CENG 3549 – Functional Programming

Simply Typed λ-Calculus ($\lambda^{\to}$)

Burak Ekici

October 20 – November 24, 2022

Outline

Church's Thesis

A function is said to be effectively computable if it could be computed in a finite amount of time using finite resources

Church's Thesis

A function is said to be effectively computable if it could be computed in a finite amount of time using finite resources

- Effectively computable functions are just those could be computed by a Turing Machine

  $$\Longleftrightarrow$$

  Effectively computable functions are just those definable in the lambda calculus

Church's Thesis

A function is said to be effectively computable if it could be computed in a finite amount of time using finite resources

- Effectively computable functions are just those could be computed by a Turing Machine

  $\Longleftrightarrow$

  Effectively computable functions are just those definable in the lambda calculus

- Effectively computable is an intuitive notion not a mathematical one: Church's thesis cannot be proven
- Only refutable – by counterexample: give a function that could be computed with some model but not with a Turing machine

Uncomputability

A problem that cannot be solved by any Turing machine in finite time (or any equivalent formalism) is called uncomputable

Uncomputability

A problem that cannot be solved by any Turing machine in finite time (or any equivalent formalism) is called uncomputable

- The Halting Problem: given an arbitrary Turing machine and its input tape, will the machine eventually halt?

Uncomputability

A problem that cannot be solved by any Turing machine in finite time (or any equivalent formalism) is called uncomputable

- The Halting Problem: given an arbitrary Turing machine and its input tape, will the machine eventually halt?
- The Halting Problem is provably uncomputable – which means that it cannot be solved in practice.

What is a Function? – Extensional View

- Functions as graphs

What is a Function? – Extensional View

- Functions as graphs
  - each function f has a fixed domain X and a co-domain Y
  - each function $f : X \to Y$ is a set of pairs $f \subseteq X \times Y$ such that for each $x \in X$, there exists exactly one $y \in Y$ such that $(x, y) \in f$

What is a Function? – Extensional View

- Functions as graphs
  - each function f has a fixed domain X and a co-domain Y
  - each function $f : X \to Y$ is a set of pairs $f \subseteq X \times Y$ such that for each $x \in X$, there exists exactly one $y \in Y$ such that $(x, y) \in f$
- Equality of functions

What is a Function? – Extensional View

- Functions as graphs
  - each function f has a fixed domain X and a co-domain Y
  - each function $f : X \to Y$ is a set of pairs $f \subseteq X \times Y$ such that for each $x \in X$, there exists exactly one $y \in Y$ such that $(x, y) \in f$
- Equality of functions
  - two functions are said to be equal, for each input they yield the same output:

What is a Function? – Extensional View

- Functions as graphs
  - each function f has a fixed domain X and a co-domain Y
  - each function $f : X \to Y$ is a set of pairs $f \subseteq X \times Y$ such that for each $x \in X$, there exists exactly one $y \in Y$ such that $(x, y) \in f$
- Equality of functions
  - two functions are said to be equal, for each input they yield the same output:

$$f, g : X \to Y, \quad f = g \iff \forall x \in X, f(x) = g(x)$$

What is a Function? – Intensional View

- A function $f : A \to B$ is an abstraction $\lambda x.e$, where x is a variable name, and e is an expression, such that when a value $a \in A$ is substituted for x in e, then this expression (i.e., $f(a)$) evaluates to some (unique) value $b \in B$

What is a Function? – Intensional View

- A function $f : A \rightarrow B$ is an abstraction $\lambda x.e$, where x is a variable name, and e is an expression, such that when a value $a \in A$ is substituted for x in e, then this expression (i.e., $f(a)$) evaluates to some (unique) value $b \in B$
- Equality of functions

What is a Function? – Intensional View

- A function $f : A \rightarrow B$ is an abstraction $\lambda x.e$, where x is a variable name, and e is an expression, such that when a value $a \in A$ is substituted for x in e, then this expression (i.e., $f(a)$) evaluates to some (unique) value $b \in B$
- Equality of functions
  - two functions are equal if they are defined by (essentially) the same abstraction/formula

Observations About Functions

- functions need not be explicitly named

Observations About Functions

- functions need not be explicitly named
  - i.e., identity functions $f(x) = g(x) = x$ could be expressed by $x \mapsto x$ having no name

Observations About Functions

- functions need not be explicitly named
  - i.e., identity functions $f(x) = g(x) = x$ could be expressed by $x \mapsto x$ having no name
- specific choice of argument names are irrelevant

Observations About Functions

- functions need not be explicitly named
  - i.e., identity functions $f(x) = g(x) = x$ could be expressed by $x \mapsto x$ having no name
- specific choice of argument names are irrelevant
  - i.e., $(x, y) \mapsto x - y$ and $(u, v) \mapsto u - v$ are the same

Observations About Functions

- functions need not be explicitly named
  - i.e., identity functions $f(x) = g(x) = x$ could be expressed by $x \mapsto x$ having no name
- specific choice of argument names are irrelevant
  - i.e., $(x, y) \mapsto x - y$ and $(u, v) \mapsto u - v$ are the same
- functions can be written in a way to accept a single input (currification)

Observations About Functions

- functions need not be explicitly named
  - i.e., identity functions $f(x) = g(x) = x$ could be expressed by $x \mapsto x$ having no name
- specific choice of argument names are irrelevant
  - i.e., $(x, y) \mapsto x - y$ and $(u, v) \mapsto u - v$ are the same
- functions can be written in a way to accept a single input (currification)
  - i.e., $(x, y) \mapsto x - y$ could be rewritten as $x \mapsto (y \mapsto x - y)$

λ-Calculus

- λ-calculus: theory of functions as formulas (based on aforementioned observations) – mathematical formalism to express computations as functions

λ-Calculus

- λ-calculus: theory of functions as formulas (based on aforementioned observations) – mathematical formalism to express computations as functions
- Easier manipulation of functions using expressions

λ-Calculus

- λ-calculus: theory of functions as formulas (based on aforementioned observations) – mathematical formalism to express computations as functions
- Easier manipulation of functions using expressions
- Examples of λ-notation (expressions):

λ-Calculus

- λ-calculus: theory of functions as formulas (based on aforementioned observations) – mathematical formalism to express computations as functions
- Easier manipulation of functions using expressions
- Examples of λ-notation (expressions):
  - The identity function $f(x) = x$ is denoted as $\lambda x.x$

λ-Calculus

- λ-calculus: theory of functions as formulas (based on aforementioned observations) – mathematical formalism to express computations as functions
- Easier manipulation of functions using expressions
- Examples of λ-notation (expressions):
  - The identity function $f(x) = x$ is denoted as $\lambda x.x$
  - $\lambda x.x$ is the same as $\lambda y.y$ (called α-equivalence)

λ-Calculus

- λ-calculus: theory of functions as formulas (based on aforementioned observations) – mathematical formalism to express computations as functions
- Easier manipulation of functions using expressions
- Examples of λ-notation (expressions):
  - The identity function $f(x) = x$ is denoted as $\lambda x.x$
  - $\lambda x.x$ is the same as $\lambda y.y$ (called α-equivalence)
  - Function defined as $f := x \mapsto x^2$ is written as $\lambda x.x^2$

λ-Calculus

- λ-calculus: theory of functions as formulas (based on aforementioned observations) – mathematical formalism to express computations as functions
- Easier manipulation of functions using expressions
- Examples of λ-notation (expressions):
  - The identity function $f(x) = x$ is denoted as $\lambda x.x$
  - $\lambda x.x$ is the same as $\lambda y.y$ (called α-equivalence)
  - Function defined as $f := x \mapsto x^2$ is written as $\lambda x.x^2$
  - $f(5)$ is $(\lambda x.x^2)(5)$, and evaluates to 25 (called β-reduction)

Definition (λ-terms)

$$
\begin{array}{lll}
\text{Terms } s, t, r := & & \\
\quad | & x & \text{variable (countable many)} \\
\quad | & \lambda x.\, t & \text{function abstraction} \\
\quad | & s\ t & \text{function application}
\end{array}
$$

Definition (λ-equations)

1. β-equivalence – to get there, we first need to define α-equivalence and substitution

Definition (Free Variables)

- An occurrence of variable x is said to be bound when it occurs in the body t of an abstraction $\lambda x.t$

Definition (Free Variables)

- An occurrence of variable x is said to be bound when it occurs in the body t of an abstraction $\lambda x.t$
- An occurrence of x is free if it appears in a position where it is not bound by an enclosing abstraction of x

Definition (Free Variables)

- An occurrence of variable x is said to be bound when it occurs in the body t of an abstraction $\lambda x.t$
- An occurrence of x is free if it appears in a position where it is not bound by an enclosing abstraction of x
- The set of free variables of a term is

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

Definition (Free Variables)

- An occurrence of variable x is said to be bound when it occurs in the body t of an abstraction $\lambda x.t$
- An occurrence of x is free if it appears in a position where it is not bound by an enclosing abstraction of x
- The set of free variables of a term is

$$
\begin{aligned}
FV(x) &= \{x\} \\
FV(\lambda x.e) &= FV(e) \setminus \{x\}
\end{aligned}
$$

Definition (Free Variables)

- An occurrence of variable x is said to be bound when it occurs in the body t of an abstraction $\lambda x.t$
- An occurrence of x is free if it appears in a position where it is not bound by an enclosing abstraction of x
- The set of free variables of a term is

$$
\begin{array}{lcl}
FV(x) & = & \{x\} \\
FV(\lambda x.e) & = & FV(e)\backslash\{x\} \\
FV(e_1 e_2) & = & FV(e_1) \cup FV(e_2)
\end{array}
$$

Definition (Free Variables)

- An occurrence of variable x is said to be bound when it occurs in the body t of an abstraction $\lambda x.t$
- An occurrence of x is free if it appears in a position where it is not bound by an enclosing abstraction of x
- The set of free variables of a term is

$$
\begin{array}{lcl}
FV(x) & = & \{x\} \\
FV(\lambda x.e) & = & FV(e)\backslash\{x\} \\
FV(e_1 e_2) & = & FV(e_1) \cup FV(e_2)
\end{array}
$$

- A term e is called closed if $FV(e) = \emptyset$

Example (Free and Bound Variables)

- Let M be the following lambda term: $\lambda x.\lambda y.\Big((\lambda z.\lambda v.z\,(z\,v))\,(x\,y)\,(z\,u)\Big)$

Example (Free and Bound Variables)

- Let M be the following lambda term: $\lambda x.\lambda y.\Big((\lambda z.\lambda v.z\,(z\,v))\,(x\,y)\,(z\,u)\Big)$

$$FV(M) \quad = \quad FV\big((\lambda z.\lambda v.z\,(z\,v))\,(x\,y)\,(z\,u)\big)\backslash\{x, y\}$$

Example (Free and Bound Variables)

- Let M be the following lambda term: $\lambda x.\lambda y.\Big((\lambda z.\lambda v.z\,(z\,v))\,(x\,y)\,(z\,u)\Big)$

$$
\begin{aligned}
FV(M) \quad &= \quad FV\big((\lambda z.\lambda v.z\,(z\,v))\,(x\,y)\,(z\,u)\big)\backslash\{x, y\} \\
&= \quad \big(FV(\lambda z.\lambda v.z\,(z\,v)) \cup FV(x\,y) \cup FV(z\,u)\big)\backslash\{x, y\}
\end{aligned}
$$

Example (Free and Bound Variables)

- Let M be the following lambda term: $\lambda x.\lambda y.\Big((\lambda z.\lambda v.z\,(z\,v))\,(x\,y)\,(z\,u)\Big)$

$$
\begin{array}{rcl}
FV(M) & = & FV\big((\lambda z.\lambda v.z\,(z\,v))\,(x\,y)\,(z\,u)\big)\backslash\{x, y\} \\
 & = & \big(FV(\lambda z.\lambda v.z\,(z\,v)) \cup FV(x\,y) \cup FV(z\,u)\big)\backslash\{x, y\} \\
 & = & \big((\{z, v\}\backslash\{z, v\}) \cup \{x, y\} \cup \{z, u\}\big)\backslash\{x, y\}
\end{array}
$$

Example (Free and Bound Variables)

- Let M be the following lambda term: $\lambda x.\lambda y.\Big((\lambda z.\lambda v.z\,(z\,v))\,(x\,y)\,(z\,u)\Big)$

$$
\begin{aligned}
FV(M) &= FV\big((\lambda z.\lambda v.z\,(z\,v))\,(x\,y)\,(z\,u)\big)\backslash\{x, y\} \\
&= \big(FV(\lambda z.\lambda v.z\,(z\,v)) \cup FV(x\,y) \cup FV(z\,u)\big)\backslash\{x, y\} \\
&= \big((\{z, v\}\backslash\{z, v\}) \cup \{x, y\} \cup \{z, u\}\big)\backslash\{x, y\} \\
&= \{z, u\}
\end{aligned}
$$

Definition (α-equivalence)

- names of λ-bound variables should not affect meaning

Definition (α-equivalence)

- names of λ-bound variables should not affect meaning
- e.g., $\lambda f.\, \lambda x.\, f\ x$ should have the same meaning as $\lambda x.\, \lambda y.\, x\ y$

