λ-Calculu

Dr. Liam O'Connor
University of Edinburgh LFCS
UNSW, Term 3 2020

λ-Calculus

The term language we defined for Higher Order Abstract Syntax is almost a full featured programming language.
Just enrich the syntax slightly:

$$\mid \quad \lambda x.\ t \qquad (\lambda\text{-}$$

There is just one rule to evaluate terms, called

$$(\lambda x.\ t)\ u \quad \mapsto_\beta \quad t[x := u]$$

Just as in Haskell, $(\lambda x.\ t)$ denotes a function that, given an argument for x, will return t.

Syntax Concerns

Function application is left associative:

$$f\ a\ b\ c \quad = \quad ((f\ a)\ b)\ c$$

λ-abstraction ex

$$\lambda a.\ f\ a\ b \quad = \quad \lambda a.\ (f\ a\ b)$$

All functions are unary, like Haskell. Multiple argument func
nested λ-abstractions:

$$\lambda x.\lambda y.\ x + y$$

β-reduction

β-reduction is a *congruence*:

$$\frac{}{(\lambda x.\ t)\ u \mapsto_\beta t[x := u]}$$

This means we can pick any reducible subexpression (called a rform β-reduction.

β-reduction

β-reduction is a *congruence*:

$$\frac{}{(\lambda x.\ t)\ u \mapsto_\beta t[x := u]}$$

This means we can pick any reducible subexpression (called a rform β-reduction.

Example:

$$(\lambda x.\ \lambda y.\ f\ (y\ x))\ 5\ (\lambda x.\ x)$$

β-reduction

β-reduction is a *congruence*:

$$\overline{(\lambda x.\ t)\ u \mapsto_\beta t[x := u]}$$

This means we can pick any reducible subexpression (called a rform β-reduction.

Example:

$$(\lambda x.\ \lambda y.\ f\ (y\ x))\ 5\ (\lambda x.\ x) \quad \mapsto_\beta \quad (\lambda y.\ f\ (y\ 5))\ (\lambda x.\ x)$$

β-reduction

β-reduction is a *congruence*:

$$\frac{}{(\lambda x.\ t)\ u \mapsto_\beta t[x := u]}$$

$$\frac{\mapsto}{\mapsto} \qquad \frac{\mapsto}{\mapsto} \qquad \frac{\mapsto}{\mapsto}$$

This means we can pick any reducible subexpression (called a rform β-reduction.

Example:

$$\begin{aligned} (\lambda x.\ \lambda y.\ f\ (y\ x))\ 5\ (\lambda x.\ x) &\mapsto_\beta (\lambda y.\ f\ (y\ 5))\ (\lambda x.\ x) \\ &\mapsto_\beta f\ ((\lambda x.\ x)\ 5) \end{aligned}$$

β-reduction

β-reduction is a *congruence*:

$$\overline{(\lambda x.\ t)\ u \mapsto_\beta t[x := u]}$$

This means we can pick any reducible subexpression (called a rform β-reduction.

Example:

$$\begin{aligned}
(\lambda x.\ \lambda y.\ f\ (y\ x))\ 5\ (\lambda x.\ x) &\mapsto_\beta (\lambda y.\ f\ (y\ 5))\ (\lambda x.\ x)\\
&\mapsto_\beta f\ ((\lambda x.\ x)\ 5)\\
&\mapsto_\beta f\ 5
\end{aligned}$$

Confluence

Supposing we arrive via one reduction path to an expression that cannot be reduced further (called a *normal form*), then any other reduction path will result in the same normal form.

Confluence

Supposing we arrive via one reduction path to an expression that cannot be reduced further (called a *normal form*), then any other reduction path will result in the same normal form.

$$(\lambda y.\ f\ y)\ 5$$

$$\downarrow$$

$$f\ 5$$

Confluence

Supposing we arrive via one reduction path to an expression that cannot be reduced further (called a *normal form*), then any other reduction path will result in the same normal form.

$$\begin{tikzcd}
 & {} \arrow[dl] \arrow[dr] & \\
(\lambda y.\ f\ y)\ 5 \arrow[dr] & & {} \arrow[dl] \\
 & f\ 5 &
\end{tikzcd}$$

Equivalence

Confluence means we can define another notion of *equivalence*, which equates more than α-equivalence. Two terms are $\alpha\beta$-*equivalent*, written $s \equiv_{\alpha\beta} t$ if they β-reduce to α-equivalent no

Equivalence

Confluence means we can define another notion of *equivalence*, which equates more than α-equivalence. Two terms are *$\alpha\beta$-equivalent*, written $s \equiv_{\alpha\beta} t$ if they β-reduce to α-equivalent no

η

There is also anoth lence alone, called η-reduction:

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

Adding this reduction to the system preserves confluence and uniqueness of normal forms, so we have a notion of *$\alpha\beta\eta$-equivalence* also.

