The Other Model: λ Calculus

So far we have only looked at Turing machines as our model.

Question: Where does the Church in the Church-Turing thesis come from?

Answer: Alonzo Church invented the λ calculus to model computing with functions.

Turns out the λ calculus is as powerful as the Turing machine, we are just not used to thinking in terms of universal computing using just functions.

The Other Model: λ Calculus

The next couple of lectures will show that the λ calculus and the Turing machine are equivalent in terms of computational power.

1. Introduce the λ calculus.
2. Introduce the primitive- and μ-recursive functions.
3. Show that the λ calculus implements the μ-recursive functions.
4. Show that a Turing machine implements the μ-recursive functions.
5. Show that a Turing machine implements the λ-calculus.
6. Show that μ-recursive functions can implement Turing machines.
7. Because of 3 we can conclude that λ calculus can implement Turing machines.
8. From 5 and 7 we conclude that the λ-calculus and Turing machines are computationally equivalent.

One way to look at the λ calculus is as a term rewriting system.

λ Calculus 101

λ-calculus aims to model computation with functions. At the core of this calculus are λ-expressions of the form

$$\lambda x.\, E$$

denoting functions with a parameter x and a function body E. Here is the variable x is assumed to be *free* in E (i.e. the variable is assumed not bound by a λ-operator).

The syntax for the calculus can be summarized by the following context-free grammar,

<function>	$::=$	λ<var>. <expression>
<expression>	$::=$	<var> $\mid$ <function> $\mid$ <application>
<application>	$::=$	<expression><expression>

Example:

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

λ Calculus 101

Rules: The calculus is very simple, it essentially consists of only three rules: α-conversion, β-reduction, and η-conversion.

Notation: Let E and E' be λ-expression and v a variable, then $E[E'/v]$ denotes the expression resulting from substituting E' for all occurrences of the variable v in expression E.

Extensions: We allow for a number of extensions in our calculus all of which can be implemented in classical λ-calculus but make the calculus easier to use:

- naming of functions
- pattern matching on structures
- constants

λ Calculus 101

α-conversion: This rule states that we are allowed to rename variables without changing the meaning of a function. Formally,

$$\lambda v.\, E = \lambda w.\, E[w/v],$$

as long as w does not appear freely in E and w is not bound by a λ in E whenever it replaces a v. Here v and w are variables and E is a λ-expression.

This gives rise to the following equivalences:

$$\lambda x.\, x = \lambda y.\, y$$
$$\lambda x.\, (\lambda x.\, x)x = \lambda y.\, (\lambda x.\, x)y$$

but note that

$$\lambda x.\, \lambda y.\, x \neq \lambda y.\, \lambda y.\, y \qquad \text{(Why?)}$$

λ Calculus 101

β-reduction: This rule expresses the idea of function application. Formally,

$$(\lambda v.\, E)E' = E[E'/v],$$

where v is a variable, E and E' are λ-expressions.

Example:

$$(\lambda n \in \mathbb{N}.\, n + 1)1 = 1 + 1 = 2$$

Note: When no more β-reductions are possible on a terms then we say that we have reached a *normal form*.

λ Calculus 101

η-conversion: This rule expresses the idea of extensionality, which in this context is that two functions are the same if and only if they give the same result for all arguments. Let M be a lambda expression then the η-conversion converts between M and $\lambda x.\, M x$ whenever x does not appear free in M.

Example: Let M be the lambda expression $\lambda n.\, n + 1$, then applying the η-conversion we have $\lambda x.\, (\lambda n.\, n + 1)\, x$.

We can formally show that $\lambda n.\, n + 1$ and $\lambda x.\, (\lambda n.\, n + 1)\, x$ are equivalent expressions:

$$\begin{array}{rcll} \lambda x.\, (\lambda n.\, n + 1)\, x & = & \lambda x.\, x + 1 & \text{(}\beta\text{-reduction)} \\ & = & \lambda n.\, n + 1 & \text{(}\alpha\text{-conversion)} \end{array}$$

λ Calculus 101

The β-reduction rule is perhaps obvious, since it actually allows us to compute the value of a function application, but what about the α-conversion? The rule states that renaming unbound variables does not change the nature of the function. Why is this useful? Consider the function $\lambda n \in \mathbb{N}.\, n + 1$ which is the successor function. Let's use this function to construct the function $\lambda n \in \mathbb{N}.\, n + 2$,

$$\lambda n \in \mathbb{N}.\, (\lambda n \in \mathbb{N}.\, n + 1)n + 1$$

Pretty confusing...let's use the α-conversion rule to rename the λ-bound variable of the inner successor function,

$$\lambda n \in \mathbb{N}.\, (\lambda k \in \mathbb{N}.\, k + 1)n + 1$$

More readable because the scoping of the λ-bound variables is now explicit.

