\textit{\lam-Calculus}

Johannes Åman Pohjola
UNSW
Term 3 2022
The term language we defined for Higher Order Abstract Syntax is almost a full featured programming language. Just enrich the syntax slightly:

\[
  t ::= \text{Symbol} \\
  \quad | \quad x \quad \text{(variables)} \\
  \quad | \quad t_1 \ t_2 \quad \text{(application)} \\
  \quad | \quad \lambda x. \ t \quad \text{(\(\lambda\)-abstraction)}
\]

There is just one rule to evaluate terms, called \(\beta\)-reduction:

\[
  (\lambda x. \ t) \ u \ \leftrightarrow_\beta \ t[x := u]
\]

Just as in Haskell, \((\lambda x. \ t)\) denotes a function that, given an argument for \(x\), returns \(t\).
Syntax Concerns

Function application is left associative:

\[ f \ a \ b \ c = ((f \ a) \ b) \ c \]

\(\lambda\)-abstraction extends as far as possible:

\[ \lambda a. \ f \ a \ b = \lambda a. \ (f \ a \ b) \]

All functions are unary, like Haskell. Multiple argument functions are modelled with nested \(\lambda\)-abstractions:

\[ \lambda x. \lambda y. \ x + y \]
\( \beta \)-reduction

\( \beta \)-reduction is a *congruence*:

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

\[
t \mapsto_\beta t'
\]
\[
s \mapsto_\beta s'
\]
\[
t \mapsto_\beta t'
\]

\[
s \, t \mapsto_\beta s \, t'
\]
\[
s \, t \mapsto_\beta s' \, t
\]
\[
\lambda x. \, t \mapsto_\beta \lambda x. \, t'
\]

This means we can pick any reducible subexpression (called a *redex*) and perform \( \beta \)-reduction.
\(\beta\)-reduction

\(\beta\)-reduction is a *congruence*:

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

\[
t \mapsto_{\beta} t'
\]

\[
s \mapsto_{\beta} s'
\]

\[
t \mapsto_{\beta} t'
\]

This means we can pick any reducible subexpression (called a *redex*) and perform \(\beta\)-reduction.

**Example:**

\[
(\lambda x. \lambda y. f (y \ x)) \ 5 \ 5
\]

\[(\lambda x. \ x)\]
\(\beta\)-reduction

\(\beta\)-reduction is a \textit{congruence}:

\[
\begin{align*}
(\lambda x.\ t) \ u & \mapsto_{\beta} t[x := u] \\
t & \mapsto_{\beta} t' \\
s & \mapsto_{\beta} s' \\
t & \mapsto_{\beta} t' \\
s t & \mapsto_{\beta} s' t \\
\lambda x.\ t & \mapsto_{\beta} \lambda x.\ t'
\end{align*}
\]

This means we can pick any reducible subexpression (called a \textit{redex}) and perform \(\beta\)-reduction.

\textbf{Example}:

\[
(\lambda x.\ \lambda y.\ f\ (y\ x))\ 5\ (\lambda x.\ x) \mapsto_{\beta} (\lambda y.\ f\ (y\ 5))\ (\lambda x.\ x)
\]
**\( \beta \)-reduction**

\( \beta \)-reduction is a *congruence*:

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

\[
\begin{align*}
\text{If } t & \mapsto_{\beta} t' & \text{then } st & \mapsto_{\beta} s' t \\
\text{If } s & \mapsto_{\beta} s' & \text{then } st & \mapsto_{\beta} s' t
\end{align*}
\]

This means we can pick any reducible subexpression (called a *redex*) and perform \( \beta \)-reduction.

**Example:**

\[
(\lambda x. \ \lambda y. \ f \ (y \ x)) \ 5 \ (\lambda x. \ x) \ \mapsto_{\beta} \ (\lambda y. \ f \ (y \ 5)) \ (\lambda x. \ x)
\]

\[
\mapsto_{\beta} \ f \ ((\lambda x. \ x) \ 5)
\]
\(\beta\)-reduction

\(\beta\)-reduction is a congruence:

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

\[
t \mapsto_\beta t'
\]

\[
s \mapsto_\beta s'
\]

\[
t \mapsto_\beta t'
\]

\[
s \; t \mapsto_\beta s' \; t
\]

\[
s \; t \mapsto_\beta s' \; t
\]

\[
\lambda x. \; t \mapsto_\beta \lambda x. \; t'
\]

This means we can pick any reducible subexpression (called a redex) and perform \(\beta\)-reduction.