Normal Forms

Does every term in

Normal Forms

Does every term in

Try to β-reduce this! (the answer is that it doesn't have a normal fo

Why learn this stuff?

- λ-calculus is a *Turing-complete* programming language.
- λ-calculus i e non-funct

- λ-calculus i *ory*, the two main foundations used for mathematics in interactive
- λ-calculus is the smallest example of a usable programm for teaching about programming languages.

Making λ-Calculus Usable

In order to demonstrate that λ calculus is actually a usable programming language, we will demonstrate terms, along with their operations.

General Idea

We transform a data type into the type of its *eli* e a function that can serve the same purpose as the data type at its use s

Booleans

How do we use booleans?

Booleans

How do we use booleans? To choose between two results!

Booleans

How do we use booleans? To choose between two results!

So, a boolean will be a function that, given two arguments, returns the first one if it is true and the second one if it is false:

How do we write conjunction? to board

Booleans

How do we use booleans? To choose between two results!

So, a boolean will be a function that, given two arguments, returns the first one if it is true and the second one if it is false:

How do we write conjunction? to board

$$\textsc{And} \equiv \lambda p.\ \lambda$$

Booleans

How do we use booleans? To choose between two results!

So, a boolean will be a function that, given two arguments, returns the first one if it is true and the second one if it is false:

How do we write conjunction? to board

$$\textsc{And} \equiv \lambda p.\, \lambda$$

Example (Test it out!)

Try β-normalising $\textsc{And}\ \textsc{True}\ \textsc{False}$.

Booleans

How do we use booleans? To choose between two results!

So, a boolean will be a function that, given two arguments, returns the first one if it is true and the second one if it is false:

How do we write conjunction? to board

$$\textsc{And} \equiv \lambda p.\, \lambda$$

Example (Test it out!)

Try β-normalising $\textsc{And}\ \textsc{True}\ \textsc{False}$.

What about $\textsc{Or}$?

Natural Numbers

How do we use natural numbers?

Natural Numbers

How do we use natural numbers? To do something n times!

Natural Numbers

How do we use natural numbers? To do something n times!

So, a natural number will be a function that takes a function f and a value x, and applies the function f to x that number of times:

$\ldots$

How do we write $\textsc{Suc}$?

Natural Numbers

How do we use natural numbers? To do something n times!

So, a natural number will be a function that takes a function f and a value x, and applies the function f to x that number of times:

$$\ldots$$

How do we write Suc?

$$\text{Suc} \equiv \lambda n.\ \lambda f.\ \lambda x$$

Natural Numbers

How do we use natural numbers? To do something n times!

So, a natural number will be a function that takes a function f and a value x, and applies the function f to x that number of times:

$$\ldots$$

How do we write $\textsc{Suc}$?

$$\textsc{Suc} \equiv \lambda n.\ \lambda f.\ \lambda x$$

How do we write $\textsc{Add}$?

Natural Numbers

How do we use natural numbers? To do something n times!

So, a natural number will be a function that takes a function f and a value x, and applies the function f to x that number of times:

$$\ldots$$

How do we write Suc?

$$\textsc{Suc} \equiv \lambda n.\ \lambda f.\ \lambda x$$

How do we write Add?

$$\textsc{Add} \equiv \lambda m.\lambda n.\ \lambda f.\ \lambda x.\ m\ f\ (n\ f\ x)$$

Natural Numbers

How do we use natural numbers? To do something n times!

So, a natural number will be a function that takes a function f and a value x, and applies the function f to x that number of times:

$$\ldots$$

How do we write $\textsc{Suc}$?

$$\textsc{Suc} \equiv \lambda n.\ \lambda f.\ \lambda x$$

How do we write $\textsc{Add}$?

$$\textsc{Add} \equiv \lambda m. \lambda n.\ \lambda f.\ \lambda x.\ m\ f\ (n\ f\ x)$$

Natural Number Practice

Example

Try β-normalis

Example

Try writing a different λ-term for defining $\textsc{Suc}$.

Example

Try writing a λ-term for defining $\textsc{Multiply}$.