Introduction to Lambda Calculus

Lecture 4
ECS 240

Plan

- Introduce lambda calculus
  - Syntax and operational semantics
  - Properties
- Relationship to programming languages (later)
- Study of types and type systems (even later)

Background

- Developed in 1930's by Alonzo Church
- Subsequently studied by many people
- "Testbed" for procedural and functional languages
  - Simple
  - Powerful
  - Easy to extend with features of interest
  - Plays similar role for PL research as Turing machines for computability

"*Whatever the next 700 languages turn out to be, they will surely be variants of lambda calculus.*"

(Landin ' 66)

Syntax

- The λ-calculus has three kinds of expressions (terms)

$e ::= x$		Variables
$\mid$	$\lambda x.e$	Functions (abstraction)
$\mid$	$e_1\ e_2$	Application

- $\lambda x.e$ is a <u>one-argument function</u> with body e
- $e_1\ e_2$ is a function application
- Application associates to the left

  $x\ y\ z$ means $(x\ y)\ z$

- Abstraction extends to the right as far as possible

  $\lambda x.x \lambda y.x\ y\ z$ means $\lambda x.(x\ (\lambda y.\ ((x\ y)\ z)))$

Examples of Lambda Expressions

- The identity function:

  $I =_{def} \lambda x.\ x$

- A function that given an argument y discards it and yields the identity function:

  $\lambda y.\ (\lambda x.\ x)$

- A function that given a function f invokes it on the identity function

  $\lambda f.\ f\ (\lambda x.\ x)$

Scope of Variables

- As in all languages with variables it is important to discuss the notion of scope
  - Recall: the scope of an identifier is the portion of a program where the identifier is accessible
- An abstraction $\lambda x.\ E$ binds variable x in E
  - x is the newly introduced variable
  - E is the scope of x
  - We say x is bound in $\lambda x.\ E$
  - Just like formal function arguments are bound in the function body

Free and Bound Variables

- A variable is said to be free in E if it has occurrences that are not bound in E
- We can define the free variables of an expression E recursively as follows:

$$\text{Free}(x) = \{ x \}$$

$$\text{Free}(E_1\ E_2) = \text{Free}(E_1) \cup \text{Free}(E_2)$$

$$\text{Free}(\lambda x.\ E) = \text{Free}(E) - \{ x \}$$

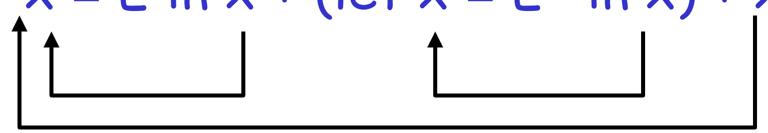
- Example: $\text{Free}(\lambda x.\ x\ (\lambda y.\ x\ y\ z)) = \{ z \}$

- Free variables are (implicitly or explicitly) declared outside the term

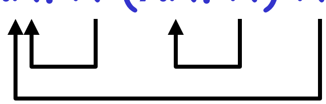
Free and Bound Variables (Cont.)

- Like in any language with statically nested scoping, we need to worry about variable shadowing (or capturing)
  - An occurrence of a variable might refer to different things in different contexts

- E.g., in IMP with locals: let x = E in x + (let x = E' in x) + x



- In λ-calculus: λx. x (λx. x) x



Renaming Bound Variables

- λ-terms that can be obtained from one another by renaming of the bound variables are considered identical. This is called <u>α-equivalence</u>.
- Example: $\lambda x.\ x$ is identical to $\lambda y.\ y$ and to $\lambda z.\ z$
- Intuition:
  - By changing the name of a formal argument and of all its occurrences in the function body, the behavior of the function does not change
  - In λ-calculus such functions are considered identical

Renaming Bound Variables (Cont.)

- Convention: we will always try to rename bound variables so that they are all unique
  - e.g., write $\lambda x.\ x\ (\lambda y.y)\ x$ instead of $\lambda x.\ x\ (\lambda x.x)\ x$
- This makes it easy to see the scope of bindings
- And also prevents confusion !

