Introduction to Theoretical Computer Science

Lecture 14: λ-Calculus

Dr. Liam O'Connor

University of Edinburgh

Semester 1, 2023/2024

Introduction

While Turing was thinking about machines, Alonzo Church was computing with a programming language – a precursor of Haskell – called *λ-calculus*.

We often think of programming languages as methods to program a computer, but languages can also be thought of as the computer itself.

Introduction

While Turing was thinking about machines, Alonzo Church was computing with a programming language – a precursor of Haskell – called *λ-calculus*.

We often think of programming languages as methods to program a computer, but languages can also be thought of as the computer itself.

Comparing models

Church and Turing famously proved that Turing Machines and λ-calculus are equivalent in computational power.

However, λ-calculus is different from other models in that it is *higher-order*: This means that computations (λ-terms) may take other computations as input. For TMs and RMs, we must work with encodings to achieve this.

Syntax

λ-calculus computations are expressed as *λ-terms*:

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

Syntax

λ-calculus computations are expressed as *λ-terms*:

$$\begin{array}{lcll} t & ::= & x & \textit{(variables)} \\ & \mid & t_1\ t_2 & \textit{(application)} \\ & \mid & \lambda x.\ t & \textit{(}\lambda\textit{-abstraction)} \end{array}$$

λ-abstraction

A λ-term $(\lambda x.\ y)$ can be thought of as a function that, given an input bound to the variable x, returns the term y.
We will give a formal definition of this in terms of *substitution* later.

For now, we will extend λ-terms with arithmetic expressions:

$$(\lambda x.\ \lambda y.\ (x + y) \div 2)\ 10\ 20$$

but this is not fundamental to the computational model. We will remove this feature later without reducing expressivity.

Higher-order functions

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:

$$\lambda x.\ \lambda y.\ f\ y\ x$$

λ-calculus is *higher-order*, in that functions may be arguments to functions themselves:

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

α-equivalence

What is the difference between these two programs?

$$(\lambda x.\, \lambda x.\, x + x) \qquad (\lambda a.\, \lambda y.\, y + y)$$

α-equivalence

What is the difference between these two programs?

$$(\lambda x.\ \lambda x.\ x + x) \qquad (\lambda a.\ \lambda y.\ y + y)$$

They are semantically identical, but differ in the choice of bound variable names. Such expressions are called *α-equivalent*.

We write $e_1 \equiv_\alpha e_2$ if e_1 is α-equivalent to e_2. The relation $\equiv_\alpha$ is an *equivalence relation*.
The process of consistently renaming variables that preserves α-equivalence is called *α-renaming* or *α-conversion*.

Substitution

A variable x is *free* in a term e if x occurs in e but is not *bound* (by a λ-abstraction) in e.

Example (Free Variables)

The variable x is free in $\lambda y.\, x + y$, but not in $\lambda x.\, \lambda y.\, x + y$.

Substitution

A variable x is *free* in a term e if x occurs in e but is not *bound* (by a λ-abstraction) in e.

Example (Free Variables)

The variable x is free in $\lambda y.\, x + y$, but not in $\lambda x.\, \lambda y.\, x + y$.

A *substitution*, written $e[{}^{t}/_{x}]$, is the replacement of all free occurrences of x in e with the term t.

Example (Substitution on Arithmetic Expressions)

$(5 \times x + 7)\left[{}^{y \times 4}/_{x}\right]$ is the same as $(5 \times (y \times 4) + 7)$.

Problems with substitution

Consider these two α-equivalent expressions.

$$(\lambda y.\, y \times x + 7)\ 5$$

and

$$(\lambda z.\, z \times x + 7)\ 5$$

What happens if you naïvely apply the substitution $\left[{}^{y \times 3}/_{x}\right]$ to both expressions?

Problems with substitution

Consider these two α-equivalent expressions.

$$(\lambda y.\, y \times x + 7)\ 5$$

and

$$(\lambda z.\, z \times x + 7)\ 5$$

What happens if you naïvely apply the substitution $[{}^{y \times 3}/_{x}]$ to both expressions? You get two non-α-equivalent expressions!

$$(\lambda y.\, y \times (y \times 3) + 7)\ 5$$

and

$$(\lambda z.\, z \times (y \times 3) + 7)\ 5$$

This problem is called *capture*.

Variable Capture

Capture can occur for a substitution $e\left[{}^{t}/_{x}\right]$ whenever there is a bound variable in the term e with the same name as a free variable occuring in t.

Fortunately

It is always possible to avoid capture. Just α-rename the offending bound variable to an unused name.