Example:

\[
(\lambda x. \; \lambda y. \; f \; (y \; x)) \; 5 \; (\lambda x. \; x) \mapsto_\beta (\lambda y. \; f \; (y \; 5)) \; (\lambda x. \; x)
\]

\[
(\lambda y. \; f \; (y \; 5)) \; (\lambda x. \; x) \mapsto_\beta f \; ((\lambda x. \; x) \; 5)
\]

\[
f \; ((\lambda x. \; x) \; 5) \mapsto_\beta f \; 5
\]
Confluence

Suppose 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

Suppose we arrive via one reduction path to an expression that cannot be reduced further (called a \textit{normal form}). Then any other reduction path will result in the same normal form.

\[(\lambda a. a) \, ((\lambda y. f \, y) \, 5)\]
Suppose 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.

\[(\lambda a. a) ((\lambda y. f y) 5)\]

\[\downarrow\]

\[(\lambda y. f y) 5 \quad (\lambda a. a) (f 5)\]

\[\downarrow\]

\[f 5\]
Equivalence

Confluence means we can define another notion of equivalence, which equates more than $\alpha$-equivalence. Two terms are $\alpha\beta$-equivalent, written $s \equiv_{\alpha\beta} t$ if they $\beta$-reduce to $\alpha$-equivalent normal forms.

There is also another equation that cannot be proven from $\beta$-equivalence alone, called $\eta$-reduction: $(\lambda x. f x) \mapsto_{\eta} f$.

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

Confluence means we can define another notion of *equivalence*, which equates more than $\alpha$-equivalence. Two terms are $\alpha\beta$-*equivalent*, written $s \equiv_{\alpha\beta} t$ if they $\beta$-reduce to $\alpha$-equivalent normal forms.

There is also another equation that cannot be proven from $\beta$-equivalence alone, called $\eta$-reduction:

$$(\lambda x. f \; x) \eta \rightarrow f$$

Adding this reduction to the system preserves confluence and uniqueness of normal forms, so we have a notion of $\alpha\beta\eta$-*equivalence* also.
Does every term in $\lambda$-calculus have a normal form?

(\lambda x. x x) (\lambda x. x x)

Try to $\beta$-reduce this! (the answer is that it doesn't have a normal form)
Normal Forms

Does every term in \( \lambda \)-calculus have a normal form?

\[(\lambda x. x \ x)(\lambda x. x \ x)\]

Try to \( \beta \)-reduce this!  (the answer is that it doesn’t have a normal form)
Why learn this stuff?

- λ-calculus is a *Turing-complete* programming language.
- λ-calculus is the foundation for every functional programming language and some non-functional ones.
- λ-calculus is the foundation of *Higher Order Logic* and *Type Theory*, the two main foundations used for mathematics in interactive proof assistants.
- λ-calculus is the smallest example of a usable programming language, so it’s good for research and teaching about programming languages.
In order to demonstrate that $\lambda$ calculus is actually a usable (in theory) programming language, we will demonstrate how to encode booleans and natural numbers as $\lambda$-terms, along with their operations.

**General Idea**

We transform a data type into the type of its *eliminator*. In other words, we make a function that can serve the same purpose as the data type at its use sites.
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:

**True** ≡ λa. λb. a

**False** ≡ λa. λb. b

How do we write conjunction?

And ≡ λp. λq. p q p

Example (Test it out!)

Try β-normalising True False.

What about Implies?
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:

True ≡ λa. λb. a
False ≡ λa. λb. b

How do we write conjunction?

And ≡ λp. λq. p q p

Example (Test it out!)

Try β-normalising And True False.