Substitution

- The substitution of E' for x in E (written $[E'/x]E$)
  - Step 1. Rename bound variables in E and E' so they are unique
  - Step 2. Perform the textual substitution of E' for x in E

- Example: $[y\ (\lambda x.\ x)\ /\ x]\ \lambda y.\ (\lambda x.\ x)\ y\ x$

  - After renaming: $[y\ (\lambda v.\ v)/x]\ \lambda z.\ (\lambda u.\ u)\ z\ x$

  - After substitution: $\lambda z.\ (\lambda u.\ u)\ z\ (y\ (\lambda v.\ v))$

- If we are not careful with scopes might get:

  $\lambda y.\ (\lambda x.\ x)\ y\ (y\ (\lambda x.\ x))$

Informal Semantics

- We consider only closed terms
- The evaluation of

$$(\lambda x.\ e)\ e'$$

1. Binds x to e'
2. Evaluates e with the new binding
3. Yields the result of this evaluation

- Like a function call, or like "let x = e' in e"

- Example:

  $(\lambda f.\ f\ (f\ e))\ g$ evaluates to $g\ (g\ e)$

Operational Semantics

- There exist many operational semantics for the λ-calculus
- All are based on the equation

$$(\lambda x.\ e)\ e' =_\beta [e'/x]e$$

usually oriented from left to right

- This is called the β-rule and the evaluation step a β-reduction

- The subterm $(\lambda x.\ e)\ e'$ is a β-redex

- $e \rightarrow_\beta e'$: e β-reduces to e' in one step
- $e \rightarrow_\beta^* e'$: e β-reduces to e' in 0 or more steps

Examples of Evaluation

- The identity function:

$$(\lambda x.\ x)\ E \rightarrow [E\ /\ x]\ x = E$$

- Another example with the identity:

$$(\lambda f.\ f\ (\lambda x.\ x))\ (\lambda x.\ x) \rightarrow$$
$$[\lambda x.\ x\ /\ f]\ f\ (\lambda x.\ x)) = [(\lambda x.\ x)\ /\ f]\ f\ (\lambda y.\ y)) =$$
$$(\lambda x.\ x)\ (\lambda y.\ y) \rightarrow$$
$$[\lambda y.\ y\ /x]\ x = \lambda y.\ y$$

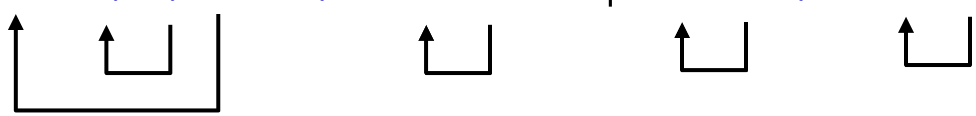
- A non-terminating evaluation:

$$(\lambda x.\ xx)(\lambda y.\ yy) \rightarrow$$
$$[\lambda y.\ yy\ /\ x]xx = (\lambda y.\ yy)(\lambda y.\ yy) \rightarrow\ ...$$

Evaluation and the Static Scope

- The definition of substitution guarantees that evaluation respects static scoping:

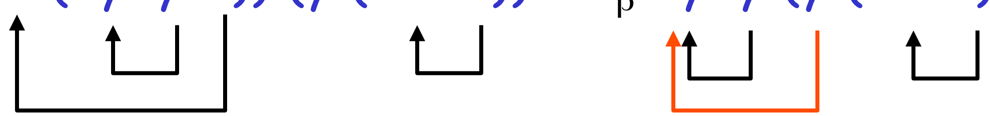
$$(\lambda\ x.\ (\lambda y.\ y\ x))\ (y\ (\lambda x.\ x)) \to_{\beta} \lambda z.\ z\ (y\ (\lambda v.\ v))$$



(y remains free, i.e., defined externally)

- If we forget to rename the bound y:

