\(\lambda\)-calculus

CS 6371: Advanced Programming Languages

Kevin W. Hamlen

March 22, 2022
Historical Roots

First, some mathematical history...
Deductive Logic

- Euclid’s *The Elements*
  - written c. 300 B.C.
  - deductive reasoning: 23 definitions, 10 axioms
  - geometry, algebra, number theory
  - foundation of western mathematics for about 2000 years

- Problem: Some theorems unprovable from axioms
  - Example: Two circles with centers closer than the sum of their radii have an intersection point.
Set Theory

- First proposed by Georg Cantor in 1874
  - new foundation for mathematics
  - early versions contained paradoxes
    - Russel’s Paradox: the set of all sets that do not contain themselves

- Deductive Set Theory
  - axiomized by Zermelo and Fraenkel between 1908 and 1930
  - Zermelo-Fraenkel set theory with axiom of choice (ZFC)

- Problem: some theorems still unprovable!
  - Example (Continuum Hypothesis): There is no set larger than $\mathbb{N}$ but smaller than $\mathbb{R}$. 
Hilbert’s Program

- Proposed by David Hilbert in 1921
- Goals:
  - Provide an unassailable foundation for all mathematics
  - Find a set of axioms and rules of logical inference sufficient to deductively prove all mathematical theorems.
- Required properties:
  - **Soundness**: no untrue statement provable
  - **Completeness**: all true statements provable
  - **Decidability**: procedure for determining whether any mathematical statement is true or false
Gödel’s Incompleteness Theorem

- Proved by Kurt Gödel in 1931
- Theorem: No finite collection of axioms is both sound and complete(!)
- Ramifications:
  - Given any sound axiomization of mathematics, there are true statements that are unprovable.
  - There exists no decision algorithm for mathematical truth.
- Essentially destroyed Hilbert’s program
- Raised another question: What is decidable?
“Decide” = “Compute”

1936: Two models of “computation” proposed:
- Turing Machines (Alan Turing)
- $\lambda$-calculus (Alonzo Church)

Both models equivalent in power

Church-Turing Thesis: All (reasonable) models of computation are equally powerful.

Birth of Computer Science
- Turing Machines = imperative programming
- $\lambda$-calculus = functional programming
Fun Fact: My Mathematical Ancestry

- **Alonzo Church**
  (PhD Princeton 1927)
  - **Stephen Kleene**
    (PhD Princeton 1934)
  - **Bob Constable**
    (PhD Wisconsin-Madison 1968)
  - **Bob Harper**
    (PhD Cornell 1985)
  - **Greg Morrisett**
    (PhD Carnegie Mellon 1995)
  - **Kevin Hamlen**
    (PhD Cornell 2006)
Today: $\lambda$-calculus
### Syntax

\[ e ::= v \mid \lambda v.e \mid e_1 e_2 \]

Only three syntaxes:
- variables \( v \)
- abstractions \( \lambda v.e \) (functions)
- applications \( e_1 e_2 \)

Some simple examples:
- \( \lambda x.x \) (the identity function)
- \( (\lambda x.x)(\lambda y.yy) \rightarrow_1 \lambda y.yy \)
- \( \lambda x.((\lambda y.y)x) \) does not reduce (already a value)
Free Variables

Legal $\lambda$-expressions must be closed (no free variables), where we define the set of free variables $FV(e)$ by

$$FV(v) = \{v\}$$
$$FV(\lambda v.e) = FV(e) \setminus \{v\}$$
$$FV(e_1 e_2) = FV(e_1) \cup FV(e_2)$$

We require $FV(e) = \emptyset$. 
Semantics

Small-step semantics of $\lambda$-calculus:

$$
\frac{e_1 \rightarrow_1 e'_1}{e_1 e_2 \rightarrow_1 e'_1 e_2} \quad (\lambda v. e_1) e_2 \rightarrow_1 e_1[e_2/v]
$$