What about Implies?
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:

\[
\text{True} \equiv \lambda a. \lambda b. a \\
\text{False} \equiv \lambda a. \lambda b. b
\]

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:

\[
\text{True} \equiv \lambda a. \lambda b. a \\
\text{False} \equiv \lambda a. \lambda b. b
\]

How do we write conjunction? to “board”

\[
\text{AND} \equiv \lambda p. \lambda q. p \; q \; p
\]
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:

\[
\text{True} \equiv \lambda a. \lambda b. a
\]
\[
\text{False} \equiv \lambda a. \lambda b. b
\]

How do we write conjunction? to “board”

\[
\text{And} \equiv \lambda p. \lambda q. p \ q \ q \ p
\]

**Example (Test it out!)**

Try \(\beta\)-normalising \text{And \ True \ 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:

$$
\text{True} \equiv \lambda a. \lambda b. a \\
\text{False} \equiv \lambda a. \lambda b. b
$$

How do we write conjunction? to “board”

$$\text{And} \equiv \lambda p. \lambda q. p \ q \ p$$

**Example (Test it out!)**

Try $\beta$-normalising $\text{And True False}$.  

What about $\text{Implies}$?
Natural Numbers

How do we use natural numbers?
Natural Numbers

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

Zero $\equiv \lambda f. \lambda x. x$

One $\equiv \lambda f. \lambda x. f x$

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

Suc $\equiv \lambda n. \lambda f. \lambda x. f (n f x)$

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:

\[
\begin{align*}
\text{Zero} & \equiv \lambda f. \lambda x. x \\
\text{One} & \equiv \lambda f. \lambda x. f x \\
\text{Two} & \equiv \lambda f. \lambda x. f (f x) \\
\ldots
\end{align*}
\]

How do we write \textit{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:

\[
\begin{align*}
\text{Zero} & \equiv \lambda f. \lambda x. x \\
\text{One} & \equiv \lambda f. \lambda x. f x \\
\text{Two} & \equiv \lambda f. \lambda x. f (f x) \\
\ldots
\end{align*}
\]

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

\[
\text{Suc} \equiv \lambda n. \lambda f. \lambda x. f (n f x)
\]
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:

\[
\begin{align*}
\text{Zero} &\equiv \lambda f. \lambda x. x \\
\text{One} &\equiv \lambda f. \lambda x. f x \\
\text{Two} &\equiv \lambda f. \lambda x. f (f x) \\
&\ldots
\end{align*}
\]

How do we write \( \text{Suc} \)?

\[
\text{Suc} \equiv \lambda n. \lambda f. \lambda x. f (n f x)
\]

How do we write \( \text{Add} \)?
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:

\[
\text{Zero} \equiv \lambda f. \lambda x. x \\
\text{One} \equiv \lambda f. \lambda x. f \ x \\
\text{Two} \equiv \lambda f. \lambda x. f \ (f \ x) \\
\ldots 
\]

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

\[
\text{Suc} \equiv \lambda n. \lambda f. \lambda x. f \ (n \ f \ x)
\]

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

\[
\text{Add} \equiv \lambda m. \lambda n. \lambda f. \lambda x. m \ f \ (n \ f \ x)
\]
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:

\[
\begin{align*}
\text{Zero} & \equiv \lambda f. \lambda x. x \\
\text{One} & \equiv \lambda f. \lambda x. f \; x \\
\text{Two} & \equiv \lambda f. \lambda x. f \; (f \; x) \\
\ldots & \\
\text{How do we write \( SUC \)?} \\
SUC & \equiv \lambda n. \lambda f. \lambda x. f \; (n \; f \; x) \\
\text{How do we write \( ADD \)?} \\
ADD & \equiv \lambda m. \lambda n. \lambda f. \lambda x. m \; f \; (n \; f \; x)
\end{align*}
\]
Natural Number Practice

Example
Try $\beta$-normalising $\textbf{Suc \ One}$. 

Example
Try writing a different $\lambda$-term for defining $\textbf{Suc}$. 

Example
Try writing a $\lambda$-term for defining $\textbf{Multiply}$. 