21-366 Lambda Calculus

Lecture Notes and Exercises

Brian Jacobs

April 6, 2015

Revision 1.04

Introduction

All sections marked with a * are extensions beyond the material taught in the course. It is included to make the notes feel more complete, cover questions which might come up after reading the course material, and explore ideas which did not arise during the lecture.

Disclaimer

*The following lecture notes are from the 2014-15 course 21-366 Lambda Calculus, taught by Professor Statman at Carnegie Mellon University. All sections not marked by a * are notes taken during the class. Any mathematical or typographic errors are my own. If you spot one, feel free to email me at* `brian@brianjaco.bz`.

Lecture 1

1.1 Introduction

A problem which arises frequently in mathematics is whether a particular statement is true or false. In some cases, this question is easy to answer. The statement $3 + 4 = 7$ can be demonstrated to be true by counting out three stones, then counting four more stones. The total number of stones in the pile can be counted as seven. Similarly, we can see that $3 + 4 = 8$ is false.

Some statements are less obviously true or false, however. Take the claim that there are a finite number of prime numbers. Consider a list of the first n prime numbers, $p_1, p_2, \ldots, p_n$. Take the product $P = 1 + \prod_{i=1}^{n} p_i$. Now there are two cases: either P is prime, or it is composite. If it is prime, then we have found a prime which is not in our list of the first n primes. If it is composite, then it must be divisible by some prime $q \neq 1$. We can see that if

$$q|(p_1 p_2 \cdots p_n + 1) \Rightarrow q|(p_1 p_2 \cdots p_n) \text{ and } q|1.$$

If q divides 1, then q must be 1, which is a contradiction, so q does not divide the list $p_1, p_2, \ldots, p_n$. This implies that q does not occur anywhere in the list. So we can show that for any list of primes, we can compute another prime which is not in the list. Since we can write an initial list of prime numbers, e.g. $2, 3$, this demonstrates that there are an infinite number of prime numbers. Our claim is false.

Divisibility Notation

If some number a divides some other number b (that is, if b/a is an integer) we write $a|b$.

If we had not been able to disprove our claim, it is not entirely clear that we could have argued that the claim definitely had to be provable one way or another. (We could have. But bear with me.) Consider a different problem: whether there exists an algorithm which, when given a computer program as input, determines whether the program runs forever, or eventually halts. This problem is known as the Halting Problem.

One attempt at a solution to this problem is to attempt to produce an algorithm which satisfies this specification. Naively, one could attempt to run the program until it either halts, or enters into an infinite loop. We can define an infinite loop in terms of program states. If a program has a repeating pattern of states, then it is in an infinite loop, and will not halt.

Unfortunately, this fails. Consider a program P which satisfies this specification. For some other program S and a string s, $P(S, s) =$true if and only if S does not halt on input s, and enters an infinite loop otherwise. What happens when we try to run $P(P, P)$? There are two possible cases. If P does not halt on P, then $P(P)$ is an infinite loop. However $P(P, P)$ returns false on the same input program, which is a contradiction. Conversely, if P halts on P, then $P(P)$ halts and $P(P, P)$ doesn't, which again is a contradiction. So no such program can exist.

Function Totality

A function is called **total** if it will return some value for every input. Our program P is not total.

In 1936, Alan Turing and Alonzo Church independently solved this problem. Turing invented the notion of a ``Turing Machine,'' which consists of an infinite tape, onto which symbols can be read and written. The machine can decide based on the current symbol and its current state what action to take next. The machine can overwrite the current symbol or move left or right one or more symbols on the tape. What Turing and Church hypothesized was that the class of functions which could be computed by Turing machines was somehow universal. This is the Church-Turing Thesis. When people talk casually about a problem being ``computable,'' they usually mean that the problem can be computed on a Turing machine.

As an aside, based loosely on the idea of Turing machines is the Von Neumann architecture for computers. Almost all modern computers are based on this architecture, or one which expands upon Von Neumann. This means that the question of whether a problem is solvable using Turing machines is equivalent to the question of whether the problem is solvable using modern computers.

Aside from Turing's machines, Chuch also came up with a model for computation. His model is the lambda calculus.

1.2 What is Lambda Calculus?

Lambda calculus is a system which allows us to explore the algebraic properties of computable functions, much in the same way that Turing machines do. It is at its heart fairly simple. There are some terms, which we will define below, and some methods for reducing those terms into different terms, which we will discuss at greater length over the next twenty or so chapters.

Turing machines map in an intuitive way to the programming paradigm of **imperative computation**. In both, an instruction pointer is bounced around and the contents of the memory at some designated location is manipulated based on the symbol at the pointer. In each case, program (or machine) state is key. Similarly, lambda calculus also maps to a programming paradigm. This paradigm is **functional programming.**

The mental model for functional programming (and lambda calculus) is that of reduction. A program consists of a series of definitions. The evaluation of the program is simply the reduction of those definitions until a final state is reached.