Definition (α-equivalence)

- names of λ-bound variables should not affect meaning
- e.g., $\lambda f.\, \lambda x.\, f\ x$ should have the same meaning as $\lambda x.\, \lambda y.\, x\ y$
- this issue is best dealt with at the level of syntax rather than semantics

Definition (α-equivalence)

- names of λ-bound variables should not affect meaning
- e.g., $\lambda f.\, \lambda x.\, f\ x$ should have the same meaning as $\lambda x.\, \lambda y.\, x\ y$
- this issue is best dealt with at the level of syntax rather than semantics
- from now on we re-define λ term to mean not an abstract syntax tree but rather an equivalence class of such trees with respect to α-equivalence $s =_\alpha t$:

$$\frac{}{x =_\alpha x}$$

$$\frac{s =_\alpha s' \qquad t =_\alpha t'}{s\ t =_\alpha s'\ t'}$$

$$\frac{t \cdot (y\ x) =_\alpha t' \cdot (y\ x') \qquad y \text{ does not occur in } \{x,x',t,t'\}}{\lambda x.\, t =_\alpha \lambda x'.\, t'}$$

where $t \cdot (y\ x)$ denotes the result of replacing all occurrences of x with y in t

Example (α-equivalence)

$$\begin{array}{rcccl} \lambda x.\, x\, x & =_\alpha & \lambda y.\, y\, y & \neq_\alpha & \lambda x.\, x\, y \\ (\lambda y.\, y)\, x & =_\alpha & (\lambda x.\, x)\, x & \neq_\alpha & (\lambda x.\, x)\, y \end{array}$$

Definition (substitution)

- substitution $t[s/x]$ denotes the result of replacing all free occurrences of variable x in term t (i.e. those not occurring within the scope of a $\lambda x.\_$ binder) by the term s

Definition (substitution)

- substitution $t[s/x]$ denotes the result of replacing all free occurrences of variable x in term t (i.e. those not occurring within the scope of a $\lambda x.\_$ binder) by the term s
- alpha-converting λ-bound variables in t to avoid them "capturing" any free variables of t

Definition (substitution)

- substitution $t[s/x]$ denotes the result of replacing all free occurrences of variable x in term t (i.e. those not occurring within the scope of a $\lambda x.\_$ binder) by the term s
- alpha-converting λ-bound variables in t to avoid them “capturing” any free variables of t
- e.g., $(\lambda y.\,(y, x))[y/x]$ is $\lambda z.\,(z, y)$ and is not $\lambda y.\,(y, y)$

Definition (substitution)

- substitution $t[s/x]$ denotes the result of replacing all free occurrences of variable x in term t (i.e. those not occurring within the scope of a $\lambda x.\,\_$ binder) by the term s
- alpha-converting λ-bound variables in t to avoid them "capturing" any free variables of t
- e.g., $(\lambda y.\,(y, x))[y/x]$ is $\lambda z.\,(z, y)$ and is not $\lambda y.\,(y, y)$
- the relation $t[s/x] = t'$ can be inductively defined by the following rules:

$$\frac{}{x[s/x] = s} \qquad \frac{y \neq x}{y[s/x] = y}$$

$$\frac{t[s/x] = t' \qquad y \neq x \text{ and } y \text{ does not freely occur in } s}{(\lambda y.\,t)[s/x] = \lambda y.\,t'}$$

$$\frac{t_1[s/x] = t_1' \qquad t_2[s/x] = t_2'}{(t_1\ t_2)[s/x] = t_1'\ t_2'}$$

Example (Substitution)

$$(\lambda x.\lambda y.\, x\ y\ z)[w/z] \quad = \quad \lambda x.\lambda y.\, x\ y\ w$$

Example (Substitution)

$$
\begin{array}{lcl}
(\lambda x.\, \lambda y.\, x\ y\ z)[w/z] & = & \lambda x.\, \lambda y.\, x\ y\ w \\
(\lambda y.\, y\ x)[y/x] & = & \lambda z.\, z\ y
\end{array}
$$

Example (Substitution)

$$
\begin{array}{lcl}
(\lambda x.\, \lambda y.\, x\ y\ z)[w/z] & = & \lambda x.\, \lambda y.\, x\ y\ w \\
(\lambda y.\, y\ x)[y/x] & = & \lambda z.\, z\ y \\
(\lambda x.\, \lambda y.\, x\ y\ z)[y/z] & = & \lambda x.\, \lambda a.\, x\ a\ y
\end{array}
$$

Example (Substitution)

$$
\begin{array}{lcl}
(\lambda x.\lambda y.\,x\ y\ z)[w/z] & = & \lambda x.\lambda y.\,x\ y\ w \\
(\lambda y.\,y\ x)[y/x] & = & \lambda z.\,z\ y \\
(\lambda x.\lambda y.\,x\ y\ z)[y/z] & = & \lambda x.\lambda a.\,x\ a\ y \\
(\lambda x.\lambda y.\,x\ y\ z)[(\lambda x.\,x\ x)/y] & = & \lambda x.\lambda y.\,x\ y\ z
\end{array}
$$

Example (Substitution)

$$
\begin{array}{lcl}
(\lambda x.\,\lambda y.\,x\ y\ z)[w/z] & = & \lambda x.\,\lambda y.\,x\ y\ w \\
(\lambda y.\,y\ x)[y/x] & = & \lambda z.\,z\ y \\
(\lambda x.\,\lambda y.\,x\ y\ z)[y/z] & = & \lambda x.\,\lambda a.\,x\ a\ y \\
(\lambda x.\,\lambda y.\,x\ y\ z)[(\lambda x.\,x\ x)/y] & = & \lambda x.\,\lambda y.\,x\ y\ z \\
(\lambda x.\,\lambda y.\,x\ y\ z)[(\lambda x.\,x\ y)/z] & = & \lambda x.\,\lambda a.\,x\ a\ (\lambda x.\,x\ y)
\end{array}
$$

Definition (β-equivalence (or β-reduction))

the relation $s =_\beta t$ (where s and t over terms) is inductively defined by the following rules:

- β-conversion

$$\frac{}{(\lambda x.\, t)\ s =_\beta t[s/x]}$$

Definition (β-equivalence (or β-reduction))

the relation $s =_\beta t$ (where s and t over terms) is inductively defined by the following rules:

- β-conversion

$$\frac{}{(\lambda x.\, t)\ s =_\beta t[s/x]}$$

- congruence rules

$$\frac{t =_\beta t'}{\lambda x.\, t =_\beta \lambda x.\, t'} \qquad \frac{s =_\beta s' \qquad t =_\beta t'}{s\ t =_\beta s'\ t'}$$

Definition (β-equivalence (or β-reduction))

the relation $s =_\beta t$ (where s and t over terms) is inductively defined by the following rules:

- β-conversion

$$\frac{}{(\lambda x.\, t)\ s =_\beta t[s/x]}$$

- congruence rules

$$\frac{t =_\beta t'}{\lambda x.\, t =_\beta \lambda x.\, t'} \qquad \frac{s =_\beta s' \qquad t =_\beta t'}{s\ t =_\beta s'\ t'}$$

- $=_\beta$ is reflexive, symmetric and transitive

$$\frac{}{t =_\beta t} \qquad \frac{s =_\beta t}{t =_\beta s} \qquad \frac{r =_\beta s \qquad s =_\beta t}{r =_\beta t}$$

Example (β-reduction)

$$\underline{(\lambda x.\, x)\ (\lambda x.\, x)}$$

Example (β-reduction)

$$\underline{(\lambda x.\, x)\ (\lambda x.\, x)} \ \to_\beta\ x[x := \lambda x.\, x]$$

Example (β-reduction)

$$\underline{(\lambda x.\, x)\ (\lambda x.\, x)} \rightarrow_{\beta} x[x := \lambda x.\, x] \qquad = \quad \lambda x.\, x$$

Example (β-reduction)

$$\underline{(\lambda x.\, x)\ (\lambda x.\, x)} \rightarrow_\beta x[x := \lambda x.\, x] \qquad = \lambda x.\, x$$

Example (β-reduction)

$$
\begin{aligned}
\underline{(\lambda x.\, x)\ (\lambda x.\, x)} &\rightarrow_\beta x[x := \lambda x.\, x] &&= \lambda x.\, x \\
\underline{(\lambda xy.\, y)\ (\lambda x.\, x)} &\rightarrow_\beta (\lambda y.\, y)[x := \lambda x.\, x]
\end{aligned}
$$

Example (β-reduction)

$$\begin{array}{rcll}
\underline{(\lambda x.\, x)\ (\lambda x.\, x)} & \rightarrow_\beta & x[x := \lambda x.\, x] & = \lambda x.\, x \\
\underline{(\lambda xy.\, y)\ (\lambda x.\, x)} & \rightarrow_\beta & (\lambda y.\, y)[x := \lambda x.\, x] & = \lambda y.\, y
\end{array}$$

Example (β-reduction)

$$
\begin{array}{rlcl}
\underline{(\lambda x.\, x)\ (\lambda x.\, x)} & \to_\beta\ x[x := \lambda x.\, x] & = & \lambda x.\, x \\
\underline{(\lambda xy.\, y)\ (\lambda x.\, x)} & \to_\beta\ (\lambda y.\, y)[x := \lambda x.\, x] & = & \lambda y.\, y
\end{array}
$$

Example (β-reduction)

$$\begin{array}{rcll}
\underline{(\lambda x.\, x)\ (\lambda x.\, x)} & \to_\beta & x[x := \lambda x.\, x] & = \lambda x.\, x \\
\underline{(\lambda xy.\, y)\ (\lambda x.\, x)} & \to_\beta & (\lambda y.\, y)[x := \lambda x.\, x] & = \lambda y.\, y \\
\underline{(\lambda xyz.\, x\, z\ (y\ z))\ (\lambda x.\, x)} & \to_\beta & \lambda yz.\, \underline{(\lambda x.\, x)\ z}\ (y\ z) &
\end{array}$$

Example (β-reduction)

$$
\begin{array}{rcll}
\underline{(\lambda x.\, x)\ (\lambda x.\, x)} & \rightarrow_\beta & x[x := \lambda x.\, x] & = \ \lambda x.\, x \\
\underline{(\lambda x y.\, y)\ (\lambda x.\, x)} & \rightarrow_\beta & (\lambda y.\, y)[x := \lambda x.\, x] & = \ \lambda y.\, y \\
\underline{(\lambda x y z.\, x\, z\, (y\, z))\ (\lambda x.\, x)} & \rightarrow_\beta & \lambda y z.\, \underline{(\lambda x.\, x)\, z}\, (y\, z) & \rightarrow_\beta \ \lambda y z.\, z\, (y\, z)
\end{array}
$$

Example (β-reduction)

$$
\begin{aligned}
\underline{(\lambda x.\, x)\ (\lambda x.\, x)} &\to_\beta x[x := \lambda x.\, x] &&= \lambda x.\, x \\
\underline{(\lambda xy.\, y)\ (\lambda x.\, x)} &\to_\beta (\lambda y.\, y)[x := \lambda x.\, x] &&= \lambda y.\, y \\
\underline{(\lambda xyz.\, x\, z\, (y\, z))\ (\lambda x.\, x)} &\to_\beta \lambda yz.\, \underline{(\lambda x.\, x)\, z}\, (y\, z) &&\to_\beta \lambda yz.\, z\, (y\, z)
\end{aligned}
$$

Example (β-reduction)

$$
\begin{array}{rlll}
\underline{(\lambda x.\, x)\ (\lambda x.\, x)} & \to_\beta\ x[x := \lambda x.\, x] & = & \lambda x.\, x \\
\underline{(\lambda xy.\, y)\ (\lambda x.\, x)} & \to_\beta\ (\lambda y.\, y)[x := \lambda x.\, x] & = & \lambda y.\, y \\
\underline{(\lambda xyz.\, x\, z\, (y\, z))\ (\lambda x.\, x)} & \to_\beta\ \lambda yz.\, \underline{(\lambda x.\, x)\, z}\, (y\, z) & \to_\beta & \lambda yz.\, z\, (y\, z) \\
\underline{(\lambda x.\, x\, x)\ (\lambda x.\, x\, x)} & \to_\beta\ \underline{(\lambda x.\, x\, x)\ (\lambda x.\, x\, x)} & &
\end{array}
$$

Example (β-reduction)

$$
\begin{array}{rlll}
\underline{(\lambda x.\, x)\ (\lambda x.\, x)} & \to_\beta\ x[x := \lambda x.\, x] & = & \lambda x.\, x \\
\underline{(\lambda xy.\, y)\ (\lambda x.\, x)} & \to_\beta\ (\lambda y.\, y)[x := \lambda x.\, x] & = & \lambda y.\, y \\
\underline{(\lambda xyz.\, x\, z\, (y\, z))\ (\lambda x.\, x)} & \to_\beta\ \lambda yz.\, \underline{(\lambda x.\, x)\ z}\ (y\, z) & \to_\beta & \lambda yz.\, z\ (y\, z) \\
\underline{(\lambda x.\, x\, x)\ (\lambda x.\, x\, x)} & \to_\beta\ \underline{(\lambda x.\, x\, x)\ (\lambda x.\, x\, x)} & \to_\beta & \cdots
\end{array}
$$

Outline

1. λ-Calculus
2. Programming In λ-Calculus
3. Typing In General
4. STLC($\lambda^{\rightarrow}$)

Programming in λ-Calculus

- Recall Church's thesis: Turing machines $\iff$ λ-Calculus

Programming in λ-Calculus

- Recall Church's thesis: Turing machines $\iff$ λ-Calculus
- We shall see how different types of data and related operations can be programmed in λ-calculus.