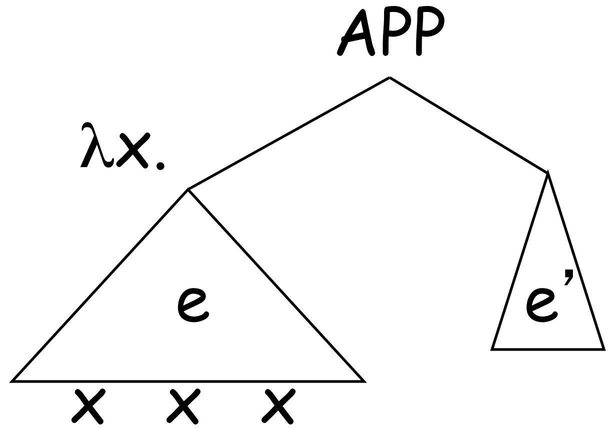
$$(\lambda\ x.\ (\lambda y.\ y\ x))\ (y\ (\lambda x.\ x)) \to^{*}_{\beta} \lambda y.\ y\ (y\ (\lambda v.\ v))$$



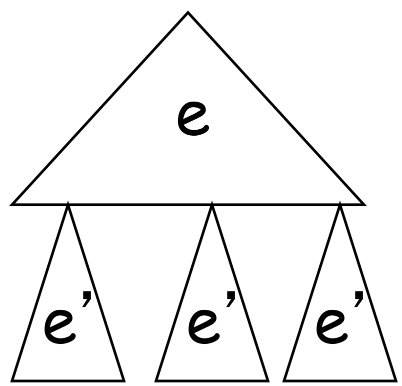
(y was free before but is bound now)

Another View of Reduction

- The application



- becomes:



Terms can "grow" substantially through β-reduction !

Normal Forms

- A term without redexes is in normal form
- A reduction sequence stops at a normal form
- If e is in normal form, then $e \rightarrow^*_\beta e'$ implies $e = e'$
- Examples
  - $\lambda x.\lambda y.\ x$ (normal form)
  - $(\lambda x.\lambda y.\ x)\ (\lambda x.\ x)$ (not normal form)

Nondeterministic Evaluation

- Define a small-step reduction relation

$$\frac{}{(\lambda x.\ e)\ e' \to [e'/x]e}$$

$$\frac{e_1 \to e_1'}{e_1\ e_2 \to e_1'\ e_2} \qquad \frac{e_2 \to e_2'}{e_1\ e_2 \to e_1\ e_2'}$$

$$\frac{e \to e'}{\lambda x.\ e \to \lambda x.\ e'}$$

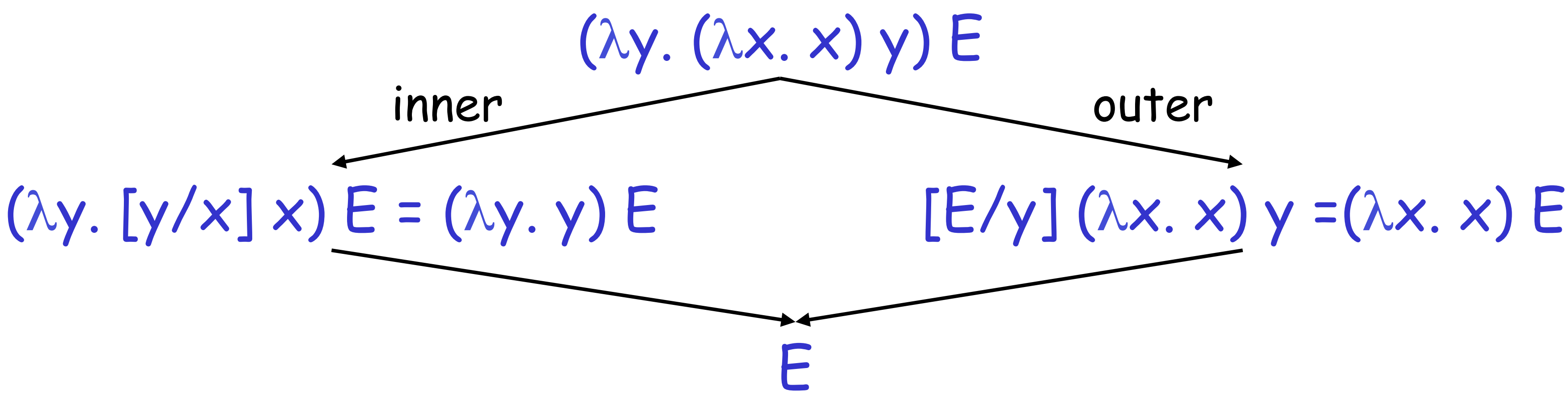
- Note
  - This is a non-deterministic semantics
  - We evaluate under λ