In the rest of the chapter, we introduce a collection of programs, which are the terms of Lambda calculus. Then we discuss the syntax of these terms.

1.2.1 Terms of λ calculus

Lambda **terms** are the mechanism by which we produce functions. $\lambda x X$ represents the substitution of x into X, where X is a term in terms of zero or more xs. For example, we could write the function $f(x) = x + 1$ as $\lambda x(1 + x)$. Obviously this requires us to define a number of things in terms of lambda calculus that we have not yet defined, like numbers and addition. For now, and until lecture 6, we will assert without evidence that such a thing is possible. This should be satisfactory for now as evidence that lambda calculus can compute things.

We have constants, which are names for particular functions, and particular kinds of data. We have variables, like x, y, and z. Church wanted to think about a domain where there is no distinction between data and functions. If X and Y are both terms, then (XY) is a term, read ``X applied to Y.'' We could alternately write this as Ap XY. If X is a term and x is a variable, then $(\lambda x\ X)$ is a term.

Alonzo Church

Church is best known for the introduction of lambda calculus and the Church-Turing thesis, among other things. He was born in 1903 in Washington D.C., and spent most of his life teaching at Princeton and UCLA.

Example: Lambda Terms

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

1.2.2 Currying

One intuitive problem that you might have with this model of computation is that there is not an obvious way to describe a function which takes multiple arguments. A model of compuation which cannot represent the addition of two numbers a and b would be a very poor model indeed! At first glance, this problem seems to apply to lambda calculus, with its single fundamental function $\lambda x(X)$.

To resolve this concern, note that if we have a function $\phi(a, b)$, then we can define a second function $\Phi(a)$ such that $\Phi(a)(b) = \phi(a, b)$. We conventionally write $\Phi(a)$ as Φ_a. In this way, we can convert a single function of many variables into many functions of a single variable each. So $\phi(a, b)$ becomes $\Phi_a(b)$.

Haskell Curry

Haskell's name has been immortalized both in the name of the functional programming language Haskell, and in the mathematical operation of currying: converting a function of many variables into a sequence of single variable functions called on each other.

1.2.3 Construction Trees

We can visually represent the recursive structure of a term by drawing a tree. We call this tree a construction tree. The construction tree allows us to see the subterms of any term. We define the tree of a constant or a variable is simply tree(x)$= x$. The tree of a term (XY) would be the tree with the trees of X and Y as subtrees. The tree of a lambda expression λxX is just the subterm X.

$$\begin{tikzcd}
\text{tree}(X) \arrow[dr, dash] & & \text{tree}(Y) \arrow[dl, dash] & & \text{tree}(X) \arrow[d, dash] \\
& (XY) & & & (\lambda xX)
\end{tikzcd}$$

Example: Construction Tree for $S := (\lambda x(\lambda y(\lambda z((XZ)(YZ)))))$

$$\begin{tikzcd}
X \arrow[dr, dash] & & Y \arrow[dl, dash] & & Y \arrow[dr, dash] & & Z \arrow[dl, dash] \\
& (XZ) \arrow[drr, dash] & & & & (YZ) \arrow[dll, dash] & \\
& & & ((XZ)(YZ)) \arrow[d, dash] & & & \\
& & & \lambda z \arrow[d, dash] & & & \\
& & & \lambda y\ \lambda z \arrow[d, dash] & & & \\
& & & \lambda x\ \lambda y\ \lambda z & & &
\end{tikzcd}$$

Trees Grow Up

In Computer Science, trees usually grow down the page from their root node. Our construction trees grow up.

1.3 Parenthesis

At this point in the discussion of the syntax of the lambda calculus, it is worth spending some time talking about parenthesis. We use parenthesis to make explicit and unambiguous what the construction tree for a term looks like. Otherwise, if we came across a term like λxXY, we wouldn't know whether to interpret it as $(\lambda xX)Y$ or $\lambda x(XY)$! On the other hand, when we begin to work with longer terms, our notation risks becoming cumbersome if every term must be written with all possible parenthesis. We spend some time now determining the minimum necessary parenthesis necessary to make a term unambiguous so as to save time manipulating cumbersome notation later.

1.3.1 Proper Pairings

A sequence of parenthesis is said to have a **proper pairing** if there exists a bijection ϕ from left parenthesis to right parenthesis satisfying two conditions:

- $\phi(l)$ is to the right of l.
- There are no crossings. $(_1(_2)_1)_2$ is bad. $()()$ is okay, and $(_1(_2)_2)_1$ is okay.

We claim that the parenthesis of a term are properly paired. We can prove this by induction on the construction tree of a term. If the term is of the form (x), then the parenthesis are properly paired, as there can be no parenthesis inside the variable x. If the term is of the form (XY), (X), or (λxX), then the outermost pair of parenthesis are properly paired if the parenthesis in X and Y are properly paired. We take the case of a single variable as our base case, and see that our induction hypothesis is satisfied, so we are done.

