

λ -Calculus: Introduction

- The λ -calculus was developed in the 1930s by Alonzo Church.
- It has subsequently been studied by many people (and still is).
- It is considered to be a foundational basis for functional and sequential programming languages.
- It can be used within Haskell, where a λ -abstraction $\lambda x.t$ is represented by `\x->t`.
- It is simple, powerful and easy to extend with features of interest.
- Landin pointed out how λ -calculus could be used to explain features of some existing programming languages:

“Whatever the next 700 languages turn out to be, they will surely be variants of λ -calculus”.

Landin 1966

Syntax of λ -Terms

- λ -terms are built up inductively in the following ways:
 - Variables, e.g. x , u and v .
 - Applications, of the form $s t$ where s and t are λ -terms, e.g. $f x$.
 - Abstractions, of the form $\lambda x.s$, e.g. $\lambda x.x$, $\lambda x.\lambda y.x$, $\lambda x.\lambda y.y$.

- This can be described by the following grammar:

$$Exp = Var \mid Exp Exp \mid \lambda Var.Exp$$

- Since the syntax is defined inductively, we can define things by *recursion* and prove things by *structural induction*.

Free Variables

- A *free variable* in a λ -term is one which is not bound to any corresponding formal parameter.
- The free variables of a λ -term t , denoted by $FV(t)$, can be defined recursively over the syntax of t as follows:

$$FV(x) = \{x\}$$

$$FV(st) = FV(s) \cup FV(t)$$

$$FV(\lambda x.s) = FV(s) \setminus \{x\}$$

- A λ -term is said to be *closed* if it does not contain any free variables.

Structural Induction: *FV* is Finite

- As an illustration of structural induction, we will prove that for any λ -term t , the set $FV(t)$ is always finite:
 - If t is a variable v , then $FV(t) = \{v\}$ by definition, and this is certainly finite.
 - If t is an application $s_1 s_2$, then by the inductive hypothesis $FV(s_1)$ and $FV(s_2)$ are both finite. But $FV(t) = FV(s_1) \cup FV(s_2)$, and the union of two finite sets is finite.
 - If t is an abstraction $\lambda x.s$, then by the inductive hypothesis $FV(s)$ is finite. But $FV(t) = FV(s) \setminus \{x\}$, which must also be finite as it is no larger.
- Q.E.D.



Substitution

- In order to express certain rules, we need to formalize the notion of substituting a term for a variable in another term. We write $t[s/x]$ for the result of substituting a term s for a variable x in another term t .
- This can be remembered by thinking of multiplication of fractions: $x[t/x] = t$
- For example: $(\lambda z.x + z + x)[y/x] = \lambda z.y + z + y$
- Of course we only substitute for *free variables*, so $(\lambda x.x)[y/x] = \lambda x.x$

Naive Substitution

- We could naively define substitution by recursion as follows:

$$\begin{aligned}x[t/x] &= t \\y[t/x] &= y, \text{ if } x \neq y \\(s_1 s_2)[t/x] &= s_1[t/x] s_2[t/x] \\(\lambda x.s)[t/x] &= \lambda x.s \\(\lambda y.s)[t/x] &= \lambda y.(s[t/x]), \text{ if } x \neq y\end{aligned}$$

- However this isn't right. We get $(\lambda y.x + y)[y/x] = \lambda y.y + y$ - *variable capture*.
- We should rename first to $\lambda w.x + w$, and only then perform the naive substitution:

$$(\lambda w.x + w)[y/x] = \lambda w.y + w$$

Renaming Substitution

- Consider two cases when defining $(\lambda y.s)[t/x]$ for $x \neq y$.
 1. If either $x \notin FV(s)$ or $y \notin FV(t)$, then we can proceed as before; variable capture will not occur.
 2. Otherwise, we pick a new variable $z \notin (FV(s) \cup FV(t))$ and form $\lambda z.(s[z/y][t/x])$. That is, first we rename the bound variable y to z , then proceed with the substitution as before.
- For definiteness, we can define z to be the lexicographically earliest variable not occurring free in s or t .
- The above definition ensures that substitution always respects the intuitive interpretation. Now we can use this operation freely.

