neq y$$

$$[x{:=}s]t_1\ t_2 \equiv [x{:=}s]t_1\ [x{:=}s]t_2$$

Not sufficient when s is an open term

$$[x{:=}w](\lambda y.x) \equiv \lambda y.\ w$$

$$[x{:=}\lambda z.z\ w](\lambda y.x) \equiv \lambda y\ z.\ z\ w$$

$$[x{:=}y](\lambda x.x) \equiv \lambda x.x$$

$$[x{:=}w\ y\ z](\lambda z.x\ z) \equiv \lambda z.(w\ y\ z)\ z$$

$$[x{:=}w\ y\ z](\lambda z.x\ z) \not\equiv \lambda z.(w\ y\ z)\ z$$

$$\equiv_\alpha [x{:=}w\ y\ z](\lambda u.x\ u) \equiv \lambda u.(w\ y\ z)\ u$$

Semantics

$$\frac{t_1 \longrightarrow t_1'}{t_1\ t_2 \longrightarrow t_1'\ t_2} \qquad \frac{\text{value } t_1 \qquad t_2 \longrightarrow t_2'}{t_1\ t_2 \longrightarrow t_1\ t_2'}$$

$$\frac{\text{value } t_2}{(\lambda x.t_1)\ t_2 \longrightarrow [x \mapsto t_2]t_1}$$

β-redex

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

$(\lambda y.\ (\lambda z.\ z)\ y)\ (\lambda w.\ w) \longrightarrow$

$(\lambda z.\ z)\ (\lambda w.\ w) \longrightarrow$

$(\lambda w.\ w)$

A term with no redexes is said to be in **normal form**

Redexes are highlighted in blue

Example

$$\frac{\texttt{t}_1 \longrightarrow \texttt{t}_1'}{\texttt{t}_1\ \texttt{t}_2 \longrightarrow \texttt{t}_1'\ \texttt{t}_2} \qquad \frac{\texttt{value t}_1 \qquad \texttt{t}_2 \longrightarrow \texttt{t}_2'}{\texttt{t}_1\ \texttt{t}_2 \longrightarrow \texttt{t}_1\ \texttt{t}_2'}$$

$$\frac{\texttt{value t}_2}{(\lambda \texttt{x.t}_1)\ \texttt{t}_2 \longrightarrow [\texttt{x}{:=}\texttt{t}_2]\texttt{t}_1}$$

(λx.x) (λx. x (λt f. f) (λt f. t)) (λt f. t)

⟶ (λx. x (λt f. f) (λt f. t)) (λt f. t)

⟶ (λt f. t) (λt f. f) (λt f. t)

⟶ (λf. (λt f. f)) (λt f. t)

⟶ λt f. f

Concept Check

Identify any redexes in the following terms:

```
(λx.x) (λx.x)

λz.(λx.x) z

(λx.x) ((λy.y) (λz.(λx.x) z))

λx y. x y x
```

Evaluation Strategies

Call-By-Value
Aka Strict

Recall that lambda abstractions and numbers are values:

The lambda calculus' values are the functions:

$$\frac{}{\text{value } \lambda x.t}$$

This is called a *call-by-value* semantics: redexes are always the top-most function that is applied to a value:

$$\frac{t_1 \longrightarrow t_1'}{t_1\ t_2 \longrightarrow t_1'\ t_2} \qquad \frac{\text{value } t_1 \quad t_2 \longrightarrow t_2'}{t_1\ t_2 \longrightarrow t_1\ t_2'}$$

$$\frac{\text{value } t_2}{(\lambda x.t_1)\ t_2 \longrightarrow [x \coloneqq t_2]t_1}$$

Examples

Plus Numbers


```
(λx. x + x) ((λx. x + x) (5 + 3)) ⟶
    (λx. x + x)((λx. x + x) 8) ⟶
        (λx. x + x)(8 + 8) ⟶
         ((λx. x + x) 16) ⟶
              16 + 16 ⟶
                  32
```

```
(λx.λy.y x)(5+2) λx.x+1
⟶(λx.λy.y x) 7 λx.x+1
⟶(λy. y 7) λx.x+1
⟶(λx. x+1) 7
⟶ 7+1
⟶ 8
```

Normalization

- If *every* program in a language is guaranteed to always evaluate to a normal term, we say the language is *strongly normalizing*.
  - Formally:
  - **Statement of Strong Normalization:**
  - For any term `t`, all sequences of reduction steps starting from `t` eventually reaches a normal form `t'`.
- Every program in a strongly normalizing language terminates.