We can now also see that if a sequence of parens has a proper pairing, then the sequence is unique. We prove this again by induction. This time, we induct on the number of parenthesis, n. If $n = 1$, then it is clear that there is only one proper pairing: $()$.

Connections

A sequence of parenthesis can be thought of as a depth-first search of a tree, thinking of a left paren as taking an edge down, and a right paren as taking an edge up.

For $n > 1$ we know there must be at least one left paren which begins the sequence. We then have a sequence of zero or more left parenthesis. The first right parenthesis we encounter must be paired with the rightmost left parenthesis:

$$(\ldots(_m)_m$$

Observe that there is only one way that this first pairing can be made. $(_m$ must be paired with $)_m$. If this is not the case, then we would have a crossing, which is not allowed in a proper pairing. We remove the m parenthesis, and now we have $2(n-1)$ parenthesis in our pairing. We can then see that the claim is true for all n by induction.

1.3.2 Checking Whether a Pairing is Proper

There is a fairly simple algorithm for check whether a pairing is proper. Start with a number $x = 0$ at the left end of the sequence of parenthesis. Every time you encounter a left paren, increment x by 1. Every time you encounter a right paren, decrement x by 1. Throughout this process, if the pairing is proper n will never be negative. The process will terminate with $x = 0$ again.

We can prove the correctness of this algorithm as well by induction on the number of pairs of parenthesis, n. For $n = 1$, there are only two possible pairings: $()$ and $)($. The first pairing is correct. The sequence of values of x is $0, 1, 0$. The second pairing has a sequence of $0, -1, 0$, so our algorithm claims (correctly) that this pairing is not proper.

Now assume that the algorithm is correct for pairings of length $2n$. Our sequence of values for x satisfies the property that the first and last numbers are zero, and that there are no negative numbers in the sequence. TODO: FINISH THIS PROOF

Lecture 2

2.1 Construction Trees

During the previous lecture, we proved that proper pairings of parenthesis are unique, and that each term has a proper pairing. We also observed that each term has an associated construction tree. From this we can demonstrate that construction trees are unique.

2.1.1 Generating Construction Trees

It is nice to know that unique construction tree representations exist for terms, but it would be useful sometimes to know what those construction trees actually are. We can describe an algorithm that, when given a term, provides the construction tree for that term.

Note

Even though we omit parens sometimes when writing terms, when we prove statements involving parenthesis, we treat the terms as if no parens have been omitted.

In the most simple case, a term with no parens is a variable. Variables are terminal nodes in our construction tree. If there are parenthesis, then we begin with the leftmost left parent. Since the pairing is proper, this leftmost paren must be paired with the rightmost right paren. Otherwise, a crossing would have occurred. We then have one of three cases: either the first symbol is a variable, the first symbol is a left paren, or the first symbol is a λ. If the first symbol is a variable, then our term is of the form (xY), and has this construction tree:

$$\begin{tikzcd} x \arrow[dr, dash] & & Y \arrow[dl, dash] \\ & (xY) & \end{tikzcd}$$

x is terminal, so we induct on Y. Otherwise, if the next symbol is a left paren, we have a term of the form $((X)(Y))$. We can then induct on both X and Y with the following construction tree:

$$\begin{tikzcd} X \arrow[dr, dash] & & Y \arrow[dl, dash] \\ & (XY) & \end{tikzcd}$$

In our final case, we have that our term is of the form $(\lambda x X)$, which has the construction tree:

$$\begin{tikzcd} (X) \arrow[d, dash] \\ (\lambda x X) \end{tikzcd}$$

Again, we induct on X. From this we can generate a construction tree for any term.

Example: Construction Tree for $(\lambda x(X(xy)))$

We begin by using the rule for constructing a tree from a lambda term:

$$\begin{tikzcd}
((X)(xy)) \arrow[d, dash] \\
(\lambda x(X(xy)))
\end{tikzcd}$$

We then use the rule for application terms:

$$\begin{tikzcd}
(X) \arrow[dr, dash] & & (xy) \arrow[dl, dash] \\
& ((X)(xy)) \arrow[d, dash] & \\
& (\lambda x(X(xy))) &
\end{tikzcd}$$

Finally, we treat the X path as completed, since we havedn't defined what X is, and use the application rule once more on the (xy) path:

$$\begin{tikzcd}
& & x \arrow[dr, dash] & & y \arrow[dl, dash] \\
& (X) \arrow[dr, dash] & & (xy) \arrow[dl, dash] & \\
& & ((X)(xy)) \arrow[d, dash] & & \\
& & (\lambda x(X(xy))) & &
\end{tikzcd}$$

2.2 Some Definitions

It is somewhat cumbersome to always have to write ``a term of the form (XY)'' when we want to talk about subtermsof an application. When we have a term of the form (XY), we say that X is in **function position**, and Y is in **argument position**. We also say that Y is applied to X.

2.2.1 Subterms

Let Y be a term. Y is a subterm of X if Y is also a substring of X, with the exception of a variable following a λ.