β-reduction

The rule to evaluate function applications is called *β-reduction*:

$$(\lambda x.\ t)\ u \quad \mapsto_\beta \quad t\left[{}^{u}/_{x}\right]$$

β-reduction

β-reduction is a *congruence*:

$$\frac{}{(\lambda x.\ t)\ u \mapsto_\beta t\left[{}^{u}/_{x}\right]}$$

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

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

β-reduction

β-reduction is a *congruence*:

$$\frac{}{(\lambda x.\ t)\ u \mapsto_\beta t\left[{}^{u}/_{x}\right]}$$

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

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

Example:

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

β-reduction

β-reduction is a *congruence*:

$$\frac{}{(\lambda x.\ t)\ u \mapsto_\beta t\,[^{u}/_{x}]}$$

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

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

Example:

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

β-reduction

β-reduction is a *congruence*:

$$\frac{}{(\lambda x.\ t)\ u \mapsto_\beta t\left[{}^{u}/_{x}\right]}$$

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

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

Example:

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

β-reduction

β-reduction is a *congruence*:

$$\frac{}{(\lambda x.\ t)\ u \mapsto_\beta t\,[^u/_x]}$$

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

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

There are often many different ways to reduce the same expression:

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

Call-by-name

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

Evaluate function args
late
(after application)

Confluence

There are often many different ways to reduce the same expression:

$$\begin{tikzcd} & (\lambda a.\, a)\ ((\lambda y.\, f\ y)\ 5) \arrow[dl, "\text{Call-by-name}"'] \arrow[dr, "\text{Call-by-value}"] & \\ (\lambda y.\, f\ y)\ 5 & & (\lambda a.\, a)\ (f\ 5) \end{tikzcd}$$

Evaluate function args
late
(after application)

Evaluate function args
early
(before application)

Confluence

There are often many different ways to reduce the same expression:

$$
\begin{tikzcd}
 & (\lambda a.\ a)\ ((\lambda y.\ f\ y)\ 5) \arrow[dl, "\text{Call-by-name}"'] \arrow[dr, "\text{Call-by-value}"] & \\
(\lambda y.\ f\ y)\ 5 \arrow[dr] & & (\lambda a.\ a)\ (f\ 5) \arrow[dl] \\
 & f\ 5 &
\end{tikzcd}
$$

Evaluate function args **late** (after application)

Evaluate function args **early** (before application)

The Church-Rosser Theorem

If a term t β-reduces to two terms a and b, then there is a common term t' to which both a and b are β-reducible.

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 terms.

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 terms.

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 therefore uniqueness of normal forms), so we have a notion of *$\alpha\beta\eta$-equivalence* also.

Normal Forms

A term that cannot be reduced further is called a *normal form*

Normal Forms

A term that cannot be reduced further is called a *normal form*

Divergence

Does every term in λ-calculus have a normal form?

Normal Forms

A term that cannot be reduced further is called a *normal form*

Divergence

Does every term in λ-calculus have a normal form?

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

Try to β-reduce this!

Uniqueness of NFs

Does any term in λ-calculus have more than one normal form?

Normal Forms

A term that cannot be reduced further is called a *normal form*

Divergence

Does every term in λ-calculus have a normal form?

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

Try to β-reduce this!

Uniqueness of NFs

Does any term in λ-calculus have more than one normal form?
No: consider Church-Rosser.

Making λ-Calculus Usable

In order to demonstrate that λ-calculus is actually a usable 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?

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:

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

How do we write an if statement?

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:

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

How do we write an if statement?

$$\text{If} \equiv \lambda c.\, \lambda t.\, \lambda e.\, c\ t\ e$$

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:

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

How do we write an if statement?

$$\text{If} \equiv \lambda c.\, \lambda t.\, \lambda e.\, c\ t\ e$$

Example (Test it out!)

Try β-normalising If True False True.

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:

$$
\begin{aligned}
\text{Zero} &\equiv \lambda f.\, \lambda x.\, x \\
\text{One} &\equiv \lambda f.\, \lambda x.\, f\ x \\
\text{Two} &\equiv \lambda f.\, \lambda x.\, f\ (f\ x) \\
&\dots
\end{aligned}
$$

How do we write 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:

$$\begin{array}{lcl} \text{Zero} & \equiv & \lambda f.\, \lambda x.\, x \\ \text{One} & \equiv & \lambda f.\, \lambda x.\, f\ x \\ \text{Two} & \equiv & \lambda f.\, \lambda x.\, f\ (f\ x) \\ \dots & & \end{array}$$

How do we write 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:

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

How do we write Suc?

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

How do we write 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:

$$
\begin{aligned}
\text{Zero} &\equiv \lambda f.\, \lambda x.\, x \\
\text{One} &\equiv \lambda f.\, \lambda x.\, f\ x \\
\text{Two} &\equiv \lambda f.\, \lambda x.\, f\ (f\ x) \\
&\dots
\end{aligned}
$$

How do we write Suc?

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

How do we write Add?

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

Natural Numbers

Example

Try β-normalising Suc One.

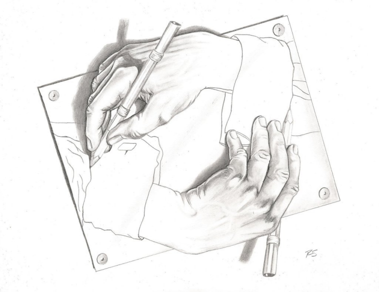
Example

Try writing a different λ-term for defining Suc.

Example

Try writing a λ-term for defining Multiply.

A Final Puzzle



Puzzle

Find a λ-term $\mathcal{Y}$ such that

$$\mathcal{Y}\, f \quad \mapsto_{\beta}^{\star} \quad f\,(\mathcal{Y}\, f)$$

Can you use this to define recursive functions? e.g. factorial?