The Order of Evaluation

- A λ-term can have more than one instances of $(\lambda x.\ E)\ E'$

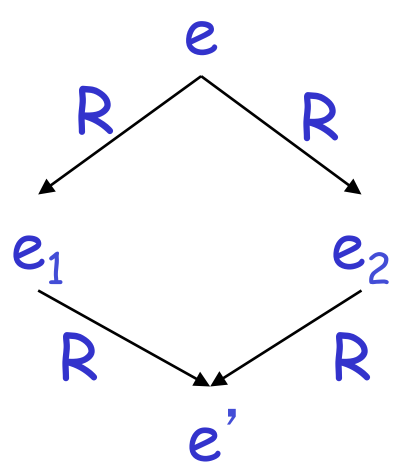
$$(\lambda y.\ (\lambda x.\ x)\ y)\ E$$

  - A choice: reduce the inner or the outer λ
  - Which one should we pick?



The Diamond Property

- A relation R has the diamond property iff
  - e R e_1 and e R e_2 implies there exists e' with e_1 R e' and e_2 R e'



- $\rightarrow_\beta$ does not have the diamond property
- $\rightarrow_\beta^*$ has the diamond property
- The simplest known proof is quite technical

The Diamond Property

- Languages defined by non-deterministic sets of rules are common
  - Logic programming languages
  - Expert systems
  - Constraint satisfaction systems

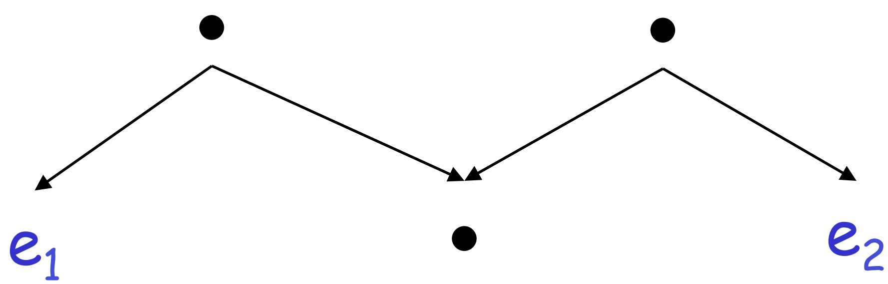
- It is useful to know whether such systems have the diamond property

Equality

- Let $=_\beta$ be the reflexive, transitive and symmetric closure of $\rightarrow_\beta$

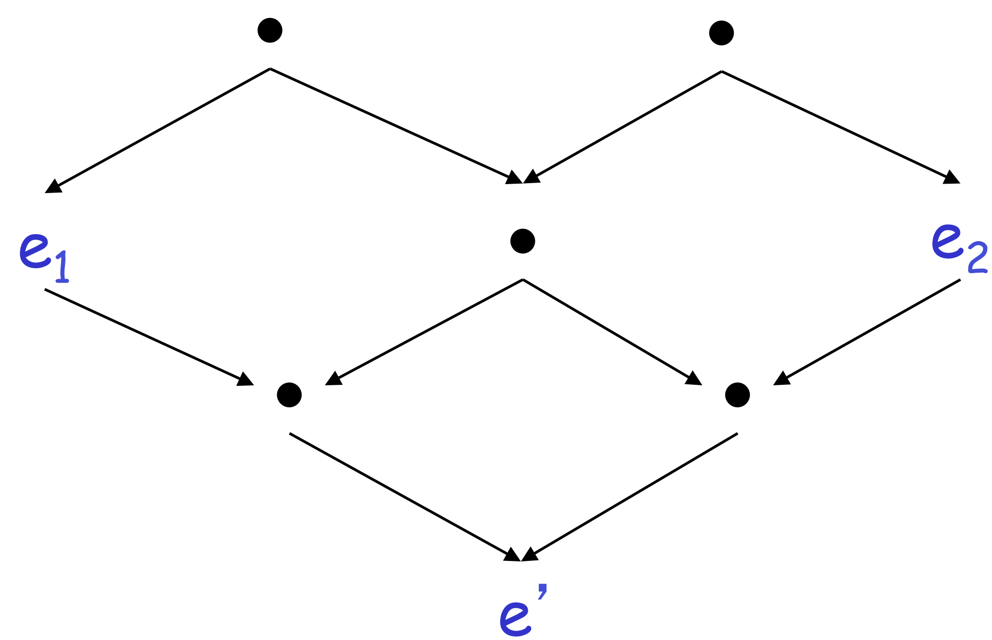
$$=_\beta \text{ is } (\rightarrow_\beta \cup \leftarrow_\beta)^*$$