Ω

- Is the lambda calculus strongly normalizing under beta reduction?
  - Does every expression eventually evaluate to a normal form?
  - No!

This is a diverging computation, i.e. one that does not terminate
We'll call this Ω

$$\Omega \equiv (\lambda\ \mathrm{x.}\ (\mathrm{x\ x}))(\lambda\ \mathrm{x.}\ (\mathrm{x\ x}))$$

Evaluation Strategies

CALL-BY-NAME
AKA LAZY

An alternative: beta-reductions are performed as soon as possible:

$$\frac{}{(\lambda x.t_1)\ t_2 \longrightarrow [x \coloneqq t_2]t_1} \qquad \frac{t_1 \longrightarrow t_1'}{t_1\ t_2 \longrightarrow t_1'\ t_2}$$

```
(λx.λy.y x)(5+2)λx.x+1
⟶(λy.y (5+2)) λx.x+1
⟶(λx.x+1) (5+2)
⟶(5 + 2) + 1
⟶ 7 + 1
⟶ 8
```

```
(λf.f 7)((λx.x x) λy.y)
⟶ ((λy. y) (λy. y)) 7
⟶ (λy. y) 7
⟶ 7
```

term duplicated!

Evaluation Strategies

Call-By-Name

$(\lambda x.x + x)(5 + 6)$
$\longrightarrow (5 + 6) + (5 + 6)$
$\longrightarrow 11 + (5 + 6)$
$\longrightarrow 11 + 11$
$\longrightarrow 22$

Laziness can lead to duplicated work!

Call-By-Value

$(\lambda x\ y.x + x)\ 5\ (5 + 6)$
$\longrightarrow (\lambda y.5 + 5)\ (5 + 6)$
$\longrightarrow (\lambda y.5 + 5)\ 11$
$\longrightarrow 5 + 5$
$\longrightarrow 10$

Strictness can lead to unnecessary work!

Concept Check

Evaluate this expression using both CBV and CBN strategies:

```
(λx.x) ((λy.y) (λz.(λx.x) z))
```

(Recall application is left-associative)

Eta-reduction

One common additional reduction rule is called **eta reduction**:

$$\frac{\texttt{x does not appear in t}}{(\lambda\texttt{x.t x}) \longrightarrow \texttt{t}}$$

Captures the idea that λx. (λy.y x) and λy.y are equivalent

Expressivity

Church's Thesis (1935): Informally, any function on the natural numbers that can be effectively computed (i.e., can be expressed as an algorithm) can be computed using the λ-calculus. In other words, λ-calculus is equivalent in its expressive power to Turing Machines.

- This property holds for the pure λ-calculus, i.e., the calculus without primitive support for numbers!
- This means that function abstraction and application are sufficiently powerful to model numbers and their operations.

Booleans


```
true  ≡  λ t. λ f. t
false ≡  λ t. λ f. f

(true v w)  ≡  ((λ t.λ f. t) v) w)
                     ((λ f. v) w)
                     v
(false v w) ≡  ((λ t.λ f. f) v) w)
                      ((λ f. f) w)
                      w
```

Booleans

- `not` $\equiv \lambda$ `b. b true false`

The function that returns true if b is false, and false if b Is true.

- `and` $\equiv \lambda$ `b.` λ `c. b c false`

The function that given two Boolean values (v and w) returns w if v is true and false if v is false. Thus, (and v w) yields true only if both v and w are true.

Church Numerals

There are no explicit operations to manipulate numbers

Encode numbers using higher-order functions:

- zero $\equiv$ `λ s. λ z. z`
- one $\equiv$ `λ s. λ z. (s z)`
- two $\equiv$ `λ s. λ z. (s (s z))`

Read “s” as successor and “z” as zero

Church Numerals

- `succ     λ n. λ s. λ z . s (n s z)`

  A function that takes s and z and applies s repeatedly to z.