Programming in λ-Calculus

- Recall Church's thesis: Turing machines $\iff$ λ-Calculus
- We shall see how different types of data and related operations can be programmed in λ-calculus.
- Functions with many arguments: currification

Representing Booleans

$$
\begin{aligned}
\texttt{true} &:= \lambda x.\lambda y.x \\
\texttt{false} &:= \lambda x.\lambda y.y \\
\texttt{not} &:= \lambda x.\lambda y.\lambda z.x\,z\,y \\
\texttt{and} &:= \lambda x.\lambda y.x\,y\,x \\
\texttt{or} &:= \lambda x.\lambda y.x\,x\,y \\
\texttt{if}\ a\ \texttt{then}\ b\ \texttt{else}\ c &:= \lambda a.\lambda b.\lambda c.a\,b\,c
\end{aligned}
$$

Representing Booleans

$$
\begin{array}{lcl}
\texttt{true} & := & \lambda x.\lambda y.x \\
\texttt{false} & := & \lambda x.\lambda y.y \\
\texttt{not} & := & \lambda x.\lambda y.\lambda z.x\,z\,y \\
\texttt{and} & := & \lambda x.\lambda y.x\,y\,x \\
\texttt{or} & := & \lambda x.\lambda y.x\,x\,y \\
\texttt{if}\ a\ \texttt{then}\ b\ \texttt{else}\ c & := & \lambda a.\lambda b.\lambda c.\,a\,b\,c
\end{array}
$$

For example:
$(\lambda a.\,\lambda b.\,\lambda c.\,a\,b\,c)(\lambda x.\lambda y.x) =_\beta \lambda b.\,\lambda c.\,(\lambda x.\lambda y.x)\ (b\,c) =_\beta \lambda b.\,\lambda c.\,(\lambda y.b)\ (c) =_\beta \lambda b.\,\lambda c.\,b$

Representing Booleans

`true`	:=	$\lambda x.\lambda y.x$
`false`	:=	$\lambda x.\lambda y.y$
`not`	:=	$\lambda x.\lambda y.\lambda z.x\,z\,y$
`and`	:=	$\lambda x.\lambda y.x\,y\,x$
`or`	:=	$\lambda x.\lambda y.x\,x\,y$
`if` a `then` b `else` c	:=	$\lambda a.\lambda b.\lambda c.\,a\,b\,c$

For example:

$(\lambda a.\,\lambda b.\,\lambda c.\,a\,b\,c)(\lambda x.\lambda y.x) =_\beta \lambda b.\,\lambda c.\,(\lambda x.\lambda y.x)\ (b\,c) =_\beta \lambda b.\,\lambda c.\,(\lambda y.b)\ (c) =_\beta \lambda b.\,\lambda c.\,b$

$(\lambda a.\,\lambda b.\,\lambda c.\,a\,b\,c)(\lambda x.\lambda y.y) =_\beta \lambda b.\,\lambda c.\,(\lambda x.\lambda y.y)\ (b\,c) =_\beta \lambda b.\,\lambda c.\,(\lambda y.y)\ (c) =_\beta \lambda b.\,\lambda c.\,c$

Representing Booleans

$$
\begin{array}{lcl}
\texttt{true} & := & \lambda x.\lambda y.x \\
\texttt{false} & := & \lambda x.\lambda y.y \\
\texttt{not} & := & \lambda x.\lambda y.\lambda z.x\,z\,y \\
\texttt{and} & := & \lambda x.\lambda y.x\,y\,x \\
\texttt{or} & := & \lambda x.\lambda y.x\,x\,y \\
\texttt{if}\ a\ \texttt{then}\ b\ \texttt{else}\ c & := & \lambda a.\lambda b.\lambda c.\,a\,b\,c
\end{array}
$$

For example:

$(\lambda a.\,\lambda b.\,\lambda c.\,a\,b\,c)(\lambda x.\lambda y.x) =_\beta \lambda b.\,\lambda c.\,(\lambda x.\lambda y.x)\,(b\,c) =_\beta \lambda b.\,\lambda c.\,(\lambda y.b)\,(c) =_\beta \lambda b.\,\lambda c.\,b$

$(\lambda a.\,\lambda b.\,\lambda c.\,a\,b\,c)(\lambda x.\lambda y.y) =_\beta \lambda b.\,\lambda c.\,(\lambda x.\lambda y.y)\,(b\,c) =_\beta \lambda b.\,\lambda c.\,(\lambda y.y)\,(c) =_\beta \lambda b.\,\lambda c.\,c$

$$
\begin{array}{lcl}
\texttt{not true} & := & (\lambda x.\lambda y.\lambda z.x\,z\,y)\,(\lambda x.\lambda y.x) \\
 & =_\beta & (\lambda y.\lambda z.(\lambda x.\lambda y.x)\,z\,y) \\
 & =_\beta & (\lambda y.\lambda z.z)
\end{array}
$$

Representing Natural Numbers

- Church numerals:

$$\mathtt{0} \quad := \quad \lambda s.\lambda z.z$$

Representing Natural Numbers

- Church numerals:

$$
\begin{aligned}
\mathtt{0} &:= \lambda s.\lambda z.z \\
\mathtt{1} &:= \lambda s.\lambda z.s\,z
\end{aligned}
$$

Representing Natural Numbers

- Church numerals:

$$
\begin{aligned}
\mathtt{0} &:= \lambda s.\lambda z.z \\
\mathtt{1} &:= \lambda s.\lambda z.s\,z \\
\mathtt{2} &:= \lambda s.\lambda z.s\,(s\,z)
\end{aligned}
$$

Representing Natural Numbers

- Church numerals:

$$\begin{array}{lcl} \mathtt{0} & := & \lambda s.\lambda z.z \\ \mathtt{1} & := & \lambda s.\lambda z.s\,z \\ \mathtt{2} & := & \lambda s.\lambda z.s\,(s\,z) \\ \mathtt{3} & := & \lambda s.\lambda z.s\,(s\,(s\,z)) \end{array}$$

Representing Natural Numbers

- Church numerals:

$$
\begin{array}{lcl}
\mathtt{0} & := & \lambda s.\lambda z.z \\
\mathtt{1} & := & \lambda s.\lambda z.s\,z \\
\mathtt{2} & := & \lambda s.\lambda z.s\,(s\,z) \\
\mathtt{3} & := & \lambda s.\lambda z.s\,(s\,(s\,z)) \\
\mathtt{n} & := & \lambda s.\lambda z.s^n\,z
\end{array}
$$

Representing Natural Numbers

- Church numerals:

$$
\begin{aligned}
\mathtt{0} &:= \lambda s.\lambda z.z \\
\mathtt{1} &:= \lambda s.\lambda z.s\,z \\
\mathtt{2} &:= \lambda s.\lambda z.s\,(s\,z) \\
\mathtt{3} &:= \lambda s.\lambda z.s\,(s\,(s\,z)) \\
\mathtt{n} &:= \lambda s.\lambda z.s^{n}\,z
\end{aligned}
$$

- Some operations:

$$
\mathtt{add} \quad := \quad \lambda M.\lambda N.\lambda s.\lambda z.N\,s\,(M\,s\,z)
$$

Representing Natural Numbers

- Church numerals:

$$
\begin{aligned}
\mathtt{0} &:= \lambda s.\lambda z.z \\
\mathtt{1} &:= \lambda s.\lambda z.s\,z \\
\mathtt{2} &:= \lambda s.\lambda z.s\,(s\,z) \\
\mathtt{3} &:= \lambda s.\lambda z.s\,(s\,(s\,z)) \\
\mathtt{n} &:= \lambda s.\lambda z.s^{n}\,z
\end{aligned}
$$

- Some operations:

$$
\begin{aligned}
\mathtt{add} &:= \lambda M.\lambda N.\lambda s.\lambda z.N\,s\,(M\,s\,z) \\
\mathtt{mult} &:= \lambda M.\lambda N.\lambda s.\lambda z.N\,(M\,s)\,z
\end{aligned}
$$

Representing Natural Numbers

- Church numerals:

$$\begin{array}{lcl} \mathtt{0} & := & \lambda s.\lambda z.z \\ \mathtt{1} & := & \lambda s.\lambda z.s\,z \\ \mathtt{2} & := & \lambda s.\lambda z.s\,(s\,z) \\ \mathtt{3} & := & \lambda s.\lambda z.s\,(s\,(s\,z)) \\ \mathtt{n} & := & \lambda s.\lambda z.s^n\,z \end{array}$$

- Some operations:

$$\begin{array}{lcl} \mathtt{add} & := & \lambda M.\lambda N.\lambda s.\lambda z.N\,s\,(M\,s\,z) \\ \mathtt{mult} & := & \lambda M.\lambda N.\lambda s.\lambda z.N\,(M\,s)\,z \\ \mathtt{pred} & := & \lambda n.\lambda s.\lambda z.n\,(\lambda g.\lambda h.h\,(g\,s))(\lambda u.z)(\lambda u.u) \end{array}$$

Representing Natural Numbers

- Church numerals:

$$
\begin{array}{lcl}
\mathtt{0} & := & \lambda s.\lambda z.z \\
\mathtt{1} & := & \lambda s.\lambda z.s\,z \\
\mathtt{2} & := & \lambda s.\lambda z.s\,(s\,z) \\
\mathtt{3} & := & \lambda s.\lambda z.s\,(s\,(s\,z)) \\
\mathtt{n} & := & \lambda s.\lambda z.s^n\,z
\end{array}
$$

- Some operations:

$$
\begin{array}{lcl}
\mathtt{add} & := & \lambda M.\lambda N.\lambda s.\lambda z.N\,s\,(M\,s\,z) \\
\mathtt{mult} & := & \lambda M.\lambda N.\lambda s.\lambda z.N\,(M\,s)\,z \\
\mathtt{pred} & := & \lambda n.\lambda s.\lambda z.n\,(\lambda g.\lambda h.h\,(g\,s))(\lambda u.z)(\lambda u.u) \\
\mathtt{subtr} & := & \lambda m.\lambda n.n\,\mathrm{pred}\,m
\end{array}
$$

Representing Natural Numbers

- Church numerals:

$$
\begin{array}{lcl}
\mathtt{0} & := & \lambda s.\lambda z.z \\
\mathtt{1} & := & \lambda s.\lambda z.s\,z \\
\mathtt{2} & := & \lambda s.\lambda z.s\,(s\,z) \\
\mathtt{3} & := & \lambda s.\lambda z.s\,(s\,(s\,z)) \\
\mathtt{n} & := & \lambda s.\lambda z.s^{n}\,z
\end{array}
$$

- Some operations:

$$
\begin{array}{lcl}
\mathtt{add} & := & \lambda M.\lambda N.\lambda s.\lambda z.N\,s\,(M\,s\,z) \\
\mathtt{mult} & := & \lambda M.\lambda N.\lambda s.\lambda z.N\,(M\,s)\,z \\
\mathtt{pred} & := & \lambda n.\lambda s.\lambda z.n\,(\lambda g.\lambda h.h\,(g\,s))(\lambda u.z)(\lambda u.u) \\
\mathtt{subtr} & := & \lambda m.\lambda n.n\,\mathrm{pred}\,m \\
\mathtt{isZero} & := & \lambda n.n(\lambda x.\mathrm{false})\,\mathrm{true}
\end{array}
$$

Representing Natural Numbers

- Church numerals:

$$
\begin{array}{lcl}
\mathtt{0} & := & \lambda s.\lambda z.z \\
\mathtt{1} & := & \lambda s.\lambda z.s\,z \\
\mathtt{2} & := & \lambda s.\lambda z.s\,(s\,z) \\
\mathtt{3} & := & \lambda s.\lambda z.s\,(s\,(s\,z)) \\
\mathtt{n} & := & \lambda s.\lambda z.s^{n}\,z
\end{array}
$$

- Some operations:

$$
\begin{array}{lcl}
\mathtt{add} & := & \lambda M.\lambda N.\lambda s.\lambda z.N\,s\,(M\,s\,z) \\
\mathtt{mult} & := & \lambda M.\lambda N.\lambda s.\lambda z.N\,(M\,s)\,z \\
\mathtt{pred} & := & \lambda n.\lambda s.\lambda z.n\,(\lambda g.\lambda h.h\,(g\,s))(\lambda u.z)(\lambda u.u) \\
\mathtt{subtr} & := & \lambda m.\lambda n.n\,\mathrm{pred}\,m \\
\mathtt{isZero} & := & \lambda n.n(\lambda x.\mathrm{false})\,\mathrm{true} \\
\mathtt{leq} & := & \lambda m.\lambda n.\mathtt{isZero}\,(\mathtt{subtr}\;m\;n)
\end{array}
$$