- In another words, $e_1 =_\beta e_2$ if e_1 converts to e_2 via a sequence of forward and backward $\rightarrow_\beta$



The Church-Rosser Theorem

- If $e_1 =_\beta e_2$ then there exists e' such that $e_1 \rightarrow_\beta^* e'$ and $e_2 \rightarrow_\beta^* e'$



- Proof (informal): apply the diamond property as many times as necessary

Corollaries

- If $e_1 =_\beta e_2$ and e_1 and e_2 are normal forms then e_1 is identical to e_2
  - From CR we have $\exists e' .\ e_1 \rightarrow^*_\beta e'$ and $e_2 \rightarrow^*_\beta e'$
  - Since e_1 and e_2 are normal forms they are identical to e'

- If $e \rightarrow^*_\beta e_1$ and $e \rightarrow^*_\beta e_2$ and e_1 and e_2 are normal forms then e_1 is identical to e_2
  - All terms have a unique normal form

Evaluation Strategies

- Church-Rosser theorem says that independent of the reduction strategy we will not find more than one normal form

- But some reduction strategies might fail to find a normal form
  - $(\lambda x.\ y)\ \underline{((\lambda y.y\ y)\ (\lambda y.y\ y))} \to (\lambda x.\ y)\ \underline{((\lambda y.y\ y)\ (\lambda y.y\ y))} \to \dots$
  - $\underline{(\lambda x.\ y)\ ((\lambda y.y\ y)\ (\lambda y.y\ y))} \to y$

- We will consider three strategies
  - normal order
  - call-by-name
  - call-by-value

Normal-Order Reduction

- A redex is <u>outermost</u> if it is not contained inside another redex.
- Use K = λx.λy.x
  S = λf.λg.λx.f x (g x)
- Example: S (K x y) (K u v)
- K x, K u and S (K x y) are all redexes
- Both K u and S (K x y) are outermost
- Normal order always reduces the *leftmost outermost* redex first
- Theorem: If e has a normal form e' then normal order reduction will reduce e to e'

Why Not Normal Order ?

- In most (all?) programming languages, functions are considered values (fully evaluated)

- Example
  - $\lambda x.\ D\ D = \perp$ (with normal order)
  - where $D = (\lambda x.\ x\ x)$

- Thus, no reduction is done under lambda

- No popular programming language uses normal order

Call-by-Name

- **Don't** reduce under λ
- **Don't** evaluate the argument to a function call
- A value is an abstraction

$$\frac{}{\lambda x.\ e \to_n^* \lambda x.\ e} \qquad \frac{e_1 \to_n^* \lambda x.\ e_1' \quad [e_2/x]e_1' \to_n^* e}{e_1\ e_2 \to_n^* e}$$

- Call-by-name is demand-driven: an expression is not evaluated unless needed
- It is <u>normalizing</u>: converges whenever normal order converges
- Call-by-name does not necessarily evaluate to a normal form. Example: $D\ D = (\lambda x.\ x\ x)\ (\lambda x.\ x\ x)$

Call by Name

- Example:

  $(\lambda y.\ (\lambda x.\ x)\ y)\ ((\lambda u.\ u)\ (\lambda v.\ v)) \rightarrow_{\beta n}$