Trichotomy Law: Given any two subterm occurances[1] Y', Z' of a term X, then either they are disjoint or one contains the other.

2.2.2 Free and Bound Variables

In a lambda term $(\lambda x.X)$, we say that the variable x is **bound** by the term, because the contents of the term will be determined by the argument provided to the lambda. Variables which are not

[1]In this course, subterm occurances will be notated with primes.

bound in this way are called **free.** It is possible for variables to be free in one term, but bound by a term which occurs higher in the construction tree. It should be clear, for example, that the term x has one free variable: x itself. However in the term $\lambda x.X$, it is bound by the lambda.

So, what happens to bound and free variables when we have a term of the form (XY)? Since there are no lambdas introduced, the status of the variables cannot change. The variables which were bound remain bound, and the variables which are free remain free. We can summarize the rules for determining which variables are bound and which are free as follows:

Summary of Bound and Free Variables

$$\begin{aligned} FV(x) &= \{x\} \\ BV(x) &= \emptyset \\ FV((XY)) &= FV(X) \cup FV(Y) \\ BV((XY)) &= BV(X) \cup BV(Y) \\ FV((\lambda x X)) &= FV(X) - \{x\} \\ BV((\lambda x X)) &= BV(X) \cup \{x\} \end{aligned}$$

It is useful when attempting to determine which variables of a term are bound and which are free by hand, to draw the construction tree for the term. It is obvious what variables in the roots of the tree are bound, and which are free. From there, the recursive definitions given above can be employed.

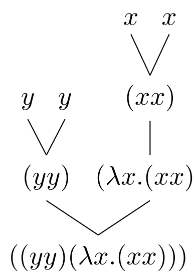
Example: Determining the Bound Variables of a Term

Say that we have a term $((xx)(\lambda x.(xx)))$ for which we want to determine the free variables. We begin by drawing the construction tree:

$$\begin{tikzcd}
 & & & x \arrow[d, dash] & x \arrow[dl, dash] \\
x \arrow[d, dash] & x \arrow[dl, dash] & & (xx) \arrow[d, dash] & \\
(xx) \arrow[dr, dash] & & & (\lambda x.(xx)) \arrow[dll, dash] & \\
 & ((xx)(\lambda x.(xx))) & & &
\end{tikzcd}$$

All of the root nodes of the tree are terms x, which have free variables x. Both terms of (xx) are of the form (XY), so their free variables are the union of the free variables of their subtrees, or x. The lambda term $\lambda x.xx$ binds x, so x is no longer free. The top term inherits the free and bound variables of its subtrees once more, so the bound variables are x, and the free variables are... x.

The example above illustrates a subtlety which is easy to miss. The variable x in the left subtree is not the same as the variable x in the right subtree. We wrote them as the same symbol, but in actuality they are wholly unrelated. If we represent the xs in the left subtree as ys instead, the situation makes itself clear.



The free variables are y, and the bound variables are x.

2.3 Minimizing Parenthesization

We've talked a bit so far about ways to reduce the number of parenthesis needed to unambiguously determine the construction tree of a term. Now we will formally introduce three strategies for removing parenthesis: Polish notation, Church's dot notation, and left associative deletion.

Jan Łusckasevicz

Jan was born in 1878 in Poland. He worked in analytical philosophy and logic. He is best known for Polish notation.

2.3.1 Polish Notation

Polish Notation

Polish notation is sometimes referred to in other contexts as ``prefix notation.'' Equivalently, there is a ``postfix notation'' which is sometimes called Reverse Polish Notation.

Polish notation was invented by Jan Łuckasevicz in 1924. We use only the symbols λ and $($, plus variables x, y, z and terms X, Y. The key feature of Polish notation is that operators are placed before their arguments. Take, for example, the expression $3 + 4$. The operator in this expression, the $+$ occurs between the arguments, $3, 4$. In Polish notation, we would instead write $+34$. In this simple case it is not immediately obvious what is gained by this. When we try it on a larger expression, it becomes clear what the advantage is:

$$((3+4) \times (2+2))/(1+5) \Rightarrow / \times +3\,4 + 2\,2 + 1\,5$$

Notice how by rearranging the expression, we can remain unambiguous without parenthesis. We can adopt a similar style for writing lambda terms. We use the terms $\lambda, (, x, y, z$ where x, y, z are variables, and X, Y are terms.

Example: Lambda Terms in Polish Notation

$$(XY$$
$$\lambda x X$$

We can notice that terms written using Polish notation can be fairly straightforwardly derived from terms which do not, simply by omitting the right parens. Although this does significantly reduce the number of necessary parenthesis, it also makes the terms much more difficult to read, so we will not actually use this method in practice.

2.3.2 Church's Dot Notation

Willard Van Orman Quine

Born in 1908, the American philsopher and logic Quine spent his life at Harvard University.