Representing Natural Numbers

- Church numerals:

$$
\begin{array}{lcl}
\texttt{0} & := & \lambda s.\lambda z.z \\
\texttt{1} & := & \lambda s.\lambda z.s\,z \\
\texttt{2} & := & \lambda s.\lambda z.s\,(s\,z) \\
\texttt{3} & := & \lambda s.\lambda z.s\,(s\,(s\,z)) \\
\texttt{n} & := & \lambda s.\lambda z.s^n\,z
\end{array}
$$

- Some operations:

$$
\begin{array}{lcl}
\texttt{add} & := & \lambda M.\lambda N.\lambda s.\lambda z.N\,s\,(M\,s\,z) \\
\texttt{mult} & := & \lambda M.\lambda N.\lambda s.\lambda z.N\,(M\,s)\,z \\
\texttt{pred} & := & \lambda n.\lambda s.\lambda z.n\,(\lambda g.\lambda h.h\,(g\,s))(\lambda u.z)(\lambda u.u) \\
\texttt{subtr} & := & \lambda m.\lambda n.n\,\mathrm{pred}\,m \\
\texttt{isZero} & := & \lambda n.n(\lambda x.\mathrm{false})\,\mathrm{true} \\
\texttt{leq} & := & \lambda m.\lambda n.\texttt{isZero}\,(\texttt{subtr}\;m\;n) \\
\texttt{eq} & := & \lambda m.\lambda n.\mathrm{and}\,(\texttt{leq}\;m\;n)\,(\texttt{leq}\;n\;m)
\end{array}
$$

Example (addition)

$$\begin{aligned}
\texttt{add 2 3} \quad &:= \quad \lambda s.\lambda z.(\lambda s.\lambda z.sssz)\,s\,((\lambda s.\lambda z.ssz)\,sz) \\
&=_\beta \quad \lambda s.\lambda z.(\lambda z.sssz)((\lambda z.ssz)\,z) \\
&=_\beta \quad \lambda s.\lambda z.sss\,((\lambda z.ssz)\,z) \\
&=_\beta \quad \lambda s.\lambda z.sssssz
\end{aligned}$$

Example (addition)

$$
\begin{aligned}
\texttt{add 2 3} &:= \lambda s.\lambda z.(\lambda s.\lambda z.sssz)\,s\,((\lambda s.\lambda z.ssz)\,s\,z) \\
&=_\beta \lambda s.\lambda z.(\lambda z.sssz)((\lambda z.ssz)\,z) \\
&=_\beta \lambda s.\lambda z.sss\,((\lambda z.ssz)\,z) \\
&=_\beta \lambda s.\lambda z.sssssz
\end{aligned}
$$

$$
\begin{aligned}
\texttt{mult 3 2} &:= \lambda s.\lambda z.(\lambda s.\lambda z.ssz)\,((\lambda s.\lambda z.sssz)\,s)\,z \\
&=_\beta \lambda s.\lambda z.(\lambda z.((\lambda s.\lambda z.sssz)s)\,((\lambda s.\lambda z.sssz)\,s)\,z)\,z \\
&=_\beta \lambda s.\lambda z.((\lambda s.\lambda z.sssz)s)\,((\lambda s.\lambda z.sssz)\,s)\,z \\
&=_\beta \lambda s.\lambda z.(\lambda z.sssz)\,(\lambda z.sssz)\,z \\
&=_\beta \lambda s.\lambda z.sss\,(\lambda z.sssz)\,z \\
&=_\beta \lambda s.\lambda z.ssssssz
\end{aligned}
$$

Representing Pairs & Tuples

- Pairs

$$\texttt{pair} \quad := \quad \lambda e_1.\lambda e_2.\lambda z.z\, e_1\, e_2$$

Representing Pairs & Tuples

- Pairs

$$\texttt{pair} \quad := \quad \lambda e_1.\lambda e_2.\lambda z.z\, e_1\, e_2$$

- Projections

$$\begin{aligned} \texttt{first} \quad &:= \quad \lambda u.u\, \texttt{true} \\ \texttt{second} \quad &:= \quad \lambda u.u\, \texttt{false} \end{aligned}$$

Representing Pairs & Tuples

- Pairs

$$\texttt{pair} := \lambda e_1.\lambda e_2.\lambda z.z\; e_1\; e_2$$

- Projections

$$\begin{aligned}\texttt{first} &:= \lambda u.u\;\texttt{true}\\ \texttt{second} &:= \lambda u.u\;\texttt{false}\end{aligned}$$

- Tuples

$$\texttt{tuple} := \lambda e_1.\cdots.\lambda e_n.\lambda z.z\; e_1\;\cdots\; e_n$$

Representing Pairs & Tuples

- Pairs

$$\texttt{pair} \;:=\; \lambda e_1.\lambda e_2.\lambda z.z\, e_1\, e_2$$

- Projections

$$\begin{aligned}\texttt{first} &:= \lambda u.u\, \texttt{true}\\ \texttt{second} &:= \lambda u.u\, \texttt{false}\end{aligned}$$

- Tuples

$$\texttt{tuple} \;:=\; \lambda e_1.\cdots.\lambda e_n.\lambda z.z\, e_1 \,\cdots\, e_n$$

- i^{th} projection

$$\texttt{proj}_i \;:=\; \lambda u.u(\lambda x_1.\cdots.\lambda x_n.x_i)$$

Representing Lists

- constructors: cons and nil

$$\texttt{cons} \quad := \quad \lambda x.\lambda y.\texttt{pair}\ \texttt{false}\ (\texttt{pair}\ x\ y)$$

Representing Lists

- constructors: cons and nil

$$
\begin{aligned}
\texttt{cons} &:= \lambda x.\lambda y.\texttt{pair}\ \texttt{false}\ (\texttt{pair}\ x\ y) \\
\texttt{nil} &:= \lambda x.x
\end{aligned}
$$

Representing Lists

- constructors: cons and nil

$$\begin{aligned}\texttt{cons} &:= \lambda x.\lambda y.\texttt{pair}\ \texttt{false}\ (\texttt{pair}\ x\ y)\\ \texttt{nil} &:= \lambda x.x\end{aligned}$$

- head, tail and nullary check

$$\texttt{hd} := \lambda x.\texttt{first}\ (\texttt{second}\ x)$$

Representing Lists

- constructors: cons and nil

$$
\begin{aligned}
\texttt{cons} &:= \lambda x.\lambda y.\texttt{pair}\ \texttt{false}\ (\texttt{pair}\ x\ y) \\
\texttt{nil} &:= \lambda x.x
\end{aligned}
$$

- head, tail and nullary check

$$
\begin{aligned}
\texttt{hd} &:= \lambda x.\texttt{first}\ (\texttt{second}\ x) \\
\texttt{tl} &:= \lambda x.\texttt{second}\ (\texttt{second}\ x)
\end{aligned}
$$

Representing Lists

- constructors: cons and nil

$$\begin{aligned}\texttt{cons} &:= \lambda x.\lambda y.\texttt{pair}\ \texttt{false}\ (\texttt{pair}\ x\ y) \\ \texttt{nil} &:= \lambda x.x\end{aligned}$$

- head, tail and nullary check

$$\begin{aligned}\texttt{hd} &:= \lambda x.\texttt{first}\ (\texttt{second}\ x) \\ \texttt{tl} &:= \lambda x.\texttt{second}\ (\texttt{second}\ x) \\ \texttt{isNil} &:= \texttt{first}\end{aligned}$$

Encoding Recursion: the $\mathcal{Y}$ Combinator

- to encode recursion, we are looking for a combinator that, given an argument some function F, would not only reproduce itself but also pass F on itself.

Encoding Recursion: the $\mathcal{Y}$ Combinator

- to encode recursion, we are looking for a combinator that, given an argument some function F, would not only reproduce itself but also pass F on itself.

- we can then define:

$$\mathcal{Y} := \lambda f.(\lambda x.f\,(x\,x))\,(\lambda x.f\,(x\,x))$$

Encoding Recursion: the $\mathcal{Y}$ Combinator

- to encode recursion, we are looking for a combinator that, given an argument some function F, would not only reproduce itself but also pass F on itself.
- we can then define:

$$\mathcal{Y} := \lambda f.(\lambda x.f\,(x\,x))\,(\lambda x.f\,(x\,x))$$

- let us observe what happens when we pass a function F to the $\mathcal{Y}$ combinator:

$$\mathcal{Y}\,F \quad := \quad \lambda f.(\lambda x.f\,(x\,x))\,(\lambda x.f\,(x\,x))\,F$$

Encoding Recursion: the $\mathcal{Y}$ Combinator

- to encode recursion, we are looking for a combinator that, given an argument some function F, would not only reproduce itself but also pass F on itself.
- we can then define:

$$\mathcal{Y} \;:=\; \lambda f.(\lambda x.f\,(x\,x))\,(\lambda x.f\,(x\,x))$$

- let us observe what happens when we pass a function F to the $\mathcal{Y}$ combinator:

$$\begin{aligned}\mathcal{Y}\,F \quad &:= \quad \lambda f.(\lambda x.f\,(x\,x))\,(\lambda x.f\,(x\,x))\,F\\ &=_\beta \quad (\lambda x.F\,(x\,x))\,(\lambda x.F\,(x\,x))\end{aligned}$$

Encoding Recursion: the $\mathcal{Y}$ Combinator

- to encode recursion, we are looking for a combinator that, given an argument some function F, would not only reproduce itself but also pass F on itself.
- we can then define:

$$\mathcal{Y} \;:=\; \lambda f.(\lambda x.f\,(x\,x))\,(\lambda x.f\,(x\,x))$$

- let us observe what happens when we pass a function F to the $\mathcal{Y}$ combinator:

$$\begin{aligned}
\mathcal{Y}\,F \quad &:= \quad \lambda f.(\lambda x.f\,(x\,x))\,(\lambda x.f\,(x\,x))\,F \\
&=_\beta \quad (\lambda x.F\,(x\,x))\,(\lambda x.F\,(x\,x)) \\
&=_\beta \quad F\Big(\underbrace{(\lambda x.F\,(x\,x))\,(\lambda x.F\,(x\,x))}_{\mathcal{Y}\,F}\Big) = F\,(\mathcal{Y}\,F)
\end{aligned}$$

Encoding Recursion: the $\mathcal{Y}$ Combinator

- to encode recursion, we are looking for a combinator that, given an argument some function F, would not only reproduce itself but also pass F on itself.
- we can then define:

$$\mathcal{Y} := \lambda f.(\lambda x.f\,(x\,x))\,(\lambda x.f\,(x\,x))$$

- let us observe what happens when we pass a function F to the $\mathcal{Y}$ combinator:

$$\begin{aligned}
\mathcal{Y}\,F \quad &:= \quad \lambda f.(\lambda x.f\,(x\,x))\,(\lambda x.f\,(x\,x))\,F \\
&=_\beta \quad (\lambda x.F\,(x\,x))\,(\lambda x.F\,(x\,x)) \\
&=_\beta \quad F\Big(\underbrace{(\lambda x.F\,(x\,x))\,(\lambda x.F\,(x\,x))}_{\mathcal{Y}\,F}\Big) = F\,(\mathcal{Y}\,F) \\
&=_\beta \quad F\,(F\,(\mathcal{Y}\,F))
\end{aligned}$$

Encoding Recursion: the $\mathcal{Y}$ Combinator

- to encode recursion, we are looking for a combinator that, given an argument some function F, would not only reproduce itself but also pass F on itself.
- we can then define:

$$\mathcal{Y} \;:=\; \lambda f.(\lambda x.f\,(x\,x))\,(\lambda x.f\,(x\,x))$$

- let us observe what happens when we pass a function F to the $\mathcal{Y}$ combinator:

$$\begin{aligned}
\mathcal{Y}\,F \quad &:= \quad \lambda f.(\lambda x.f\,(x\,x))\,(\lambda x.f\,(x\,x))\,F \\
&=_\beta \quad (\lambda x.F\,(x\,x))\,(\lambda x.F\,(x\,x)) \\
&=_\beta \quad F\Big(\underbrace{(\lambda x.F\,(x\,x))\,(\lambda x.F\,(x\,x))}_{\mathcal{Y}\,F}\Big) = F\,(\mathcal{Y}\,F) \\
&=_\beta \quad F\,(F\,(\mathcal{Y}\,F)) \\
&=_\beta \quad F\,(F\,(F\,(\mathcal{Y}\,F)))
\end{aligned}$$

Encoding Recursion: the $\mathcal{Y}$ Combinator

- to encode recursion, we are looking for a combinator that, given an argument some function F, would not only reproduce itself but also pass F on itself.
- we can then define:

$$\mathcal{Y} := \lambda f.(\lambda x.f\,(x\,x))\,(\lambda x.f\,(x\,x))$$

- let us observe what happens when we pass a function F to the $\mathcal{Y}$ combinator:

$$\begin{aligned}
\mathcal{Y}\,F \quad &:= \quad \lambda f.(\lambda x.f\,(x\,x))\,(\lambda x.f\,(x\,x))\,F \\
&=_\beta \quad (\lambda x.F\,(x\,x))\,(\lambda x.F\,(x\,x)) \\
&=_\beta \quad F\Big(\underbrace{(\lambda x.F\,(x\,x))\,(\lambda x.F\,(x\,x))}_{\mathcal{Y}\,F}\Big) = F\,(\mathcal{Y}\,F) \\
&=_\beta \quad F\,(F\,(\mathcal{Y}\,F)) \\
&=_\beta \quad F\,(F\,(F\,(\mathcal{Y}\,F))) \\
&=_\beta \quad \ldots
\end{aligned}$$