λ Calculus 101

Examples: Compute the normal forms of the following λ-expressions:

1. $(\lambda x.\, x)\, 2$
2. $(\lambda x.\, (x, x))\, 2$
3. $(\lambda (x, y).\, (x, y, z))\, (5, 3)$
4. $(\lambda (x, y).\, y)\, (5, 3)$
5. $(\lambda x.\, (y, w))\, 2$
6. $(\lambda x.\, x)\, (\lambda x.\, x)$
7. $(\lambda x.\, (x\, x))\, (\lambda x.\, (x\, x))$
8. We let sequences of values be represented by $a :: b :: c :: []$, for example the string ‘fun’ would be the sequence of symbols $f :: u :: n :: []$, then what is the result of the following computation,
$$(\lambda x :: q.\, q)(f :: u :: n :: [])$$
9. We call the expression $c?\, x : y$ a conditional expression which returns x if c evaluates to true and it returns y if c evaluates to false. Given this, what does the following expression evaluate to: [a]
$$(\lambda xyz.\, x > 0?\, y : z)\, 3\, (\lambda q.\, q - 1)\, (\lambda p.\, p + 1)\, 1$$

[a] The notation $(\lambda xyz.\, E)$ is a short hand for the function $(\lambda x.\, \lambda y.\, \lambda z.\, E)$.

λ-calc computability

Example: We show that the λ-calculus can generate a Turing recognizable language that is not context-free,

$$L = \{a^n b^n (ab)^n \mid n \geq 0\}$$

We do this in two stages, first we build a function that given an index will generate the corresponding string and then we build an iterator that uses the generator to generate all the strings in the language (theoretically at least, it would loop forever).

$$\begin{array}{rcl}
\text{GEN} & = & \lambda n.\, \text{APPEND}\,(\text{APPEND}\,(\text{ASTRING}\,n)\,(\text{BSTRING}\,n))\,(\text{ABSTRING}\,n) \\
\text{APPEND} & = & \lambda xy.\,(x = [])?\,y : (\text{HD}\,x) :: (\text{APPEND}\,(\text{TL}\,x)\,y) \\
\text{HD} & = & \lambda x :: y.\, x \\
\text{TL} & = & \lambda x :: y.\, y \\
\text{ASTRING} & = & \lambda n.\,(n = 0)?\,[] : (a :: (\text{ASTRING}\,(n-1)) \\
\text{BSTRING} & = & \lambda n.\,(n = 0)?\,[] : (b :: (\text{BSTRING}\,(n-1)) \\
\text{ABSTRING} & = & \lambda n.\,(n = 0)?\,[] : (a :: b :: (\text{ABSTRING}\,(n-1)) \\
 & & \\
\text{ITER} & = & \lambda x.\,(\text{STEP}\,0\,[]) \\
\text{STEP} & = & \lambda nk.\,(\text{STEP}\,(n+1)\,((\text{GEN}\,n) :: k))
\end{array}$$

λ-calc computability

Notes:

- Convince yourself that you understand what the λ-expression GEN does. What does the expression return for an input of 0? 1? 2?
- What does the output of the λ-expression ITER look like?
- A set of strings generated by a *recursive* λ-expression is called *recursively enumerable language*. Furthermore, the set of all recursively enumerable languages is exactly the same set as the Turing-recognizable languages.[a]

[a] We will see that is true later when we show that the $lambda$-calculus and the Turing machine are computationally equivalent.

λ-calc computability

Example: To highlight the fact that we can construct algorithms at par with the Turing machine in the λ-calculus, let's us build a recognizer for the Turing-recongizable language

$$L = \{a^n a^n a^n \mid n \geq 0\}$$

We will make use of the fact that for any string $s \in L$ we have[a]

$$a^n a^n a^n = a^{3n}$$

for any $n \geq 0$. We construct the λ-expression R that will recognize this language,

$$\begin{aligned}
\text{R} &= \lambda x.\,(\text{CHECKA}\,x) \wedge (\text{DIV3}\,x) \\
\text{CHECKA} &= \lambda x.\,x = [\,]?\,\text{TRUE} : ((\text{H}D\ x) \neq a?\,\text{FALSE} : (\text{CHECKA}\,(\text{TL}\,x))) \\
\text{DIV3} &= \lambda x.\,x = [\,]?\,\text{TRUE} : ((\text{LENGTH}\,x) < 3?\,\text{FALSE} : (\text{DIV3}\,(\text{TL}\,(\text{TL}\,(\text{TL}\,x))))) \\
\text{LENGTH} &= \lambda x.\,x = [\,]?\,0 : 1 + (\text{LENGTH}\,(\text{TL}\,x)) \\
\text{HD} &= \lambda x :: y.\,x \\
\text{TL} &= \lambda x :: y.\,y
\end{aligned}$$

[a] Easily shown to be true by induction on n.