Another convention we can use is a dot notation developed by Quine, called Church's dot notation. Using this notation, we take a term with some number of initial lambdas, and remove the parens around all of them, placing a dot after the last one. So if we have an expression of the form

$$(\lambda x_1(\lambda x_2(\lambda x_3 \ldots (\lambda x_n X) \ldots)))$$

We can write it as

$$(\lambda x_1 x_2 \ldots x_n.X)$$

This makes it much easier to write out functions which take in multiple arguments as lambda terms. It is worth mentioning currying again here. Writing a curried function in dot notation might make it seem like we are again working with functions which take in multiple arguments again. We are not.

Example: Dot Notation Representation of S

$$S := (\lambda x(\lambda y(\lambda z((xy)(yz))))) = \lambda xyz.((xy)(yz))$$

2.3.3 Left Association

We can delete all of the parenthesis around applications where the subterm is not in argument position. So in a term (XY), we can delete parens around X, if any occur. We also retain the outermost set of parenthesis in a formal sense, although we will tend to omit them in practice. So if $X = (UV)$ and we have a term $((UV)Y)$ we would delte the parenthesis around (UV) to get (UVY).

Example: Left Association Representation of S

$$S := (\lambda x(\lambda y(\lambda z((xy)(yz))))) = (\lambda x(\lambda y(\lambda z(xy(yz)))))$$

It is not immediately obvious that this strategy for deleting parenthesis preserves uniqueness. This is proven in the next lecture.

2.4 Proper Extensions

We say that a string s_1 is a proper extension of some string s_2 if s_1 can be written as the concatenation of s_2 and another nonempty string. The string ``Hello world'' is therefore a proper extension of the string ``Hello'' by the concatenation of the extra string `` world''. We claim that no term is a proper extension of another term as a string read from left to right.

We can prove this by induction on the sum of the lengths of the two sequences. The smallest sequence is a variable x. No term can be an extension of a variable because a left paren is necessary. Now consider a term X with an extension Y. There are two possible cases: either X is of the form $X = \lambda uU$ or $X = (UV)$.

If $X = \lambda uU$, then the only way to extend X is if Y begins with λu, which can only be extended into a term of the form λuV. Since we claim by our induction hypothesis that V is not a proper extension of U for ally U, V smaller than X,Y, and U must necessarily be an extension of U, we have a contradiction.

In our other case, we have that $X = (UV$. Therefore Y must begin with a paren, and so be of the form $Y = (ZW$. This implies that ZW is an extension of UV. By our induction hypothesis, $U = Z$ and $V = W$, so there can be no extension. So construction trees are unique.

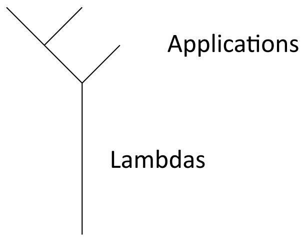
Lecture 3

3.1 Left Association Deletion of Parens

In the previous lecture, we asserted that we could take all of the parenthesis around terms which were not in argument position away without violating the uniqueness of the term. Let X be a term. We define X^* as the result of the left associative deletion of parenthesis from X. We show here what left associative deletion looks like for various terms. Later, in our uniqueness proof, we will refer back to these example terms.

$$
\begin{aligned}
(1) &= x \\
(2) &= (\lambda x_1 \ldots x_n . x_i) \\
(3) &= (\lambda x_1 \ldots x_n . ((\lambda x X_0) X_1) \ldots) X_m)) \\
(4) &= (\underbrace{\lambda x_1 \ldots x_n}_{\text{Lambda prefix}} \underbrace{. ((\ldots (\underbrace{x_i}_{\text{head}} X_i) \ldots) X_m)}_{\text{matrix}}
\end{aligned}
$$

Illustration of 3 and 4



Left association deletion for case (1) is trival. There are no parenthesis at all, so we delete nothing. In case (2), we have a sequence of lambdas followed by a single application. This does not have any parenthesis around subterms which are not in argument position, so again we cannot delete any parenthesis. Case (3) is the first case where we can delete something. The term at the head of the application is of the form $((\lambda x X_0) X_1)$, which can be reduced to $(\lambda x X_0 X_1)$. In case 4, the first subterm is of the form $((x_i X_i) \ldots)$. The innermost parens here can be removed, giving another term which has parenthesis around the subterm not in argument position. By repeating this process, $(4*)$ is derived.

$$
\begin{aligned}
(1*) &= x \\
(2*) &= (\lambda x_1 \ldots x_n . x_i) \\
(3*) &= (\lambda x_1 \ldots x_n . (\lambda x X_0 X_1 \ldots X_m) \\
(4*) &= (\lambda x_1 \ldots x_n . x_L X_1 \ldots X_m)
\end{aligned}
$$

3.1.1 Proof of Uniqueness

Our claim is that $X \neq Y$ if and only if $X^* \neq Y$. There are four possible cases for the form of our terms X and Y. We will prove the claim by induction on X. Our base case is X as a variable. It is either of the form $((x)y)$ or $(x(y))$, or a single variable z, where x, y and z could have subterms.