Example ($\mathcal{Y}$ Combinator)

Let F be $\lambda f.\lambda x.(\text{if } x == 0 \text{ then } 1 \text{ else } x * f(x-1))$

Example ($\mathcal{Y}$ Combinator)

Let F be $\lambda f.\lambda x.(\text{if } x == 0 \text{ then } 1 \text{ else } x * f(x-1))$

$$\mathcal{Y}F\,3 \quad =^{+}_{\beta} \quad F\,(\mathcal{Y}F)\,3$$

Example ($\mathcal{Y}$ Combinator)

Let F be $\lambda f.\lambda x.(\text{if } x == 0 \text{ then } 1 \text{ else } x * f(x-1))$

$$
\begin{aligned}
\mathcal{Y} F\, 3 \quad & =_\beta^+ \quad F\,(\mathcal{Y} F)\, 3 \\
& := \quad \lambda f.\lambda x.(\text{if } x == 0 \text{ then } 1 \text{ else } x * f\,(x-1))\,(\mathcal{Y} F)\, 3
\end{aligned}
$$

Example ($\mathcal{Y}$ Combinator)

Let F be $\lambda f.\lambda x.(\text{if } x == 0 \text{ then } 1 \text{ else } x * f(x-1))$

$$
\begin{aligned}
\mathcal{Y} F\, 3 \quad & =_\beta^+ \quad F\,(\mathcal{Y} F)\,3 \\
& := \quad \lambda f.\lambda x.(\text{if } x == 0 \text{ then } 1 \text{ else } x * f\,(x-1))\,(\mathcal{Y} F)\,3 \\
& =_\beta \quad \lambda x.(\text{if } x == 0 \text{ then } 1 \text{ else } x * (\mathcal{Y} F)\,(x-1))\,3
\end{aligned}
$$

Example ($\mathcal{Y}$ Combinator)

Let F be $\lambda f.\lambda x.(\text{if } x == 0 \text{ then } 1 \text{ else } x * f(x-1))$

$$
\begin{aligned}
\mathcal{Y} F\, 3 \quad &=_\beta^+ \quad F\,(\mathcal{Y} F)\, 3 \\
&:= \quad \lambda f.\lambda x.(\text{if } x == 0 \text{ then } 1 \text{ else } x * f\,(x-1))\,(\mathcal{Y} F)\, 3 \\
&=_\beta \quad \lambda x.(\text{if } x == 0 \text{ then } 1 \text{ else } x * (\mathcal{Y} F)\,(x-1))\, 3 \\
&=_\beta \quad \text{if } 3 == 0 \text{ then } 1 \text{ else } 3 * (\mathcal{Y} F)\,(3-1)
\end{aligned}
$$

Example ($\mathcal{Y}$ Combinator)

Let F be $\lambda f.\lambda x.(\text{if } x == 0 \text{ then } 1 \text{ else } x * f(x-1))$

$$
\begin{aligned}
\mathcal{Y}F\,3 \quad &=_\beta^+ \quad F\,(\mathcal{Y}F)\,3 \\
&:= \quad \lambda f.\lambda x.(\text{if } x == 0 \text{ then } 1 \text{ else } x * f\,(x-1))\,(\mathcal{Y}F)\,3 \\
&=_\beta \quad \lambda x.(\text{if } x == 0 \text{ then } 1 \text{ else } x * (\mathcal{Y}F)\,(x-1))\,3 \\
&=_\beta \quad \text{if } 3 == 0 \text{ then } 1 \text{ else } 3 * (\mathcal{Y}F)\,(3-1) \\
&=_\beta \quad 3 * (\mathcal{Y}F)\,2
\end{aligned}
$$

Example ($\mathcal{Y}$ Combinator)

Let F be $\lambda f.\lambda x.(\text{if } x == 0 \text{ then } 1 \text{ else } x * f(x-1))$

$$\begin{aligned}
\mathcal{Y}F\,3 \quad &=^{+}_{\beta} \quad F\,(\mathcal{Y}F)\,3 \\
&:= \quad \lambda f.\lambda x.(\text{if } x == 0 \text{ then } 1 \text{ else } x * f\,(x-1))\,(\mathcal{Y}F)\,3 \\
&=_{\beta} \quad \lambda x.(\text{if } x == 0 \text{ then } 1 \text{ else } x * (\mathcal{Y}F)\,(x-1))\,3 \\
&=_{\beta} \quad \text{if } 3 == 0 \text{ then } 1 \text{ else } 3 * (\mathcal{Y}F)\,(3-1) \\
&=_{\beta} \quad 3 * (\mathcal{Y}F)\,2 \\
&=^{+}_{\beta} \quad 3 * F\,(\mathcal{Y}F)\,2
\end{aligned}$$

Example ($\mathcal{Y}$ Combinator)

Let F be $\lambda f.\lambda x.(\text{if } x == 0 \text{ then } 1 \text{ else } x * f(x-1))$

$$
\begin{aligned}
\mathcal{Y}F\,3 \quad &=_\beta^+ \quad F\,(\mathcal{Y}F)\,3 \\
&:= \quad \lambda f.\lambda x.(\text{if } x == 0 \text{ then } 1 \text{ else } x * f\,(x-1))\,(\mathcal{Y}F)\,3 \\
&=_\beta \quad \lambda x.(\text{if } x == 0 \text{ then } 1 \text{ else } x * (\mathcal{Y}F)\,(x-1))\,3 \\
&=_\beta \quad \text{if } 3 == 0 \text{ then } 1 \text{ else } 3 * (\mathcal{Y}F)\,(3-1) \\
&=_\beta \quad 3 * (\mathcal{Y}F)\,2 \\
&=_\beta^+ \quad 3 * F\,(\mathcal{Y}F)\,2 \\
&=_\beta \quad 3 * (\lambda f.\lambda x.(\text{if } x == 0 \text{ then } 1 \text{ else } x * f\,(x-1))\,(\mathcal{Y}F)\,2)
\end{aligned}
$$

Example ($\mathcal{Y}$ Combinator)

Let F be $\lambda f.\lambda x.(\text{if } x == 0 \text{ then } 1 \text{ else } x * f(x-1))$

$$
\begin{array}{lll}
\mathcal{Y}F\,3 & =^{+}_{\beta} & F\,(\mathcal{Y}F)\,3 \\
 & := & \lambda f.\lambda x.(\text{if } x == 0 \text{ then } 1 \text{ else } x * f\,(x-1))\,(\mathcal{Y}F)\,3 \\
 & =_{\beta} & \lambda x.(\text{if } x == 0 \text{ then } 1 \text{ else } x * (\mathcal{Y}F)\,(x-1))\,3 \\
 & =_{\beta} & \text{if } 3 == 0 \text{ then } 1 \text{ else } 3 * (\mathcal{Y}F)\,(3-1) \\
 & =_{\beta} & 3 * (\mathcal{Y}F)\,2 \\
 & =^{+}_{\beta} & 3 * F\,(\mathcal{Y}F)\,2 \\
 & =_{\beta} & 3 * (\lambda f.\lambda x.(\text{if } x == 0 \text{ then } 1 \text{ else } x * f\,(x-1))\,(\mathcal{Y}F)\,2) \\
 & =_{\beta} & 3 * (\lambda x.(\text{if } x == 0 \text{ then } 1 \text{ else } x * (\mathcal{Y}F)\,(x-1))\,2)
\end{array}
$$

Example ($\mathcal{Y}$ Combinator)

Let F be $\lambda f.\lambda x.(\text{if } x == 0 \text{ then } 1 \text{ else } x * f(x-1))$

$$
\begin{aligned}
\mathcal{Y} F\, 3 \quad & =_\beta^+ \quad F\,(\mathcal{Y} F)\, 3 \\
& := \quad \lambda f.\lambda x.(\text{if } x == 0 \text{ then } 1 \text{ else } x * f\,(x-1))\,(\mathcal{Y} F)\, 3 \\
& =_\beta \quad \lambda x.(\text{if } x == 0 \text{ then } 1 \text{ else } x * (\mathcal{Y} F)\,(x-1))\, 3 \\
& =_\beta \quad \text{if } 3 == 0 \text{ then } 1 \text{ else } 3 * (\mathcal{Y} F)\,(3-1) \\
& =_\beta \quad 3 * (\mathcal{Y} F)\, 2 \\
& =_\beta^+ \quad 3 * F\,(\mathcal{Y} F)\, 2 \\
& =_\beta \quad 3 * (\lambda f.\lambda x.(\text{if } x == 0 \text{ then } 1 \text{ else } x * f\,(x-1))\,(\mathcal{Y} F)\, 2) \\
& =_\beta \quad 3 * (\lambda x.(\text{if } x == 0 \text{ then } 1 \text{ else } x * (\mathcal{Y} F)\,(x-1))\, 2) \\
& =_\beta \quad 3 * (\text{if } 2 == 0 \text{ then } 1 \text{ else } 2 * (\mathcal{Y} F)\,(2-1))
\end{aligned}
$$

Example ($\mathcal{Y}$ Combinator)

Let F be $\lambda f.\lambda x.(\text{if } x == 0 \text{ then } 1 \text{ else } x * f(x-1))$

$$
\begin{array}{lll}
\mathcal{Y} F\, 3 & =_\beta^+ & F\,(\mathcal{Y} F)\, 3 \\
 & := & \lambda f.\lambda x.(\text{if } x == 0 \text{ then } 1 \text{ else } x * f\,(x-1))\,(\mathcal{Y} F)\, 3 \\
 & =_\beta & \lambda x.(\text{if } x == 0 \text{ then } 1 \text{ else } x * (\mathcal{Y} F)\,(x-1))\, 3 \\
 & =_\beta & \text{if } 3 == 0 \text{ then } 1 \text{ else } 3 * (\mathcal{Y} F)\,(3-1) \\
 & =_\beta & 3 * (\mathcal{Y} F)\, 2 \\
 & =_\beta^+ & 3 * F\,(\mathcal{Y} F)\, 2 \\
 & =_\beta & 3 * (\lambda f.\lambda x.(\text{if } x == 0 \text{ then } 1 \text{ else } x * f\,(x-1))\,(\mathcal{Y} F)\, 2) \\
 & =_\beta & 3 * (\lambda x.(\text{if } x == 0 \text{ then } 1 \text{ else } x * (\mathcal{Y} F)\,(x-1))\, 2) \\
 & =_\beta & 3 * (\text{if } 2 == 0 \text{ then } 1 \text{ else } 2 * (\mathcal{Y} F)\,(2-1)) \\
 & =_\beta & 3 * 2 * (\mathcal{Y} F)\, 1
\end{array}
$$

Example ($\mathcal{Y}$ Combinator)

Let F be $\lambda f.\lambda x.(\text{if } x == 0 \text{ then } 1 \text{ else } x * f(x-1))$

$$
\begin{array}{lll}
\mathcal{Y}F\,3 & =_\beta^+ & F\,(\mathcal{Y}F)\,3 \\
 & := & \lambda f.\lambda x.(\text{if } x == 0 \text{ then } 1 \text{ else } x * f\,(x-1))\,(\mathcal{Y}F)\,3 \\
 & =_\beta & \lambda x.(\text{if } x == 0 \text{ then } 1 \text{ else } x * (\mathcal{Y}F)\,(x-1))\,3 \\
 & =_\beta & \text{if } 3 == 0 \text{ then } 1 \text{ else } 3 * (\mathcal{Y}F)\,(3-1) \\
 & =_\beta & 3 * (\mathcal{Y}F)\,2 \\
 & =_\beta^+ & 3 * F\,(\mathcal{Y}F)\,2 \\
 & =_\beta & 3 * (\lambda f.\lambda x.(\text{if } x == 0 \text{ then } 1 \text{ else } x * f\,(x-1))\,(\mathcal{Y}F)\,2) \\
 & =_\beta & 3 * (\lambda x.(\text{if } x == 0 \text{ then } 1 \text{ else } x * (\mathcal{Y}F)\,(x-1))\,2) \\
 & =_\beta & 3 * (\text{if } 2 == 0 \text{ then } 1 \text{ else } 2 * (\mathcal{Y}F)\,(2-1)) \\
 & =_\beta & 3 * 2 * (\mathcal{Y}F)\,1 \\
 & =_\beta & 6 * (\mathcal{Y}F)\,1
\end{array}
$$

Example ($\mathcal{Y}$ Combinator)