(\beta\text{-reduction})

where notation $e_1[e/v]$ denotes capture-avoiding substitution:

$$
v[e/v] = e
$$

$$
v_1[e/v_2] = v_1 \text{ when } v_1 \neq v_2 \text{ (i.e., different variables)}
$$

$$
(\lambda v. e_1)[e/v] = \lambda v. e_1
$$

$$
(\lambda v_1. e_1)[e/v_2] = \lambda v_1. (e_1[e/v_2]) \text{ when } v_1 \neq v_2 \text{ (i.e. different variables)}
$$

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

Intuition: $e_1[e_2/x]$ means replace only the free $x$’s in $e_1$ with $e_2$.

Optional exercise: Devise equivalent large-step and denotational semantics.
Reduction example

\[(\lambda x. (\lambda y. (xy))) (\lambda y. y) (\lambda z. z) \rightarrow_1 ?\]
Reduction example

\[(\lambda x.(\lambda y.(xy)))(\lambda y.y)(\lambda z.z) \rightarrow_1 ?\]
Reduction example

\[
((\lambda x.(\lambda y. (xy))) (\lambda y. y)) (\lambda z. z) \rightarrow_1 \\
(\lambda y. (((\lambda y. y)y)) (\lambda z. z) \rightarrow_1 ?
\]
Reduction example

\[(\lambda x.(\lambda y.(xy)))(\lambda y.y)(\lambda z.z) \rightarrow_1\]
\[(\lambda y.((\lambda y.y)y))(\lambda z.z) \rightarrow_1\]
\[(\lambda y.y)(\lambda z.z) \rightarrow_1?\]
Reduction example

\((\lambda x.(\lambda y.(xy)))(\lambda y.y)(\lambda z.z) \rightarrow_1\)
\((\lambda y.(\lambda y.y)y)(\lambda z.z) \rightarrow_1\)
\((\lambda y.y)(\lambda z.z) \rightarrow_1\)
\((\lambda z.z)\)
Reduction example

\[(\lambda x. (\lambda y. (xy))) (\lambda y. y) (\lambda z. z) \rightarrow_1\]
\[(\lambda y. ((\lambda y. y)y)) (\lambda z. z) \rightarrow_1\]
\[(\lambda y. y)(\lambda z. z) \rightarrow_1\]
\[(\lambda z. z)\]

Important observations:

- Don’t change any variable names as you evaluate!
- There are no stores involved here!
- Semantics of \(\lambda\)-calculus are based on capture-avoiding substitution, not stores or variable renaming.
- Function bodies never evaluate (even if they could) until their \(\lambda\)-binder gets stripped off (at which point they’re not functions anymore).

Strategy: Pretend that “\(\lambda v.e\)” is OCaml “\(\text{fun } v \rightarrow e\)”.
Precedence and Associativity

Precedence and associativity conventions:

\[ \lambda v. e_1 e_2 = \lambda v. (e_1 e_2) \]  
(application binds tighter than abstraction)

\[ e_1 e_2 e_2 = (e_1 e_2) e_3 \]  
(application associates left)

Parenthesize anything else that might be ambiguous.
Amazing fact: This incredibly simple language is Turing-complete (can perform any computation implementable by modern computers)!

Proof by reduction (recall from computability theory): Let’s reduce a (simple) Turing-complete programming language to \( \lambda \)-calculus.
Higher-arity Functions

$\lambda$-calculus only gives us 1-argument functions $\lambda v.e$.

Q: How could I create a multi-argument function?
Higher-arity Functions

$\lambda$-calculus only gives us 1-argument functions $\lambda v.e$.

**Q:** How could I create a multi-argument function?

**A:** Nest the $\lambda$’s: $\lambda x.\lambda y.\lambda z.(\ldots)$

**Definition (currying):** In functional programming, changing a function on tuple-arguments to use distinct (non-tuple) arguments is called *currying* the function.