  $(\lambda x.\ x)\ ((\lambda u.\ u)\ (\lambda v.\ v)) \rightarrow_{\beta n}$

  $(\lambda u.\ u)\ (\lambda v.\ v) \rightarrow_{\beta n}$

  $\lambda v.\ v$

Call-by-Value Evaluation

- **Don't** reduce under lambda
- **Do** evaluate the arguments to a function call
- A value is an abstraction

$$\frac{}{\lambda x.\ e \to_v^* \lambda x.\ e} \qquad \frac{e_1 \to_v^* \lambda x.\ e_1' \quad e_2 \to_v^* e_2' \quad [e'_2/x]e_1' \to_v^* e}{e_1\ e_2 \to_v^* e}$$

- Most languages are primarily call-by-value
- But CBV is not normalizing: $(\lambda x.\ I)\ (D\ D)$
- CBV diverges more often than normal order and CBN

Call by Value

- Example:

  $(\lambda y.\ (\lambda x.\ x)\ y)\ ((\lambda u.\ u)\ (\lambda v.\ v)) \rightarrow_{\beta v}$

  $(\lambda y.\ (\lambda x.\ x)\ y)\ (\lambda v.\ v) \rightarrow_{\beta v}$

  $(\lambda x.\ x)\ (\lambda v.\ v) \rightarrow_{\beta v}$

  $\lambda v.\ v$

Considerations

- Call-by-value:
  - easy to implement
  - well-behaved (predictable) with respect to side-effects
- Call-by-name:
  - More difficult to implement (must pass unevaluated expressions)
  - The order of evaluation is harder to predict (e.g., difficulty with side-effects)
  - Has a simpler theory than call-by-value
  - Allows the natural expression of infinite data structures (e.g. streams)
  - Terminates more often than call-by-value
- Various other (not as common) evaluation strategies

Functional Programming

- The λ-calculus is a prototypical functional language with:
  - no side effects
  - several evaluation strategies
  - lots of functions
  - nothing but functions (pure λ-calculus does not have any other data type)

- How can we program with functions?
- How can we program with only functions?

Programming With Functions

- Functional programming style is a programming style that relies on lots of functions
- A typical functional paradigm is using functions as arguments or results of other functions
  - Higher-order programming
- Some “impure” functional languages permit side-effects (e.g., Lisp, ML)
  - references (pointers), in-place update, arrays, exceptions

Variables in Functional Languages

- We can introduce new variables:

  let $x = e_1$ in e_2

  - x is bound by let
  - x is statically scoped in e_2

- This is pretty much like $(\lambda x.\ e_2)\ e_1$
- In a functional language, variables are never updated
  - they are just names for expressions or values
  - E.g., x is a name for the value denoted by e_1 in e_2

- This models the meaning of “let” in math

Referential Transparency

- In "pure" functional programs, we can reason equationally, by substitution

  $$\text{let } x = e_1 \text{ in } e_2 \quad \equiv [e_1/x]e_2$$

- In an imperative language a "side-effect" in e_1 might invalidate the above equation

- The behavior of a function in a "pure" functional language depends only on the actual arguments
  - Just like a function in math
  - This makes it easier to understand and to reason about functional programs

Expressiveness of λ-Calculus

- The λ-calculus is a minimal system but can express
  - data types (integers, booleans, lists, trees, etc.)
  - branching
  - recursion
- This is enough to encode Turing machines
- Corollary: $e =_\beta e'$ is undecidable
- Still, how do we encode all these constructs using only functions?
- Idea: encode the "behavior" of values and not their structure

Encoding Booleans in Lambda Calculus

- What can we do with a boolean?
  - we can make a binary choice
- A boolean is a function that *given* two choices selects one of them
  - true $=_{def}$ $\lambda x.\ \lambda y.\ x$
  - false $=_{def}$ $\lambda x.\ \lambda y.\ y$
  - if E_1 then E_2 else E_3 $=_{def}$ $E_1\ E_2\ E_3$
- Example: "if true then u else v" is

  $(\lambda x.\ \lambda y.\ x)\ u\ v \rightarrow_{\beta} (\lambda y.\ u)\ v \rightarrow_{\beta} u$