Let F be $\lambda f.\lambda x.(\text{if } x == 0 \text{ then } 1 \text{ else } x * f(x-1))$

$$
\begin{aligned}
\mathcal{Y} F\, 3 &=_\beta^+ && F\,(\mathcal{Y} F)\, 3 \\
&:= && \lambda f.\lambda x.(\text{if } x == 0 \text{ then } 1 \text{ else } x * f\,(x-1))\,(\mathcal{Y} F)\, 3 \\
&=_\beta && \lambda x.(\text{if } x == 0 \text{ then } 1 \text{ else } x * (\mathcal{Y} F)\,(x-1))\, 3 \\
&=_\beta && \text{if } 3 == 0 \text{ then } 1 \text{ else } 3 * (\mathcal{Y} F)\,(3-1) \\
&=_\beta && 3 * (\mathcal{Y} F)\, 2 \\
&=_\beta^+ && 3 * F\,(\mathcal{Y} F)\, 2 \\
&=_\beta && 3 * (\lambda f.\lambda x.(\text{if } x == 0 \text{ then } 1 \text{ else } x * f\,(x-1))\,(\mathcal{Y} F)\, 2) \\
&=_\beta && 3 * (\lambda x.(\text{if } x == 0 \text{ then } 1 \text{ else } x * (\mathcal{Y} F)\,(x-1))\, 2) \\
&=_\beta && 3 * (\text{if } 2 == 0 \text{ then } 1 \text{ else } 2 * (\mathcal{Y} F)\,(2-1)) \\
&=_\beta && 3 * 2 * (\mathcal{Y} F)\, 1 \\
&=_\beta && 6 * (\mathcal{Y} F)\, 1 \\
&=_\beta^+ && 6 * F\,(\mathcal{Y} F)\, 1
\end{aligned}
$$

Example ($\mathcal{Y}$ Combinator (cont'd))

$$6 * F\,(\mathcal{Y} F)\,1$$

Example ($\mathcal{Y}$ Combinator (cont'd))

$$
\begin{aligned}
& 6 * F\,(\mathcal{Y} F)\,1 \\
=_\beta \quad & 6 * (\lambda f.\lambda x.(\text{if } x == 0 \text{ then } 1 \text{ else } x * f\,(x-1))\,(\mathcal{Y} F)\,1)
\end{aligned}
$$

Example ($\mathcal{Y}$ Combinator (cont'd))

$$
\begin{aligned}
& 6 * F\,(\mathcal{Y} F)\,1 \\
=_\beta \quad & 6 * (\lambda f.\lambda x.(\text{if } x == 0 \text{ then } 1 \text{ else } x * f\,(x-1))\,(\mathcal{Y} F)\,1) \\
=_\beta \quad & 6 * (\lambda x.(\text{if } x == 0 \text{ then } 1 \text{ else } x * (\mathcal{Y} F)\,(x-1))\,1)
\end{aligned}
$$

Example ($\mathcal{Y}$ Combinator (cont'd))

$$\begin{aligned}
& 6 * F\,(\mathcal{Y} F)\,1 \\
=_\beta\ & 6 * (\lambda f.\lambda x.(\text{if } x == 0 \text{ then } 1 \text{ else } x * f\,(x-1))\,(\mathcal{Y} F)\,1) \\
=_\beta\ & 6 * (\lambda x.(\text{if } x == 0 \text{ then } 1 \text{ else } x * (\mathcal{Y} F)\,(x-1))\,1) \\
=_\beta\ & 6 * (\text{if } 1 == 0 \text{ then } 1 \text{ else } 1 * (\mathcal{Y} F)\,(1-1))
\end{aligned}$$

Example ($\mathcal{Y}$ Combinator (cont'd))

$$
\begin{aligned}
& 6 * F\,(\mathcal{Y}F)\,1 \\
=_\beta\; & 6 * (\lambda f.\lambda x.(\text{if } x == 0 \text{ then } 1 \text{ else } x * f\,(x-1))\,(\mathcal{Y}F)\,1) \\
=_\beta\; & 6 * (\lambda x.(\text{if } x == 0 \text{ then } 1 \text{ else } x * (\mathcal{Y}F)\,(x-1))\,1) \\
=_\beta\; & 6 * (\text{if } 1 == 0 \text{ then } 1 \text{ else } 1 * (\mathcal{Y}F)\,(1-1)) \\
=_\beta\; & 6 * (\mathcal{Y}F)\,0
\end{aligned}
$$

Example ($\mathcal{Y}$ Combinator (cont'd))

$$
\begin{aligned}
& 6 * F\,(\mathcal{Y}F)\,1 \\
=_\beta\ & 6 * (\lambda f.\lambda x.(\text{if } x == 0 \text{ then } 1 \text{ else } x * f\,(x-1))\,(\mathcal{Y}F)\,1) \\
=_\beta\ & 6 * (\lambda x.(\text{if } x == 0 \text{ then } 1 \text{ else } x * (\mathcal{Y}F)\,(x-1))\,1) \\
=_\beta\ & 6 * (\text{if } 1 == 0 \text{ then } 1 \text{ else } 1 * (\mathcal{Y}F)\,(1-1)) \\
=_\beta\ & 6 * (\mathcal{Y}F)\,0 \\
=_\beta^+\ & 6 * F\,(\mathcal{Y}F)\,0
\end{aligned}
$$

Example ($\mathcal{Y}$ Combinator (cont'd))

$$
\begin{aligned}
& 6 * F\,(\mathcal{Y} F)\,1 \\
=_\beta\;& 6 * (\lambda f.\lambda x.(\text{if } x == 0 \text{ then } 1 \text{ else } x * f\,(x-1))\,(\mathcal{Y} F)\,1) \\
=_\beta\;& 6 * (\lambda x.(\text{if } x == 0 \text{ then } 1 \text{ else } x * (\mathcal{Y} F)\,(x-1))\,1) \\
=_\beta\;& 6 * (\text{if } 1 == 0 \text{ then } 1 \text{ else } 1 * (\mathcal{Y} F)\,(1-1)) \\
=_\beta\;& 6 * (\mathcal{Y} F)\,0 \\
=_\beta^+\;& 6 * F\,(\mathcal{Y} F)\,0 \\
=_\beta\;& 6 * (\lambda f.\lambda x.(\text{if } x == 0 \text{ then } 1 \text{ else } x * f\,(x-1))\,(\mathcal{Y} F)\,0)
\end{aligned}
$$

Example ($\mathcal{Y}$ Combinator (cont'd))

$$
\begin{aligned}
& 6 * F\,(\mathcal{Y} F)\,1 \\
=_\beta\ & 6 * (\lambda f.\lambda x.(\text{if } x == 0 \text{ then } 1 \text{ else } x * f\,(x-1))\,(\mathcal{Y} F)\,1) \\
=_\beta\ & 6 * (\lambda x.(\text{if } x == 0 \text{ then } 1 \text{ else } x * (\mathcal{Y} F)\,(x-1))\,1) \\
=_\beta\ & 6 * (\text{if } 1 == 0 \text{ then } 1 \text{ else } 1 * (\mathcal{Y} F)\,(1-1)) \\
=_\beta\ & 6 * (\mathcal{Y} F)\,0 \\
=_\beta^+\ & 6 * F\,(\mathcal{Y} F)\,0 \\
=_\beta\ & 6 * (\lambda f.\lambda x.(\text{if } x == 0 \text{ then } 1 \text{ else } x * f\,(x-1))\,(\mathcal{Y} F)\,0) \\
=_\beta\ & 6 * (\lambda x.(\text{if } x == 0 \text{ then } 1 \text{ else } x * (\mathcal{Y} F)\,(x-1))\,0)
\end{aligned}
$$

Example ($\mathcal{Y}$ Combinator (cont'd))

$$
\begin{array}{ll}
 & 6 * F\,(\mathcal{Y}\,F)\,1 \\
=_\beta & 6 * (\lambda f.\lambda x.(\text{if } x == 0 \text{ then } 1 \text{ else } x * f\,(x-1))\,(\mathcal{Y}\,F)\,1) \\
=_\beta & 6 * (\lambda x.(\text{if } x == 0 \text{ then } 1 \text{ else } x * (\mathcal{Y}\,F)\,(x-1))\,1) \\
=_\beta & 6 * (\text{if } 1 == 0 \text{ then } 1 \text{ else } 1 * (\mathcal{Y}\,F)\,(1-1)) \\
=_\beta & 6 * (\mathcal{Y}\,F)\,0 \\
=_\beta^+ & 6 * F\,(\mathcal{Y}\,F)\,0 \\
=_\beta & 6 * (\lambda f.\lambda x.(\text{if } x == 0 \text{ then } 1 \text{ else } x * f\,(x-1))\,(\mathcal{Y}\,F)\,0) \\
=_\beta & 6 * (\lambda x.(\text{if } x == 0 \text{ then } 1 \text{ else } x * (\mathcal{Y}\,F)\,(x-1))\,0) \\
=_\beta & 6 * (\text{if } 0 == 0 \text{ then } 1 \text{ else } 1 * (\mathcal{Y}\,F)\,(0-1))
\end{array}
$$

Example ($\mathcal{Y}$ Combinator (cont'd))

$$
\begin{aligned}
& 6 * F\,(\mathcal{Y} F)\,1 \\
=_\beta\; & 6 * (\lambda f.\lambda x.(\text{if } x == 0 \text{ then } 1 \text{ else } x * f\,(x-1))\,(\mathcal{Y} F)\,1) \\
=_\beta\; & 6 * (\lambda x.(\text{if } x == 0 \text{ then } 1 \text{ else } x * (\mathcal{Y} F)\,(x-1))\,1) \\
=_\beta\; & 6 * (\text{if } 1 == 0 \text{ then } 1 \text{ else } 1 * (\mathcal{Y} F)\,(1-1)) \\
=_\beta\; & 6 * (\mathcal{Y} F)\,0 \\
=_\beta^+\; & 6 * F\,(\mathcal{Y} F)\,0 \\
=_\beta\; & 6 * (\lambda f.\lambda x.(\text{if } x == 0 \text{ then } 1 \text{ else } x * f\,(x-1))\,(\mathcal{Y} F)\,0) \\
=_\beta\; & 6 * (\lambda x.(\text{if } x == 0 \text{ then } 1 \text{ else } x * (\mathcal{Y} F)\,(x-1))\,0) \\
=_\beta\; & 6 * (\text{if } 0 == 0 \text{ then } 1 \text{ else } 1 * (\mathcal{Y} F)\,(0-1)) \\
=_\beta\; & 6 * 1
\end{aligned}
$$

Example ($\mathcal{Y}$ Combinator (cont'd))

$$
\begin{aligned}
& 6 * F\,(\mathcal{Y}F)\,1 \\
=_\beta\ & 6 * (\lambda f.\lambda x.(\text{if } x == 0 \text{ then } 1 \text{ else } x * f\,(x-1))\,(\mathcal{Y}F)\,1) \\
=_\beta\ & 6 * (\lambda x.(\text{if } x == 0 \text{ then } 1 \text{ else } x * (\mathcal{Y}F)\,(x-1))\,1) \\
=_\beta\ & 6 * (\text{if } 1 == 0 \text{ then } 1 \text{ else } 1 * (\mathcal{Y}F)\,(1-1)) \\
=_\beta\ & 6 * (\mathcal{Y}F)\,0 \\
=_\beta^+\ & 6 * F\,(\mathcal{Y}F)\,0 \\
=_\beta\ & 6 * (\lambda f.\lambda x.(\text{if } x == 0 \text{ then } 1 \text{ else } x * f\,(x-1))\,(\mathcal{Y}F)\,0) \\
=_\beta\ & 6 * (\lambda x.(\text{if } x == 0 \text{ then } 1 \text{ else } x * (\mathcal{Y}F)\,(x-1))\,0) \\
=_\beta\ & 6 * (\text{if } 0 == 0 \text{ then } 1 \text{ else } 1 * (\mathcal{Y}F)\,(0-1)) \\
=_\beta\ & 6 * 1 \\
=_\beta\ & 6
\end{aligned}
$$

Outline

1. λ-Calculus
2. Programming In λ-Calculus
3. Typing In General
4. STLC($\lambda^{\rightarrow}$)

Need for Types

- In (untyped) λ-Calculus, we can easily misuse terms:

$$
\begin{array}{lcl}
\text{false} & := & \lambda x.\lambda y.y \\
0 & := & \lambda s.\lambda z.z
\end{array}
$$

Need for Types

- In (untyped) λ-Calculus, we can easily misuse terms:

$$\begin{array}{lll} \text{false} & := & \lambda x.\lambda y.y \\ 0 & := & \lambda s.\lambda z.z \\ \text{false } 0 & =_\beta & \lambda y.y \end{array}$$

Need for Types

- In (untyped) λ-Calculus, we can easily misuse terms:

$$\begin{array}{lll} \text{false} & := & \lambda x.\lambda y.y \\ 0 & := & \lambda s.\lambda z.z \\ \text{false } 0 & =_\beta & \lambda y.y \end{array}$$

- there is an obvious need to guarantee that function parameters are “valid” in term of function application to prevent errors during the evaluation

Need for Types

- In (untyped) λ-Calculus, we can easily misuse terms:

$$\begin{array}{lll} \text{false} & := & \lambda x.\lambda y.y \\ 0 & := & \lambda s.\lambda z.z \\ \text{false } 0 & =_\beta & \lambda y.y \end{array}$$

- there is an obvious need to guarantee that function parameters are “valid” in term of function application to prevent errors during the evaluation
- this is in fact the fundamental purpose of type systems

Type Systems in General

- type system is a set of rules that assigns a property called a type to the terms (perhaps to other various constructs) in a program with a purpose to reduce possibilities for bugs, and evaluation errors

Type Systems in General

- type system is a set of rules that assigns a property called a type to the terms (perhaps to other various constructs) in a program with a purpose to reduce possibilities for bugs, and evaluation errors
- some mechanism to distinguish "good" and "bad" programs

0 + 1	is well-typed	good
0 + false	is ill-typed	bad
if false then 10 else 20	is well-typed	good
1 + (if true then 10 else false)	is ill-typed	bad

Type Systems in General (cont'd)

- main point is to classify terms into types
- given a set of (inductively generated) types

$$Ty \quad := \quad T_1 \mid T_2 \mid T_3 \mid \ldots$$

- a term t might be of type T_1, T_2, T_3, ...