Example:
**Uncurried:** let add (x,y) = x+y;;
**Curried:** let add x y = x+y;;

**Benefits:** More opportunities for code-reuse through partial evaluation, and more opportunities for compiler optimization through specialization.
How might we encode boolean expressions as \( \lambda \)-terms? Let’s start with constants and the ternary operator:

\[
\begin{align*}
\text{true} &= \ ? \\
\text{false} &= \ ? \\
\text{e}_1 \ ? \text{e}_2 : \text{e}_3 &= \ ?
\end{align*}
\]
How might we encode boolean expressions as λ-terms? Let’s start with constants and the ternary operator:

\[
\begin{align*}
\text{true} &= (\lambda x.\lambda y.x) \\
\text{false} &= (\lambda x.\lambda y.y) \\
e_1 \ ? \ e_2 \ : \ e_3 &= ((e_1)(e_2)(e_3))
\end{align*}
\]
How might we encode boolean expressions as $\lambda$-terms? Let’s start with constants and the ternary operator:

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

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

$$e_1 ? e_2 : e_3 = ((e_1)(e_2)(e_3))$$

Using the above, how might we encode not, and, and or as functions over booleans?

$$not = ?$$

$$and = ?$$

$$or = ?$$
How might we encode boolean expressions as $\lambda$-terms? Let’s start with constants and the ternary operator:

- $true = (\lambda x. \lambda y. x)$
- $false = (\lambda x. \lambda y. y)$
- $e_1 ? e_2 : e_3 = ((e_1)(e_2)(e_3))$

Using the above, how might we encode $\text{not}$, $\text{and}$, and $\text{or}$ as functions over booleans?

- $\text{not} = (\lambda b. (b ? false : true))$
- $\text{and} = ?$
- $\text{or} = ?$
How might we encode boolean expressions as $\lambda$-terms? Let’s start with constants and the ternary operator:

\[
\begin{align*}
true &= (\lambda x.\lambda y.x) \\
false &= (\lambda x.\lambda y.y) \\
e_1 ? e_2 : e_3 &= ((e_1)(e_2)(e_3))
\end{align*}
\]

Using the above, how might we encode $\text{not}$, $\text{and}$, $\text{and}$ or $\text{or}$ as functions over booleans?

\[
\begin{align*}
\text{not} &= (\lambda b.(b ? \text{false} : \text{true})) \\
\text{and} &= (\lambda b_1.\lambda b_2.(b_1 ? b_2 : \text{false})) \\
\text{or} &= ?
\end{align*}
\]
How might we encode boolean expressions as $\lambda$-terms? Let’s start with constants and the ternary operator:

\[
\text{true} = (\lambda x.\lambda y.x) \\
\text{false} = (\lambda x.\lambda y.y) \\
e_1 \ ? \ e_2 : e_3 = ((e_1)(e_2)(e_3))
\]

Using the above, how might we encode \text{not}, \text{and}, \text{and} or as functions over booleans?

\[
\text{not} = (\lambda b.(b \ ? \ \text{false} : \text{true})) \\
\text{and} = (\lambda b_1.\lambda b_2.(b_1 \ ? \ b_2 : \text{false})) \\
\text{or} = (\lambda b_1.\lambda b_2.(b_1 \ ? \ \text{true} : b_2))
\]
Tuples

How might we encode pairs?

- The \texttt{pair} function should take two arguments (could be anything) and package them together into some kind of object.
- The $\pi_1$ function (\texttt{fst} in OCaml) should accept a pair as input and recover (project out) the first element.
- The $\pi_2$ function (\texttt{snd} in OCaml) should analogously project out the second element.

\[
\begin{align*}
\text{pair} &= (\lambda x. \lambda y. ?) \\
\pi_1 &= (\lambda p. ?) \\
\pi_2 &= (\lambda p. ?)
\end{align*}
\]
How might we encode pairs?

- The pair function should take two arguments (could be anything) and package them together into some kind of object.
- The $\pi_1$ function (fst in OCaml) should accept a pair as input and recover (project out) the first element.
- The $\pi_2$ function (snd in OCaml) should analogously project out the second element.

\[
\text{pair} = (\lambda x. \lambda y. \lambda b. (b \ ? \ x : y)) \\
\pi_1 = (\lambda p. p \text{ true}) \\
\pi_2 = (\lambda p. p \text{ false})
\]
How might we encode natural numbers?