Our first case is that $X := z$. It has no parens, so we are done. If it is of the form $((xy)z)$ or $(x(yz))$, however, we can evaluate $((xy)z)*$ and $(x(yz))*$:

$$\begin{aligned} ((xy)z)* &= xyz \\ (x(yz))* &= x(yz) \end{aligned}$$

We can say that we can uniquely restore the parens, based on whether we are in the first or second listed case.

Induction Step: In case (2), we have two parens. This can only happen the two parens are the outermost parens around the term:

$$(\lambda x_1 \ldots x_n . X_{i_1} \ldots X_{i_k})$$

In our left association deletion scheme, we do not delete outermost parens, so the term remains unchanged. If all of the variables can be uniquely restored, then case (2) can be uniquely restored as well.

$X^* = Y^*$ in cases where $(1)^*, (2)^* \Rightarrow X = Y$.

Consider the proper pairing of parens. Case (2) applies to X implies (2) applies to Y and case (3) applies to X implies (3) applies to Y.

$$(\lambda x_1 \ldots x_n (\lambda x X_0))(\quad)(\quad))$$

The parens in argument position must be reconstructable by the induction hypothesis. FIGURE OUT AND FIX THIS PROOF.

3.2 Substitution of a Term for a Variable

A problem arises when we attempt to substitute one term into another where both terms contain the same variables. The problem is that it is ambiguous which variables mean what when both are represented by the same symbol. Take for example the following application:

$$\begin{gathered} (\lambda x.yx)(zx) \\ (\lambda x.y(zx)) \end{gathered}$$

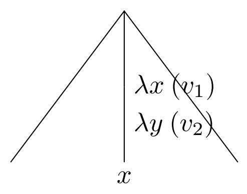
When we make this application, we have xs both before and after the application, but the xs mean different things. In this simplified example, it can be figured out what is going on by manual inspection, but it gets worse when the terms are more complicated. We therefore have an incentive to resolve this naming issue before then. An easy solution is to simply change the name of the symbol in the lambda:

$$\begin{gathered} (\lambda u.yu) \\ (\lambda u.(zx)u) \end{gathered}$$

TODO: THIS IS WRONG. WHY.

3.2.1 Clash Graphs

We have some abstract term. From the root, there is only one path to any given leaf.



We define a **clash graph** which has verticies that are the λ abstraction nodes of the term. v_1 is adjacent to v_2 provided that v_2 lies on the path from v_1 to a leaf which it binds, or vice versa.

Lecture 4

4.1 Substitution

We have been talking about the idea of substitution for a while now, so it's time to formally describe what we mean. Let X be a term with subterm x. We say that we are substituting Y for x in X to mean that we are replacing every occurance of x in the construction tree of X with Y. We notate this with Curry's substitution prefix: $[X/x]Y$ read ``substitute X for each free occurance of x in Y.'' We call the substitution of a single term a **single substitution**.

Effects of Substitutions

$$\begin{aligned}
[X/x]x &= X \\
[X/x]y &= y \\
[X/x](UV) &= ([X/x]U[X/x]V) \\
[X/x](\lambda uU) &= \lambda u[X/x]U \quad (u \neq x) \\
[X/x](\lambda xU) &= \lambda xU
\end{aligned}$$

It is possible to perform multiple substitutions at once. We call this **simultaneous substitution**. A simultaneous substitution is notated by $[X_1/x_1, \ldots, X_n/x_n]Y$. We can also set an order for the substitutions which are performed by writing an expression of the form $[X_{n-1}/x_{n-1}]([X_n/x_n]Y)$, which we call a **sequential substitution**.

It is important to note that simultaneous and sequential substitutions do not necessarily produce the same result. This is because the first substitution to be applied is then substituted by the second, and so on.

$$[U/u]([V/v]Y) \quad = \quad [U/u, [U/u]V/v]Y$$

Example: A Substitution both Simultaneous and Sequential

Take the substitution of X/x and Y/X into the term x. We can see that the simultaneous substitution is

$$[X/x, Y/X]x = X$$

while the sequential substitution is either

$$[X/x]([Y/X]x) = X$$

or

$$[Y/X]([X/x]x) = Y$$

depending on the order in which the substitutions are applied.

4.2 What to do about Bound Variables?

We have the clash graph of an abstract term. Verticies are occurances of λ's in the term. An edge is between two adjacent λ's. What does it mean to assign new variables to the λ's and maintain the same abstract term such that no two λ's that are adjacent get the same variable? It affects all the variables which are bound by that λ.

Any free variables can be bound by adding more λ's at the top. This is called a **closure**.

We can think of renaming the variables of the term $\lambda[X/x]U$ as attempting to color a graph. How many colors do we need?

A **clique** in a graph is a set of verticies such that any 2 are adjacent. If a graph has a clique of size n, then you need n colors to color the graph. The converse of this statement is not true. Take for example the graph of a pentagram. It has cliques of size 2, but requires 3 colors to color it.