Encoding Pairs in Lambda Calculus

- What can we do with a pair?
  - we can select one of its elements
- A pair is a function that given a boolean returns the left or the right element

  mkpair x y $=_{def}$ λb. b x y

  fst p $=_{def}$ p true

  snd p $=_{def}$ p false
- Example:

  fst (mkpair x y) → (mkpair x y) true → true x y → x

Encoding Natural Numbers in Lambda Calculus

- What can we do with a natural number?
  - we can iterate a number of times over some function
- A natural number is a function that given an operation f and a starting value s, applies f a number of times to s:

  $0 =_{def} \lambda f.\ \lambda s.\ s$

  $1 =_{def} \lambda f.\ \lambda s.\ f\ s$

  $2 =_{def} \lambda f.\ \lambda s.\ f\ (f\ s)$

  and so on

- These are numerals in unary representation
  - Also called Church numerals

Computing with Natural Numbers

- The successor function

  succ n $=_{def}$ $\lambda f.\ \lambda s.\ f\ (n\ f\ s)$

  or succ n $=_{def}$ $\lambda f.\lambda s.n\ f\ (f\ s)$

- Addition

  add n_1 n_2 $=_{def}$ n_1 succ n_2

- Multiplication

  mult n_1 n_2 $=_{def}$ n_1 (add n_2) 0

- Testing equality with 0

  iszero n $=_{def}$ n (λb. false) true

Computing with Natural Numbers. Example

succ n = λ f. λ s. f (n f s)
add n1 n2 = n1 succ n2
mult n1 n2 = n1 (add n2) 0

mult 2 2 →
2 (add 2) 0 →
(add 2) ((add 2) 0) →
2 succ (add 2 0) →
2 succ (2 succ 0) →
succ (succ (succ (succ 0))) →
succ (succ (succ (λf. λs. f (0 f s)))) →
succ (succ (succ (λf. λs. f s))) →
succ (succ (λg. λy. g ((λf. λs. f s) g y)))
succ (succ (λg. λy. g (g y))) $\rightarrow^*$ λg. λy. g (g (g (g y))) = 4

Computing with Natural Numbers. Example

- What is the result of the application add 0 ?

  $(\lambda n_1.\ \lambda n_2.\ n_1\ \text{succ}\ n_2)\ 0 \ \rightarrow_\beta$

  $\lambda n_2.\ 0\ \text{succ}\ n_2 =$

  $\lambda n_2.\ (\lambda f.\ \lambda s.\ s)\ \text{succ}\ n_2 \rightarrow_\beta$

  $\lambda n_2.\ n_2 =$

  $\lambda x.\ x$

- By computing with functions we can express some optimizations
  - But we need to reduce under the lambda

Encoding Recursion

- Given a predicate P encode the function "find" such that "find P n" is the smallest natural number which is larger than n and satisfies P
  - with find we can encode all recursion
- "find" satisfies the equation

  find p n = if p n then n else find p (succ n)
- Define

  $F = \lambda f.\lambda p.\lambda n.(p\ n)\ n\ (f\ p\ (succ\ n))$
- We need a fixed point of F

  find = F find

  or

  find p n = F find p n

The Fixed-Point Combinator

- Let $Y = \lambda F.\ (\lambda y.F(y\ y))\ (\lambda x.\ F(x\ x))$
  - This is called the fixed-point combinator
  - Verify that $Y\ F$ is a fixed point of F

    $$Y\ F \to_\beta (\lambda y.F\ (y\ y))\ (\lambda x.\ F\ (x\ x)) \to_\beta F\ (Y\ F)$$
  - Thus $Y\ F =_\beta F\ (Y\ F)$
- Given any function in λ-calculus we can compute its fixed-point
- Thus we can define "find" as the fixed-point of the function from the previous slide
- The essence of recursion is the self-application "$y\ y$"

Expressiveness of Lambda Calculus

- Encodings are fun
- But programming in pure λ-calculus is painful
- We will add constants (0, 1, 2, …, true, false, if-then-else, etc.)
- And we will add <u>types</u> (later!)