- Each number $0_N, 1_N, 2_N, \ldots$ should be encoded as a λ-calculus value (must not reduce to something else).
- Approach: Encode $0_N$, then code up a successor function $\text{succ}_N$.
- Should also have predecessor $\text{pred}_N$ (don’t care what it returns for $0_N$).
- Also need a test $\text{iszero}_N$ (returns a boolean).

\[
\begin{align*}
0_N &= \? \\
\text{succ}_N &= (\lambda n \ . \ ?) \\
\text{pred}_N &= (\lambda n \ . \ ?) \\
\text{iszero}_N &= (\lambda n \ . \ ?)
\end{align*}
\]
Natural Numbers

How might we encode natural numbers?

- Each number $0_N, 1_N, 2_N, \ldots$ should be encoded as a λ-calculus value (must not reduce to something else).
- Approach: Encode $0_N$, then code up a successor function $\text{succ}_N$.
- Should also have predecessor $\text{pred}_N$ (don’t care what it returns for $0_N$).  
- Also need a test $\text{iszero}_N$ (returns a boolean).

\[
0_N = (\lambda x.x) \\
\text{succ}_N = (\lambda n. \text{pair} (?) n) \\
\text{pred}_N = (\lambda n. ?) \\
\text{iszero}_N = (\lambda n. ?)
\]
Natural Numbers

How might we encode natural numbers?

- Each number $0_N, 1_N, 2_N, \ldots$ should be encoded as a $\lambda$-calculus value (must not reduce to something else).
- Approach: Encode $0_N$, then code up a successor function $\text{succ}_N$.
- Should also have predecessor $\text{pred}_N$ (don’t care what it returns for $0_N$)
- Also need a test $\text{iszero}_N$ (returns a boolean).

\[
\begin{align*}
0_N &= (\lambda x.x) \\
\text{succ}_N &= (\lambda n. \text{pair}(?) n) \\
\text{pred}_N &= \pi_2 \\
\text{iszero}_N &= (\lambda n. ?)
\end{align*}
\]
How might we encode natural numbers?

- Each number $0_\mathbb{N}, 1_\mathbb{N}, 2_\mathbb{N}, \ldots$ should be encoded as a $\lambda$-calculus value (must not reduce to something else).
- Approach: Encode $0_\mathbb{N}$, then code up a successor function $\text{succ}_\mathbb{N}$.
- Should also have predecessor $\text{pred}_\mathbb{N}$ (don’t care what it returns for $0_\mathbb{N}$)
- Also need a test $\text{iszero}_\mathbb{N}$ (returns a boolean).

\[
\begin{align*}
0_\mathbb{N} &= (\lambda x. x) \\
\text{succ}_\mathbb{N} &= (\lambda n. \text{pair false } n) \\
\text{pred}_\mathbb{N} &= \pi_2 \\
\text{iszero}_\mathbb{N} &= (\lambda n. ?)
\end{align*}
\]
Natural Numbers

How might we encode natural numbers?

- Each number $0_N, 1_N, 2_N, \ldots$ should be encoded as a λ-calculus value (must not reduce to something else).
- Approach: Encode $0_N$, then code up a successor function $\text{succ}_N$.
- Should also have predecessor $\text{pred}_N$ (don’t care what it returns for $0_N$)
- Also need a test $\text{iszero}_N$ (returns a boolean).

$$
0_N = (\lambda x.x) \\
\text{succ}_N = (\lambda n. \text{pair false } n) \\
\text{pred}_N = \pi_2 \\
\text{iszero}_N = \pi_1
$$
Natural Numbers

\[ 0_N = (\lambda x.x) \]
\[ \text{succ}_N = (\lambda n. \text{pair} \text{ false } n) \]
\[ \text{pred}_N = \pi_2 \]
\[ \text{iszero}_N = \pi_1 \]