4.2.1 How to rewrite a term's bound variables

Notice that if we take any λ which is closest to the frontier[1], then all things which are adjacent to it lie above it. Now if we take any clique in the clash graph, and take any λ in that clique which is as close to the frontier as possible, all things adjacent to it lie above it. Additionally, all of them are adjacent to each other.

We take the largest clique in the clash graph, and take the lowest possible λ and remove it. We now have a term in one less λ. Suppose now that we can rewrite the entire term with the λ deleted. We can do this by rewriting the bound variables recursively. If we have another clique of size n, we can color our lowest λ with the color x and we then have to color the rest of the clique with $n-1$ colors, giving us n total colors. If we do not have another clique of size n, then we can color our lowest λ with x and again color the rest with $n-1$ colors. Thus our maximum number of colors is the size of the largest clique in the clash graph.

[1] That is, that it has no λ's below it.

4.3 Equivalence Relations on Terms

Intuitively, an equivalence relation on a set is a function $f : (a, b) \mapsto \{\text{true}, \text{false}\}$ which is true only if a is ``equal'' to b in some way. More precisely, if the set is partitioned in some way, then an equivalence relation is true only if the two elements are in the same subset of the partition.

An equivalence relation f satisfies the following properties:

- **Identity Property:** An element a is equivalent to itself.
$$(f(a, a) = \text{true})$$
- **Symmetric Property:** If and only if an element a is equivalent to some other element b, then b is equivalent to a.
$$f(a, b) = f(b, a)$$
- **Transitive Property:** If a is equivalent to b and b is equivalent to c, then a is equivalent to c.
$$f(a, b), f(b, c) \Rightarrow f(a, c)$$

4.3.1 Alpha Conversion

Two terms X and Y are α equivalent if one can be derived from the other by changing bound variables. This derivation is called α conversion. The simplest case of alpha conversion is that where $X = \lambda xX$ and $Y = \lambda yX$. It is clear that you can convert X into Y by simply replacing the bound variable x with the bound variable y. An α conversion which changes only a single bound variable is called an elementary α conversion.

The α equivalence relation clearly satisfies the identity property. Any term can be converted into itself by changing no bound variables. It also satisfies the symmetric property. Any α conversion can be undone, and the conversion which undoes it proves the relation in the opposite direction. That is, a conversion from X to Y can be reversed to create a conversion from Y to X by changing the variables back. Finally, the α equivalence relation satisfies the transitive property. Take some terms X, Y and Z such that X and Y are alpha equivalent and Y and Z are alpha equivalent. This implies that there is a conversion from X to Y and a conversion from Y to Z. We can convert X into Y and then into Z using those two conversions, demonstrating that X can be converted into Z. This satisfies the equivalence between X and Z.

α conversions are not particularly difficult to grasp or work with, so this is the only time we will really talk about them. It is useful, though, to know that the names we give to things do not affect how they behave.

4.3.2 Eta Conversion

Our second equivalence relation is η equivalence. Two terms U and V are η equivalent if (without loss of generality) $U = \lambda x(Xx)$ and $V = X$. A conversion from U to V is called an η reduction, and a conversion from V to U is called an η expansion.

η expansion and reduction

$X \to_\eta \lambda x(Xx)$	$\lambda x(Xx) \to_\eta X$
η expansion	η reduction

For the sake of our equivalence relation, we say that a term is η equivalent to itself.

Like with α equivalence, we are not going to talk about η conversion much, at least for the first half of the course. In the second half, it will come up again. This is because there really isn't much to η conversion.

4.3.3 Beta Conversion

We hinted earlier at the fact that we could create something analogous to functions in lambda calculus. The mechanism by which this happens is called β conversion, and it allows substitution in terms. The equivalence relation for β conversion is based on the following transformation:

$$(\lambda xX)Y \to_\beta [Y/x]X$$

In this transformation, we call $(\lambda xX)Y$ the **redex**. Curry called the $[Y/x]X$ term the **reducant** or **contractant**, but we will refer to them much, if at all.

Remark: General β conversion. We define as a β redex as the term $(\lambda xX)Y$. Curry defined $[Y/x]X$ as the reductant or contractant. We will not use these terms. β reduction can be represented as:

$$(\lambda xX)Y \to_\beta [Y/x]X$$

We allow ourselves to do this inside of bigger terms. We can β reduce a small term inside a bigger term to β reduce the entire term.

The equivalence relation β conversion is based on the following: $(\lambda xX)Y =_\beta [Y/x]X$. This is analogous to the example of having a polynomial $p(x) = a_0 + a_1x + \ldots + a_nx^n$, and evaluating it at a point:

$$\text{Eval}(p(x), \pi) = a_0 + a_1\pi + a_2\pi^2 + \ldots + a_n\pi^n$$

Lecture 5

5.1 Reductions

Note that we hide α conversions while we do more interesting things. We will also concentrate on β reduction. η reduction is somewhat trivial.