- `plus     λ m. λ n. λ s. λ z . m s (n s z)`

  takes two Church numerals and yields another Church numeral that given s and z applies s iterated n times to z and then applies s iterated m times to the result.

```
plus one two succ zero ->

one succ (two succ zero) ->

succ (two succ zero) ->

succ (succ (succ zero)) ->

3
```

Naming and substitution

Although we claimed that lambda calculus essentially manipulates functions (it does), we've spent a lot of time thinking about variables

- substitutions
- free variables
- equivalence upto renaming

Implementations must consider these issues seriously

- Rename bound variables when performing substitutions with "fresh" names.
- Impose a condition that all bound variables be distinct from each other, and other free variables.
- Derive a canonical representation that does not require renaming at all.

Terms and Contexts

De Brujin indices:

- Have variable occurrences "point" directly to their binders rather than referring to them by name.

- Do so by replacing variable occurrences with numbers:

number k stands for "the variable bound by the kth enclosing λ-term

Example: `λ x. λ y. x  (y x) ≡ λ . λ . 1 (0 1)`

Similar to static offsets in an activation record or display.

Examples


```
identity ≡ λ x. x ≡ λ .0
true ≡ λ x. λ y. x ≡ λ.λ. 1
false ≡ λ x. λ y. y ≡ λ.λ. 0
two ≡ λ s. λ z. s (s z) ≡ λ . λ . (1 (1 0))
```

Contexts

How do we replace free variables with their binders?

- Assume an ordered context listing all free variables that can occur, and map free variables to their index in this context (counting right to left)

Context: a, b

```
a ↦ 1, b ↦ 0
     λ x. a  ≡ λ . 2
     λ x. b ≡ λ. 1
     λ x.b (λ y. a) ≡ λ.1(λ.3)
```

Shifting and substitution

When substituting into a λ term, indices must be adjusted:

`λ y. x [ z/x]` in context `x,y,z`

`[2 ↦ 0] λ. 2 ≡ λ . [3 ↦ 1] 3 ≡ λ .1`

Key point: context becomes longer when substituting inside an abstraction. Need to be careful to adjust free variables, not bound ones.

```
shift(d,c)(k) = k if k < c
                k + d if k ≥ c
shift(d,c)(λ.t) = (λ.shift(d,c+1)(t))
shift(d,c)(t1 t2) = (shift(d,c)(t1))(shift(d,c)(t2))
```

Here c is a cutoff and d is the shift amount

shift(d,c) thus shifts the indices of free variables equal to or above cutoff c by d

Example

shift(2,0)(λ.λ. 1 (0 2))

λ.λ. 1 (0 4)

shift(2,0)(λ. 0 1 (λ. 0 1 2))

λ. 0 3 (λ. 0 1 4)

Substitution


```
[j ↦ s] k = s if k = j
            k otherwise
```

```
[j ↦ s](λ .t ) = λ. [j+1 ↦ shift(1,0)s] t
```

[j ↦ s](t_1 t_2) = ([j ↦ s) t_1) ([j ↦ s] t_2)

Beta-reduction:

```
(λ t) v → shift(-1,0)([0 ↦ shift(1,0)(v)] t)
```

Examples

Assume context <a,b> Then, a ↦ 1, b ↦ 0

```
[a / b] b λ x. λ y. b
[0 ↦ 1] 0 λ . λ . 2
1 λ . λ . 3 ≡  a λ x. λ y. a
```

```
[(a (λ z. a)) / b] (b (λ x.b))
[0 ↦ (1 (λ. 2))] (0 λ. 1)
1 (λ . 2) (λ . (2 (λ . 3)))
(a (λ z. a)) (λ x. (a (λ z. a)))
```

Examples

[a/b] (λ b. (b a))

[0 ↦ 1] (λ . (0 2))

(λ . (0 2))

(λ b. (b a))

[a/b] (λ a. (b a))

[0 ↦ 1] λ . (1 0)

λ . (2 0)

(λ a′. a a′)