Conversion Rules

- λ -calculus is based on three fundamental ‘conversions’ which transform one term into another one, intuitively equivalent to it.
- These are traditionally denoted by the Greek letters α (alpha), β (beta) and η (eta).
- The most important to us is β .
- α -conversion: $\lambda x.s \xrightarrow{\alpha} \lambda y.s[y/x]$ provided $y \notin FV(s)$. For example, $\lambda u.u v \xrightarrow{\alpha} \lambda w.w v$, but $\lambda u.u v \not\xrightarrow{\alpha} \lambda v.v v$. This restriction avoids another instance of name capture.
- β -conversion: $(\lambda x.s) t \xrightarrow{\beta} s[t/x]$
- η -conversion: $\lambda x.t x \xrightarrow{\eta} t$, provided $x \notin FV(t)$. For example $\lambda u.v u \xrightarrow{\eta} v$ but $\lambda u.u u \not\xrightarrow{\eta} u$.

λ -Equality

- We say that two λ -terms s and t are equal, and write $s = t$, if there is a finite sequence of α , β or η conversions, forwards or backwards, at any depth, which connects them.
- Note that this ‘equality’ is a defined notion and is not the same as equality at the syntactic level. We call syntactic equality ‘identity’, and use the symbol \equiv .
- For example $\lambda x.x = \lambda y.y$, but it is not the case that $\lambda x.x \equiv \lambda y.y$.
- The equality relation is *extensional*, i.e. if two functions f and g give equal results for all arguments, then they are themselves equal.



Evaluation

- The basic evaluation strategy in the λ -calculus is *reduction*.
- Any sub-expression of the form $(\lambda x.e_1) e_2$ is a suitable candidate for reduction (by β -conversion).
- Such an expression is called a *reducible expression* or *redex* for short.
- At any point in a reduction, there might be a choice of which expression to reduce next.
- One evaluation strategy we could use is to always select the *leftmost innermost* redex. This is called *applicative-order* reduction and corresponds to *call-by-value* evaluation. This is used in most programming languages.
- Another strategy is to always select the *leftmost outermost* redex. This is called *normal-order* reduction and corresponds to *call-by-name* evaluation. This is used by many functional programming languages such as Haskell.
- An evaluation is said to have completed when the expression contains no more redexes. The expression is then said to be in *normal form*.

Example 1

- Consider the following λ -term:

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

- The normal-order reduction of this is as follows:

$$\begin{aligned} \underline{\lambda f.f(f(f(a))) (\lambda x.g x)} &\Rightarrow \underline{(\lambda x.g x) ((\lambda x.g x) ((\lambda x.g x) a))} \\ &\Rightarrow \underline{g ((\lambda x.g x) ((\lambda x.g x) a))} \\ &\Rightarrow g (g ((\lambda x.g x) a)) \\ &\Rightarrow g (g (g a)) \end{aligned}$$

- The applicative-order reduction of this is as follows:

$$\begin{aligned} \lambda f.f(f(f(a))) \underline{(\lambda x.g x)} &\Rightarrow \underline{\lambda f.f(f(f(a))) g} \\ &\Rightarrow g (g (g a)) \end{aligned}$$

- The applicative-order reduction is therefore more efficient in this instance.

Example 2

- Consider the following λ -term:

$$(\lambda x.y) ((\lambda x.xx)(\lambda x.xx))$$

- The applicative-order reduction of this is as follows:

$$\begin{aligned}(\lambda x.y) \underline{((\lambda x.xx)(\lambda x.xx))} &\Rightarrow (\lambda x.y) \underline{((\lambda x.xx)(\lambda x.xx))} \\ &\Rightarrow (\lambda x.y) \underline{((\lambda x.xx)(\lambda x.xx))} \\ &\Rightarrow \dots\end{aligned}$$

- The normal-order reduction of this is as follows:

$$\underline{(\lambda x.y) ((\lambda x.xx)(\lambda x.xx))} \Rightarrow y$$

- The normal-order reduction therefore terminates while the applicative-order reduction does not.

The Church Rosser Theorem

- The Church Rosser theorem states that if $s \rightarrow t_1$ and $s \rightarrow t_2$, then there is a term u such that $t_1 \rightarrow u$ and $t_2 \rightarrow u$.
- By applying this repeatedly, we get the stronger form that if $t_1 = t_2$, then there is a term u with $t_1 \rightarrow u$ and $t_2 \rightarrow u$.
- So *if* a term reduces to, or in general is equal to, a term in normal form, that normal form is unique up to α -conversion.
- Also, a consequence of the theorem is that λ -equality is non-trivial, because two terms in normal form that are not α equivalent are not equal. For example $\lambda x y.x \neq \lambda x y.y$.
- Another theorem (also proved by Church and Rosser) is that if any reduction sequence terminates, then the one arrived at by systematically reducing the leftmost outermost redexes will terminate.