Type Systems in General (cont'd)

- main point is to classify terms into types
- given a set of (inductively generated) types

$$Ty \quad := \quad T_1 \mid T_2 \mid T_3 \mid \ldots$$

- a term t might be of type T_1, T_2, T_3, ... (thanks to a typing relation)

Typing Relations in General

- typing relation ":" assigns types to terms

Typing Relations in General

- typing relation ":" assigns types to terms
- formally, a typing relation is a partial binary predicate ":" $: \mathcal{E} \times \mathcal{T}y \rightarrow Bool$ where
  - $\mathcal{E}$ is the set of all possible terms
  - $\mathcal{T}y$ is the set of all possible types

Typing Relations in General

- typing relation “:” assigns types to terms
- formally, a typing relation is a partial binary predicate “:” $: \mathcal{E} \times \mathcal{T}y \rightarrow \mathit{Bool}$ where
  - $\mathcal{E}$ is the set of all possible terms
  - $\mathcal{T}y$ is the set of all possible types
- some related notions:

language	:=	as a set $\mathcal{E}$ of all possible terms
type language	:=	as a set $\mathcal{T}y$ of all possible types
typing relation	:=	as a partial relation “:” $\subseteq \mathcal{E} \times \mathcal{T}y$

Typing Relations in General

- typing relation ":" assigns types to terms
- formally, a typing relation is a partial binary predicate ":" $: \mathcal{E} \times \mathcal{T}y \rightarrow \mathit{Bool}$ where
  - $\mathcal{E}$ is the set of all possible terms
  - $\mathcal{T}y$ is the set of all possible types
- some related notions:

$$\begin{array}{lcl} \text{language} & := & \text{as a set } \mathcal{E} \text{ of all possible terms} \\ \text{type language} & := & \text{as a set } \mathcal{T}y \text{ of all possible types} \\ \text{typing relation} & := & \text{as a partial relation ``:''} \subseteq \mathcal{E} \times \mathcal{T}y \end{array}$$

- categorical approach is slightly different:

$$\begin{array}{lcl} \text{language} & := & \text{internal language of a certain category } \mathcal{C} \\ \text{types} & := & \text{objects of } \mathcal{C} \\ \text{terms} & := & \text{arrows of } \mathcal{C} \end{array}$$

Outline

1. λ-Calculus
2. Programming In λ-Calculus
3. Typing In General
4. STLC($\lambda^{\rightarrow}$)

Definition (Simply Typed λ-Calculus ($\lambda^{\rightarrow}$))

Types $A, B, C, \ldots$:=

$\mid$	$G, G', G'', \ldots$	"ground" types
$\mid$	unit	unit type
$\mid$	$A \times B$	product type
$\mid$	$A \rightarrow B$	function type

Definition (Simply Typed λ-Calculus ($\lambda^{\to}$))

Types $A, B, C, \ldots :=$

$\mid$	$G, G', G'', \ldots$	"ground" types
$\mid$	unit	unit type
$\mid$	$A \times B$	product type
$\mid$	$A \to B$	function type

Terms $s, t, r :=$

$\mid$	c^A	constants (of given type A)
$\mid$	x	variable (countable many)
$\mid$	$()$	unit value
$\mid$	(s, t)	pair
$\mid$	fst t	first pair projection
$\mid$	snd t	second pair projection
$\mid$	$\lambda x\colon A.\, t$	function abstraction
$\mid$	$s\ t$	function application

Example (term examples)

- $\lambda z : (A \to B) \times (A \to C).\, \lambda x : A.\, ((\text{fst } z)\ x, (\text{snd } z)\ x)$

Example (term examples)

- $\lambda z : (A \to B) \times (A \to C).\, \lambda x : A.\, ((\mathrm{fst}\ z)\ x, (\mathrm{snd}\ z)\ x)$
- $\lambda z : A \to (B \times C).\, (\lambda x : A.\, \mathrm{fst}\ (z\ x), \lambda y : A.\, \mathrm{snd}\ (z\ y))$

Example (term examples)

- $\lambda z : (A \to B) \times (A \to C).\, \lambda x : A.\, ((\mathsf{fst}\ z)\ x, (\mathsf{snd}\ z)\ x)$
- $\lambda z : A \to (B \times C).\, (\lambda x : A.\, \mathsf{fst}\ (z\ x), \lambda y : A.\, \mathsf{snd}\ (z\ y))$
- $\lambda z : A \to (B \times C).\, \lambda x : A.\, ((\mathsf{fst}\ z)\ x, (\mathsf{snd}\ z)\ x)$

Definition ($\lambda^{\rightarrow}$ typing relation: $\Gamma \vdash t : A$)

Γ ranges over typing environments (or typing contexts)

$$
\begin{array}{lll}
\Gamma := & & \\
\mid & [\,] & \text{“empty” environment} \\
\mid & \Gamma, x : A & \text{“non-empty” environment}
\end{array}
$$

Definition ($\lambda^{\rightarrow}$ typing relation: $\Gamma \vdash t : A$)

Γ ranges over typing environments (or typing contexts)

$$
\begin{array}{lll}
\Gamma := & & \\
\quad | & [\,] & \text{“empty” environment} \\
\quad | & \Gamma, x : A & \text{“non-empty” environment}
\end{array}
$$

typing environments are comma-separated snoc-lists of (variable,type)-pairs – in fact only the lists whose variables are mutually distinct get used

Definition ($\lambda^{\rightarrow}$ typing relation: $\Gamma \vdash t : A$)

Γ ranges over typing environments (or typing contexts)

$$
\begin{array}{lll}
\Gamma := & & \\
\quad | & [\,] & \text{“empty” environment} \\
\quad | & \Gamma, x : A & \text{“non-empty” environment}
\end{array}
$$

typing environments are comma-separated snoc-lists of (variable,type)-pairs – in fact only the lists whose variables are mutually distinct get used

Notation

- Γ ok means that no variable occurs more than once in Γ

Definition ($\lambda^{\to}$ typing relation: $\Gamma \vdash t : A$)

Γ ranges over typing environments (or typing contexts)

$$
\begin{array}{rlll}
\Gamma := & & & \\
 & | & [\,] & \text{"empty" environment} \\
 & | & \Gamma, x : A & \text{"non-empty" environment}
\end{array}
$$

typing environments are comma-separated snoc-lists of (variable,type)-pairs – in fact only the lists whose variables are mutually distinct get used

Notation

- Γ ok means that no variable occurs more than once in Γ
- dom Γ denotes the finite set of variables occurring in Γ

Definition ($\lambda^{\rightarrow}$ typing relation: $\Gamma \vdash t : A$ (cont'd))

$$\frac{\Gamma \text{ ok} \qquad x \notin \text{dom}\,\Gamma}{\Gamma, x : A \vdash x : A}\ (\text{var}) \qquad \frac{\Gamma \vdash x : A \qquad x' \notin \text{dom}\,\Gamma}{\Gamma, x' : A \vdash x : A}\ (\text{var'}) \qquad \frac{\Gamma \text{ ok}}{\Gamma \vdash c^A : A}\ (\text{const})$$

$$\frac{\Gamma \text{ ok}}{\Gamma \vdash () : \text{unit}}\ (\text{unit}) \qquad \frac{\Gamma \vdash s : A \qquad \Gamma \vdash t : B}{\Gamma \vdash (s, t) : A \times B}\ (\text{pair}) \qquad \frac{\Gamma \vdash t : A \times B}{\Gamma \vdash \text{fst}\ t : A}\ (\text{fstT})$$

$$\frac{\Gamma \vdash t : A \times B}{\Gamma \vdash \text{snd}\ t : B}\ (\text{sndT}) \qquad \frac{\Gamma, x : A \vdash t : B}{\Gamma \vdash \lambda x : A.\, t : A \rightarrow B}\ (\text{fun}) \qquad \frac{\Gamma \vdash s : A \rightarrow B \qquad \Gamma \vdash t : A}{\Gamma \vdash s\ t : B}\ (\text{app})$$

Example (term examples)

- $[\,] \vdash \lambda z : (A \rightarrow B) \times (A \rightarrow C).\, \lambda x : A.\, ((\mathsf{fst}\ z)\ x, (\mathsf{snd}\ z)\ x)$

Example (term examples)

- $[\,] \vdash \lambda z : (A \rightarrow B) \times (A \rightarrow C).\, \lambda x : A.\, ((\mathrm{fst}\ z)\ x, (\mathrm{snd}\ z)\ x)$ has type $((A \rightarrow B) \times (A \rightarrow C) \rightarrow (A \rightarrow (B \times C)))$

Example (term examples)

- $[\,] \vdash \lambda z\colon (A \to B) \times (A \to C).\, \lambda x\colon A.\, ((\mathrm{fst}\ z)\ x, (\mathrm{snd}\ z)\ x)$ has type $((A \to B) \times (A \to C) \to (A \to (B \times C)))$
- $[\,] \vdash \lambda z\colon A \to (B \times C).\, (\lambda x\colon A.\, \mathrm{fst}\ (z\ x), \lambda y\colon A.\, \mathrm{snd}\ (z\ y))$ has type

Example (term examples)

- $[\,] \vdash \lambda z\colon (A \rightarrow B) \times (A \rightarrow C).\, \lambda x\colon A.\, ((\mathsf{fst}\ z)\ x, (\mathsf{snd}\ z)\ x)$ has type $((A \rightarrow B) \times (A \rightarrow C) \rightarrow (A \rightarrow (B \times C)))$
- $[\,] \vdash \lambda z\colon A \rightarrow (B \times C).\, (\lambda x\colon A.\, \mathsf{fst}\ (z\ x), \lambda y\colon A.\, \mathsf{snd}\ (z\ y))$ has type $(A \rightarrow (B \times C)) \rightarrow ((A \rightarrow B) \times (A \rightarrow C))$

Example (term examples)

- $[\,] \vdash \lambda z\colon (A \to B) \times (A \to C).\, \lambda x\colon A.\, ((\mathrm{fst}\ z)\ x, (\mathrm{snd}\ z)\ x)$ has type $((A \to B) \times (A \to C) \to (A \to (B \times C)))$
- $[\,] \vdash \lambda z\colon A \to (B \times C).\, (\lambda x\colon A.\, \mathrm{fst}\ (z\ x), \lambda y\colon A.\, \mathrm{snd}\ (z\ y))$ has type $(A \to (B \times C)) \to ((A \to B) \times (A \to C))$
- $[\,] \vdash \lambda z\colon A \to (B \times C).\, \lambda x\colon A.\, ((\mathrm{fst}\ z)\ x, (\mathrm{snd}\ z)\ x)$

Example (term examples)

- $[\,] \vdash \lambda z\colon (A \to B) \times (A \to C).\, \lambda x\colon A.\, ((\mathrm{fst}\ z)\ x, (\mathrm{snd}\ z)\ x)$ has type $((A \to B) \times (A \to C) \to (A \to (B \times C)))$
- $[\,] \vdash \lambda z\colon A \to (B \times C).\, (\lambda x\colon A.\, \mathrm{fst}\ (z\ x), \lambda y\colon A.\, \mathrm{snd}\ (z\ y))$ has type $(A \to (B \times C)) \to ((A \to B) \times (A \to C))$
- $[\,] \vdash \lambda z\colon A \to (B \times C).\, \lambda x\colon A.\, ((\mathrm{fst}\ z)\ x, (\mathrm{snd}\ z)\ x)$ has no type (ill-typed term)

Example (typing derivation)

in a typing context $\Gamma = [\,], f : A \to B, g : B \to C$, we have an example derivation of a term $s : A \to C$ as follows:

$$
\dfrac{
\dfrac{
\dfrac{\dfrac{}{\Gamma \vdash g : B \to C}\,(\text{var})}{\Gamma, x : A \vdash g : B \to C}\,(\text{var'})
\qquad
\dfrac{
\dfrac{\dfrac{\dfrac{}{[\,], f : A \to B \vdash f : A \to B}\,(\text{var})}{\Gamma \vdash f : A \to B}\,(\text{var'})}{\Gamma, x : A \vdash f : A \to B}\,(\text{var'})
\quad
\dfrac{}{\Gamma, x : A \vdash x : A}\,(\text{var})
}{\Gamma, x : A \vdash f\ x : B}\,(\text{app})
}{\Gamma, x : A \vdash g\ (f\ x) : C}\,(\text{app})
}{\Gamma \vdash \lambda x : A.\, g\ (f\ x) : A \to C}\,(\text{fun})
$$

Example (typing derivation)

in a typing context $\Gamma = [\,], f : A \to B, g : B \to C$, we have an example derivation of a term $s : A \to C$ as follows:

$$
\dfrac{
  \dfrac{
    \dfrac{}{\Gamma \vdash g : B \to C}\,(\text{var})
  }{\Gamma, x : A \vdash g : B \to C}\,(\text{var'})
  \quad
  \dfrac{
    \dfrac{
      \dfrac{
        \dfrac{}{[\,], f : A \to B \vdash f : A \to B}\,(\text{var})
      }{\Gamma \vdash f : A \to B}\,(\text{var'})
    }{\Gamma, x : A \vdash f : A \to B}\,(\text{var'})
    \quad
    \dfrac{}{\Gamma, x : A \vdash x : A}\,(\text{var})
  }{\Gamma, x : A \vdash f\ x : B}\,(\text{app})
}{
  \dfrac{\Gamma, x : A \vdash g\ (f\ x) : C}{\Gamma \vdash \lambda x : A.\, g\ (f\ x) : A \to C}\,(\text{fun})
}\,(\text{app})
$$

Remark

the $\lambda^{\rightarrow}$ typing rules are “syntax-directed”, by the structure of terms t and then in the case of variables x, by the structure of typing environments Γ.