Does \( \text{iszero}_N(0_N) \) really work (should return true)?

\[ 0_N = (\lambda x.x) \text{ is not even a pair!} \]
Natural Numbers

\[ 0_N = (\lambda x.x) \]
\[ \text{succ}_N = (\lambda n. \text{pair} \ false \ n) \]
\[ \text{pred}_N = \pi_2 \]
\[ \text{iszero}_N = \pi_1 \]

Does \( \text{iszero}_N(0_N) \) really work (should return \text{true})?

\[ \text{iszero}_N(0_N) = \pi_1(\lambda x.x) = (\lambda p. p \ true)(\lambda x.x) \]
\[ \rightarrow_1 (\lambda x.x)\text{true} \]
\[ \rightarrow_1 \text{true} \]

It worked!

*Warning: On the homework, I’ll ask you to first fully expand all the encodings into pure \( \lambda \)-terms before doing any evaluation steps. I did it without expanding \text{true} here to illustrate a point.*
Take-aways:

- **λ-calculus** is an **untyped** language.
  - Every syntactically legal, closed term evaluates to something.
  - Can do some very weird things (as we will see...!)

- There is a different language (which we will learn) called **typed** *λ*-calculus.
  - Don’t confuse it with this language!
  - Watch out for web resources that look similar but that concern a different *λ*-calculus (there are many)!
Loops

We’re close to a full Turing-complete language now, but one major thing is missing: loops.

Q: Is it possible to code an infinite loop in λ-calculus?
Loops

We’re close to a full Turing-complete language now, but one major thing is missing: loops.

Q: Is it possible to code an infinite loop in $\lambda$-calculus?
A: Yes. Smallest possible example: $(\lambda x.xx)(\lambda x.xx)$
Recursion

What about useful loops?
Case-study: Can we code an addition function for natural numbers?

\[ \text{add}_N = \lambda m.\lambda n. ? \]
What about useful loops?

Case-study: Can we code an addition function for natural numbers?

$$\text{add}_N = \lambda m. \lambda n. (\text{iszero}_N m ? n : \text{add}_N (\text{pred}_N m)(\text{succ}_N n))$$
Recursion

What about useful loops?
Case-study: Can we code an addition function for natural numbers?

\[
\text{add}_N = \lambda m. \lambda n. (\text{iszero}_N m \ ? n : \text{add}_N (\text{pred}_N m)(\text{succ}_N n))
\]

Circular definition! Remember, the encoding part (\(=\)) is supposed to be a definition; it’s not part of the \(\lambda\)-term.

How can we remove the recursion from this formula?
Fixed points

\[ \text{add}_N = \lambda m.\lambda n. (\text{iszero}_N m \ ? \ n : \text{add}_N (\text{pred}_N m)(\text{succ}_N n)) \]

Define a functional whose least fixed point is \( \text{add}_N \):

\[ \text{Add}_N = \lambda f.\lambda m.\lambda n. (\text{iszero}_N m \ ? \ n : f(\text{pred}_N m)(\text{succ}_N n)) \]

Then define \( \text{add}_N \) to be its least fixed point:

\[ \text{add}_N = \text{fix}(\text{Add}_N) \]

But \( \text{fix} \) is not part of \( \lambda \)-calculus, so we’re still stuck...?
A very interesting function (discovered by Haskell Curry):

\[ Y = \lambda f. (\lambda x. f(xx))(\lambda x. f(xx)) \]

Amazing claim: \( Y = \text{fix} \)

Proof: Let’s evaluate it...

\[ Y \ g \rightarrow_1 ? \]
A very interesting function (discovered by Haskell Curry):

\[ Y = \lambda f. (\lambda x. f(xx))(\lambda x. f(xx)) \]

Amazing claim: \( Y = fix \)

Proof: Let’s evaluate it...

