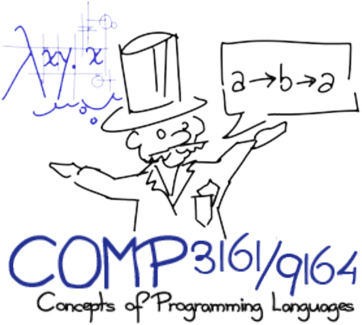


λ-Calculus

Thomas Sewell
UNSW
Term 3 2024

λ-Calculus

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

$$
\begin{array}{rcll}
t & ::= & \texttt{Symbol} & \\
& | & x & \textit{(variables)} \\
& | & t_1\ t_2 & \textit{(application)} \\
& | & \lambda x.\ t & \textit{(}\lambda\textit{-abstraction)}
\end{array}
$$

There is just one rule to evaluate terms, called *β-reduction*:

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

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

Syntax Concerns

Function application is left associative:

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

λ-abstraction extends as far as possible:

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

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

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

β-reduction

β-reduction is a *congruence*:

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

$$\frac{t \mapsto_\beta t'}{s\ t \mapsto_\beta s\ t'} \qquad \frac{s \mapsto_\beta s'}{s\ t \mapsto_\beta s'\ t} \qquad \frac{t \mapsto_\beta t'}{\lambda x.\ t \mapsto_\beta \lambda x.\ t'}$$

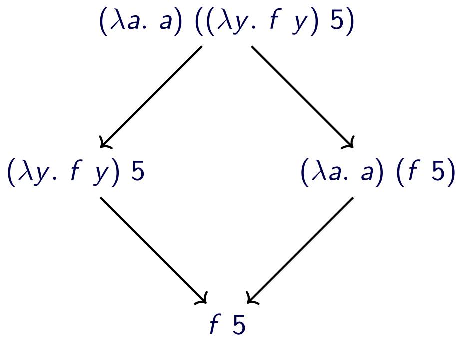
This means we can pick any reducible subexpression (called a *redex*) and perform β-reduction.

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

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.



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 normal forms.

η

There is also another equation that cannot be proven from β-equivalence alone, called η-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 λ-calculus have a normal form?

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

Try to β-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.

Making λ-Calculus Usable

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:

$$\begin{aligned} \textsc{True} &\equiv \lambda a.\ \lambda b.\ a \\ \textsc{False} &\equiv \lambda a.\ \lambda b.\ b \end{aligned}$$

How do we write conjunction? to "board"

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

Example (Test it out!)

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

What about $\textsc{Implies}$?

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:

$$\begin{array}{lcl} \textsc{Zero} & \equiv & \lambda f.\ \lambda x.\ x \\ \textsc{One} & \equiv & \lambda f.\ \lambda x.\ f\ x \\ \textsc{Two} & \equiv & \lambda f.\ \lambda x.\ f\ (f\ x) \\ \ldots & & \end{array}$$

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

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

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-normalising $\textsc{Suc One}$.

Example

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

Example

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