Definition (α-equivalence)

- names of λ-bound variables should not affect meaning

Definition (α-equivalence)

- names of λ-bound variables should not affect meaning
- e.g., $\lambda f : A \to B.\, \lambda x : A.\, f\ x$ should have the same meaning as $\lambda x : A \to B.\, \lambda y : A.\, x\ y$

Definition (α-equivalence)

- names of λ-bound variables should not affect meaning
- e.g., $\lambda f : A \to B.\, \lambda x : A.\, f\ x$ should have the same meaning as $\lambda x : A \to B.\, \lambda y : A.\, x\ y$
- this issue is best dealt with at the level of syntax rather than semantics

Definition (α-equivalence)

- names of λ-bound variables should not affect meaning
- e.g., $\lambda f : A \to B.\, \lambda x : A.\, f\ x$ should have the same meaning as $\lambda x : A \to B.\, \lambda y : A.\, x\ y$
- this issue is best dealt with at the level of syntax rather than semantics
- from now on we re-define $\lambda^{\to}$ term to mean not an abstract syntax tree but rather an equivalence class of such trees with respect to α-equivalence $s =_\alpha t$:

$$\frac{}{c^A =_\alpha c^A} \qquad \frac{}{x =_\alpha x} \qquad \frac{}{() =_\alpha ()}$$

$$\frac{s =_\alpha s' \qquad t =_\alpha t'}{(s, t) =_\alpha (s', t')} \qquad \frac{t =_\alpha t'}{\mathsf{fst}\ t =_\alpha \mathsf{fst}\ t'} \qquad \frac{t =_\alpha t'}{\mathsf{snd}\ t =_\alpha \mathsf{snd}\ t'}$$

$$\frac{s =_\alpha s' \qquad t =_\alpha t'}{s\ t =_\alpha s'\ t'} \qquad \frac{t \cdot (y\ x) =_\alpha t' \cdot (y\ x') \qquad y \text{ does not occur in } \{\text{x,x',t,t'}\}}{\lambda x : A.\, t =_\alpha \lambda x' : A.\, t'}$$

where $t \cdot (y\ x)$ denotes the result of replacing all occurrences of x with y in t

Example (α-equivalence)

$$
\begin{array}{lclcl}
\lambda x\colon A.\, x\, x & =_\alpha & \lambda y\colon A.\, y\, y & \neq_\alpha & \lambda x\colon A.\, x\, y \\
(\lambda y\colon A.\, y)\, x & =_\alpha & (\lambda x\colon A.\, x)\, x & \neq_\alpha & (\lambda x\colon A.\, x)\, y
\end{array}
$$

Definition (substitution)

- substitution $t[s/x]$ denotes the result of replacing all free occurrences of variable x in term t (i.e. those not occurring within the scope of a $\lambda x : A.\_$ binder) by the term s

Definition (substitution)

- substitution $t[s/x]$ denotes the result of replacing all free occurrences of variable x in term t (i.e. those not occurring within the scope of a $\lambda x : A.\_$ binder) by the term s
- alpha-converting λ-bound variables in t to avoid them "capturing" any free variables of t

Definition (substitution)

- substitution $t[s/x]$ denotes the result of replacing all free occurrences of variable x in term t (i.e. those not occurring within the scope of a $\lambda x : A.\ \_$ binder) by the term s
- alpha-converting λ-bound variables in t to avoid them "capturing" any free variables of t
- e.g., $(\lambda y : A.\ (y, x))[y/x]$ is $\lambda z : A.\ (z, y)$ and is not $\lambda y : A.\ (y, y)$

Definition (substitution)

- substitution $t[s/x]$ denotes the result of replacing all free occurrences of variable x in term t (i.e. those not occurring within the scope of a $\lambda x\colon A.\,\_$ binder) by the term s
- alpha-converting λ-bound variables in t to avoid them "capturing" any free variables of t
- e.g., $(\lambda y\colon A.\,(y,x))[y/x]$ is $\lambda z\colon A.\,(z,y)$ and is not $\lambda y\colon A.\,(y,y)$
- the relation $t[s/x] = t'$ can be inductively defined by the following rules:

$$\frac{}{c^A[s/x] = c^A} \qquad \frac{}{x[s/x] = s} \qquad \frac{y \neq x}{y[s/x] = y} \qquad \frac{}{()[s/x] = ()}$$

$$\frac{t_1[s/x] = t_1' \qquad t_2[s/x] = t_2'}{(t_1, t_2)[s/x] = (t_1', t_2')} \qquad \frac{t[s/x] = t'}{(\mathrm{fst}\ t)[s/x] = \mathrm{fst}\ t'} \qquad \frac{t[s/x] = t'}{(\mathrm{snd}\ t)[s/x] = \mathrm{snd}\ t'} \qquad \frac{t_1[s/x] = t_1' \qquad t_2[s/x] = t_2'}{(t_1\ t_2)[s/x] = t_1'\ t_2'}$$

$$\frac{t[s/x] = t' \qquad y \neq x \text{ and } y \text{ does not freely occur in } s}{(\lambda y\colon A.\,t)[s/x] = \lambda y\colon A.\,t'}$$

Definition ($\beta\eta$-equality)

the relation $\Gamma \vdash s =_{\beta\eta} t : A$ (where Γ ranges over typing environments, s and t over terms and A over types) is inductively defined by the following rules:

- β-conversion

$$\frac{\Gamma, x : A \vdash t : B \qquad \Gamma \vdash s : A}{\Gamma \vdash (\lambda x : A.\, t)\ s =_{\beta\eta} t[s/x] : B} \qquad \frac{\Gamma \vdash s : A \qquad \Gamma \vdash t : B}{\Gamma \vdash \mathrm{fst}(s, t) =_{\beta\eta} s : A} \qquad \frac{\Gamma \vdash s : A \qquad \Gamma \vdash t : B}{\Gamma \vdash \mathrm{snd}(s, t) =_{\beta\eta} t : B}$$

Definition ($\beta\eta$-equality)

the relation $\Gamma \vdash s =_{\beta\eta} t : A$ (where Γ ranges over typing environments, s and t over terms and A over types) is inductively defined by the following rules:

- β-conversion

$$\frac{\Gamma, x : A \vdash t : B \qquad \Gamma \vdash s : A}{\Gamma \vdash (\lambda x : A.\, t)\ s =_{\beta\eta} t[s/x] : B} \qquad \frac{\Gamma \vdash s : A \qquad \Gamma \vdash t : B}{\Gamma \vdash \mathsf{fst}(s, t) =_{\beta\eta} s : A} \qquad \frac{\Gamma \vdash s : A \qquad \Gamma \vdash t : B}{\Gamma \vdash \mathsf{snd}(s, t) =_{\beta\eta} t : B}$$

- η-conversion

$$\frac{\Gamma \vdash t : A \to B \qquad x \text{ does not occur in } t}{\Gamma \vdash t =_{\beta\eta} (\lambda x : A.\, t\ x) : A \to B} \qquad \frac{\Gamma \vdash t : A \times B}{\Gamma \vdash t =_{\beta\eta} (\mathsf{fst}\ t, \mathsf{snd}\ t) : A \times B} \qquad \frac{\Gamma \vdash t : \mathrm{unit}}{\Gamma \vdash t =_{\beta\eta} () : \mathrm{unit}}$$

Definition ($\beta\eta$-equality)

the relation $\Gamma \vdash s =_{\beta\eta} t : A$ (where Γ ranges over typing environments, s and t over terms and A over types) is inductively defined by the following rules:

- β-conversion

$$\frac{\Gamma, x : A \vdash t : B \qquad \Gamma \vdash s : A}{\Gamma \vdash (\lambda x : A.\, t)\ s =_{\beta\eta} t[s/x] : B} \qquad \frac{\Gamma \vdash s : A \qquad \Gamma \vdash t : B}{\Gamma \vdash \mathrm{fst}(s, t) =_{\beta\eta} s : A} \qquad \frac{\Gamma \vdash s : A \qquad \Gamma \vdash t : B}{\Gamma \vdash \mathrm{snd}(s, t) =_{\beta\eta} t : B}$$

- η-conversion

$$\frac{\Gamma \vdash t : A \to B \qquad x \text{ does not occur in } t}{\Gamma \vdash t =_{\beta\eta} (\lambda x : A.\, t\ x) : A \to B} \qquad \frac{\Gamma \vdash t : A \times B}{\Gamma \vdash t =_{\beta\eta} (\mathrm{fst}\ t, \mathrm{snd}\ t) : A \times B} \qquad \frac{\Gamma \vdash t : \mathrm{unit}}{\Gamma \vdash t =_{\beta\eta} () : \mathrm{unit}}$$

- congruence rules

$$\frac{\Gamma, x : A \vdash t =_{\beta\eta} t' : B}{\Gamma \vdash \lambda x : A.\, t =_{\beta\eta} \lambda x : A.\, t' : A \to B} \qquad \frac{\Gamma \vdash s =_{\beta\eta} s' : A \to B \qquad \Gamma \vdash t =_{\beta\eta} t' : A}{\Gamma \vdash s\ t =_{\beta\eta} s'\ t' : B}$$

Definition ($\beta\eta$-equality)

the relation $\Gamma \vdash s =_{\beta\eta} t : A$ (where Γ ranges over typing environments, s and t over terms and A over types) is inductively defined by the following rules:

- β-conversion

$$\frac{\Gamma, x : A \vdash t : B \qquad \Gamma \vdash s : A}{\Gamma \vdash (\lambda x : A.\, t)\ s =_{\beta\eta} t[s/x] : B} \qquad \frac{\Gamma \vdash s : A \qquad \Gamma \vdash t : B}{\Gamma \vdash \mathrm{fst}(s, t) =_{\beta\eta} s : A} \qquad \frac{\Gamma \vdash s : A \qquad \Gamma \vdash t : B}{\Gamma \vdash \mathrm{snd}(s, t) =_{\beta\eta} t : B}$$

- η-conversion

$$\frac{\Gamma \vdash t : A \to B \qquad x \text{ does not occur in } t}{\Gamma \vdash t =_{\beta\eta} (\lambda x : A.\, t\ x) : A \to B} \qquad \frac{\Gamma \vdash t : A \times B}{\Gamma \vdash t =_{\beta\eta} (\mathrm{fst}\ t, \mathrm{snd}\ t) : A \times B} \qquad \frac{\Gamma \vdash t : \mathrm{unit}}{\Gamma \vdash t =_{\beta\eta} () : \mathrm{unit}}$$

- congruence rules

$$\frac{\Gamma, x : A \vdash t =_{\beta\eta} t' : B}{\Gamma \vdash \lambda x : A.\, t =_{\beta\eta} \lambda x : A.\, t' : A \to B} \qquad \frac{\Gamma \vdash s =_{\beta\eta} s' : A \to B \qquad \Gamma \vdash t =_{\beta\eta} t' : A}{\Gamma \vdash s\ t =_{\beta\eta} s'\ t' : B}$$

- $=_{\beta\eta}$ is reflexive, symmetric and transitive

$$\frac{\Gamma \vdash t : A}{\Gamma \vdash t =_{\beta\eta} t : A} \qquad \frac{\Gamma \vdash s =_{\beta\eta} t : A}{\Gamma \vdash t =_{\beta\eta} s : A} \qquad \frac{\Gamma \vdash r =_{\beta\eta} s : A \qquad \Gamma \vdash s =_{\beta\eta} t : A}{\Gamma \vdash r =_{\beta\eta} t : A}$$

Thanks! $\&$ Questions?