\[ \lambda y. \lambda x. x \]

\[ a \rightarrow b \rightarrow a \]

\text{COMP 3161/9164}

\text{Concepts of Programming Languages}

\text{\lambda-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:

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

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

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

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*:

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

\[
t \mapsto_\beta t'
\]

\[
s \mapsto_\beta s'
\]

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

\[
lx. \; t \mapsto_\beta lx. \; t'
\]

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

**Example:**

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

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

\[
\mapsto_\beta \quad f \; 5
\]
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.

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

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

\[f 5\]
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.

$\eta$

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.
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?

- $\lambda$-calculus is a *Turing-complete* programming language.
- $\lambda$-calculus is the foundation for every functional programming language and some non-functional ones.
- $\lambda$-calculus is the foundation of *Higher Order Logic* and *Type Theory*, the two main foundations used for mathematics in interactive proof assistants.
- $\lambda$-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 λ calculus is actually a usable (in theory) programming language, we will demonstrate how to encode booleans and natural numbers as λ-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:

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

How do we write conjunction? to “board”

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

Example (Test it out!)

Try $\beta$-normalising And True False.

What about Implies?
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 \( \text{Add} \)?

\[
\text{Add} \equiv \lambda m.\lambda n. \lambda f. \lambda x. m \ f \ (n \ f \ x)
\]
Example
Try $\beta$-normalising $\text{Suc \ One}$.

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

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