\[ Y \, g \rightarrow_1 (\lambda x. g(xx))(\lambda x. g(xx)) \]
\[ \rightarrow_1 ? \]
A very interesting function (discovered by Haskell Curry):

\[ Y = \lambda f. (\lambda x. f(xx))(\lambda x. f(xx)) \]

Amazing claim: \( Y = fix \)

Proof: Let’s evaluate it...

\[ Y \ g \rightarrow_1 (\lambda x. g(xx))(\lambda x. g(xx)) \]
\[ \rightarrow_1 g((\lambda x. g(xx))(\lambda x. g(xx))) \]
A very interesting function (discovered by Haskell Curry):

\[ Y = \lambda f. (\lambda x. f(xx))(\lambda x. f(xx)) \]

Amazing claim: \(Y = \text{fix}\)

Proof: Let’s evaluate it...

\[ Y \ g \rightarrow_1 (\lambda x. g(xx))(\lambda x. g(xx)) \]
\[ \rightarrow_1 g((\lambda x. g(xx))(\lambda x. g(xx))) = g(Y \ g) \]

Conclusion: \(Y \ g\) is the fixed point of \(g\). (Whoa!)
**Exercise:** Define an addition function in $\lambda$-calculus.

The following definition is illegal (not well-founded):

$$\text{add}_\mathbb{N} = \lambda m.\lambda n. (\text{iszero}_\mathbb{N} m \ ? \ n : \text{add}_\mathbb{N}(\text{pred}_\mathbb{N} m)(\text{succ}_\mathbb{N} n))$$

So instead define a functional whose least fixed point is $\text{add}_\mathbb{N}$:

$$\lambda f.\lambda m.\lambda n. (\text{iszero}_\mathbb{N} m \ ? \ n : f(\text{pred}_\mathbb{N} m)(\text{succ}_\mathbb{N} n))$$

Then apply $Y$ to it:

$$\text{add}_\mathbb{N} = Y(\lambda f.\lambda m.\lambda n. (\text{iszero}_\mathbb{N} m \ ? \ n : f(\text{pred}_\mathbb{N} m)(\text{succ}_\mathbb{N} n)))$$

Now we have a legal definition of an addition function with no explicit recursions in it.
Exercise: Define a multiplication function for natural numbers in $\lambda$-calculus.

Try to define it recursively first:

\[
\text{mul}_N = \lambda m. \lambda n. (\text{iszero } N \, m \, ? \, 0 \, N : \text{add } N (\text{mul } N (\text{pred } N \, m) \, n) \, n)
\]
Exercise: Multiplication

**Exercise:** Define a multiplication function for natural numbers in $\lambda$-calculus.

Try to define it recursively first:

$$\text{mul}_N = \lambda m. \lambda n. (\text{iszero}_N m ? 0_N : \text{add}_N (\text{mul}_N (\text{pred}_N m) n) n)$$
Exercise: Multiplication

**Exercise:** Define a multiplication function for natural numbers in \( \lambda \)-calculus.

Try to define it recursively first:

\[
\text{mul}_N = \lambda m.\lambda n. (\text{iszero}_N m ? 0_N : \text{add}_N (\text{mul}_N (\text{pred}_N m)n)n)
\]

Then change it to a non-recursive functional and apply \( Y \) to it:

\[
\text{mul}_N = Y (\lambda f.\lambda m.\lambda n. (\text{iszero}_N m ? 0_N : \text{add}_N (f(\text{pred}_N m)n)n))
\]
When solving these sorts of problems on homeworks, quizzes, and exams:

- Please DO use the abbreviations in your code.
  - Don't write \((\lambda x.\lambda y.x)\) when you mean \texttt{true}.
  - Strive for readability (otherwise becomes very hard to grade!).

- Please DO define named helper functions.
  - Less writing is good; don't repeatedly write out same subroutine.
  - But any recursions must always be eliminated with \(Y\).
  - Use informative names (not \(f\)).

- Don't name variables the same as any helper functions (really confusing!).

- \(\lambda\)-calculus is a math formalism not a modern language, so extra effort is required to make it readable.
Equality

$\lambda$-terms are ASTs. They are only “equal” ($\equiv$) if they are identical after expansion of all macro abbreviations.

(Also recall that the parentheses are not symbols in the AST; they just show the structure of the AST.)

Examples:

$$(\lambda y.y)(\lambda x.x) \not\equiv \lambda x.x$$  (though they evaluate to the same terms)

$$(\lambda x.(x)) = \lambda x.x$$

$$\lambda x.x \not\equiv \lambda y.y$$

However, there are some notions of term equivalence that are important to understand.
**Definition (α-equivalence):** Term $\lambda x.e$ is $\alpha$-equivalent to term $\lambda y.(e'[y/x])$ (written $\lambda x.e \equiv_{\alpha} \lambda y.(e'[y/x])$) whenever $e \equiv_{\alpha} e'$ (recursively).

Intuition: Terms that are identical except for consistent, capture-avoiding renaming of the variables are $\alpha$-equivalent.

Examples:

\[
\begin{align*}
\lambda x.x & \equiv_{\alpha} \lambda y.y \\
\lambda x.\lambda x.x & \equiv_{\alpha} \lambda y.\lambda x.x \\
\lambda x.\lambda x.x & \not\equiv_{\alpha} \lambda y.\lambda x.y
\end{align*}
\]

Colloquially: Functional programmers refer to renaming their variables as "$\alpha$-conversion".
**β-equivalence**

**Definition (β-equivalence):** Terms \((\lambda v.e_1)e_2\) and \(e_1[e_2/x]\) are β-equivalent (written \((\lambda v.e_1)e_2 \equiv_\beta e_1[e_2/x]\)).

Intuition: An application of a function \(f\) to an argument \(a\) is β-equivalent to a term consisting of the body of \(f\) with all its parameters replaced with the argument term \(a\).

Examples:

\[
(\lambda x.xx)(\lambda y.y) \equiv_\beta (\lambda y.y)(\lambda y.y)
\]

\[
(\lambda x.xx)(\lambda y.y) \equiv_\beta \lambda y.y \quad \text{(by transitivity)}
\]

\[
((\lambda x.xx)(\lambda y.y))(\lambda z.z) \not\equiv_\beta ((\lambda y.y)(\lambda y.y))(\lambda z.z)
\]

The last example is because that reduction doesn’t only use the β-rule. In that case the left subterms are β-equivalent, but not the full-sized terms that contain them.
Definition (η-equivalence): Terms $\lambda v.(fv)$ and $f$ are η-equivalent (written $\lambda v.(fv) \equiv_\eta f$) if $v \notin FV(f)$.

Intuition: A “wrapper function” that merely applies some other function $f$ to whatever argument it receives is equivalent to just $f$.

Example:

$$\lambda n.\text{pair false } n \equiv_\eta \text{pair false}$$

Example from OCaml:

```ocaml
let sum x = List.fold_left (+) 0 x;;
≡_η
let sum = List.fold_left (+) 0;;
```
Don’t confuse equivalence with the operational semantics of λ-calculus:

- Only β-equivalence is a rule of the operational semantics.
  - α-equivalent terms don’t always evaluate to the same final terms (variables might be different, which makes them different ASTs).
  - β-equivalent terms do always evaluate to the same terms.
  - η-equivalent terms “behave the same” when applied, but η-equivalence is not a reduction step of λ-calculus.

- There is no = or ≡ test operation in λ-calculus!
  - The following is NOT a legal λ-term:
    \[ \lambda x.\lambda y.(x = y) \? \text{true} : \text{false} \]
  - It is impossible to code up such an operation (exercise: prove it!).

- In denotational semantics, λ-terms denote (mathematical) functions.
  - In math we have another definition of functional equivalence (identical input-output relations).
  - But functional equivalence is not decidable (Rice’s Theorem).
  - And equivalence of λ-term denotations is NOT the same as equivalence of the terms themselves.