5.1.1 β Reduction

$$\underbrace{(\lambda x X)Y}_{\text{Redex}} \to_\beta \underbrace{[Y/x]X}_{\text{reductum}}$$

$$\to \text{contraction} \to$$

5.1.2 η Reduction

$$\underbrace{\lambda x(Xx)}_{\text{redex}} \to_\eta \underbrace{X}_{\text{reductum}} \qquad\qquad x \notin FV(X)$$

Additionally note that we call it ``reduction,'' but it is entirely possible that the reductum is equal in length to or longer than the redex. For example:

$$\begin{array}{rcl} 1)\ (\lambda xx)Y & \to_\beta & Y \\ 2)\ (\lambda xy.x)YZ & \to_\beta & (\lambda yY)Z \\ (\lambda x(\lambda yx))Y & \to_\beta & [Y/x](\lambda yx) = \lambda yY \\ & \to_\beta & Y \end{array}$$

5.1.3 ω and Ω

We now introduce a pair of term ω and Ω which often come up as the counterexamples to otherwise true claims in lambda calculus.

$$\begin{array}{rcl} \omega & := & \lambda x(xx) \\ \Omega & := & (\omega\omega) = (\lambda x(xx))(\lambda x(xx)) \end{array}$$

Definition: If $FV(X) = \emptyset$, then X is said to be **closed,** or a **combinator.** (Here we can also say that X is a closure.) Now we define a step beyond Ω:

$$\Omega^+ := (\lambda x(xxx))(\lambda x(xxx))$$

What happens if we apply Ω^+ to itself?

$$\Omega^+\Omega^+ \to_\beta (\Omega^+\Omega^+)\Omega^+ \to_\beta \Omega^+\Omega^+\Omega^+\Omega^+$$

We can also refer to situations where the redex is inside a larger term as a beta reduction. Imagine that U has a subterm $(\lambda x X)Y$. We can write that $U \to_\beta V$ when V has a subterm $[Y/x]Y$. So beta reduction can occur on subterms.

Interestingly, if you are given U and V, and that $U \to_\beta V$, you cannot tell what subterms are being reduced. The (essentially only) counterexample is Ω:

$$x\Omega\Omega \to x\Omega\Omega$$

Now we want to generalize this to many steps. This is called a **transitive closure.** We notate transitive closure as

$$U \twoheadrightarrow V$$

if there exists a sequence $U = U_0 \to_\beta U_1 \to_\beta \cdots \to_\beta U_n = V$. n can be zero, in which case $U = V$, and we allow U to reduce to itself. This is conventional, and it doesn't hurt anything.

We say that $U =_\beta V$ (read as U is beta equivalent to V) if there exists a sequence $U = U_0 \cdots U_{n-1} U_n = V$ such that

$$U_0 \twoheadrightarrow_\beta U_1 \twoheadleftarrow_\beta U_2 \twoheadrightarrow_\beta U_3 \cdots \twoheadleftarrow_\beta U_{n-1} \twoheadrightarrow_\beta U_n = V$$

Note that beta equivalence is transitive:

$$U =_\beta V, V =_\beta W \Rightarrow U =_\beta W$$

Example of beta equivalence:

$$(\lambda xy.x)\Omega\Omega \twoheadrightarrow_\beta \Omega \twoheadleftarrow_\beta (\lambda xy.x)\Omega\Omega^+$$

We can therefore say that $(\lambda xy.x)\Omega\Omega =_\beta (\lambda xy.x)\Omega\Omega^+$.

TODO: MAKE SURE K IS DEFINED IN A NOTICEABLE WAY. IT IS REFERENCED LATER.
Another example: Let $K := \lambda xy.x$ and $I := \lambda x.x$

$$\omega K \omega \twoheadrightarrow_\beta (KK)\omega \twoheadrightarrow_\beta K \twoheadleftarrow_\beta IK$$

A third example: xXY. If $X \to_\beta X^+$ and $Y \to_\beta Y^+$.

$$xXY^+ \twoheadleftarrow_\beta xXY \twoheadrightarrow_\beta xX^+Y$$

5.2 Properties of Beta Equivalence

Beta equivalence satisfies the properties of an equivalence relation.

- $U =_\beta U$
- if $U =_\beta V$ then $V =_\beta U$
- if $U =_\beta V$, $V =_\beta W$, then $U =_\beta W$

We then have the following properties:

1. $(\lambda x X)Y \twoheadrightarrow_\beta [Y/x]X$
2. if $X \twoheadrightarrow_\beta Y$, $U \twoheadrightarrow_\beta V$, then $(XU) \twoheadrightarrow_\beta (YV)$.
3. if $X \twoheadrightarrow_\beta Y$, then $(\lambda x X) \twoheadrightarrow_\beta (\lambda x Y)$.

Similar properties hold for $=_\beta$ as do for $\twoheadrightarrow_\beta$.

5.3 Redex Creation

We take a term U with subterm $(\lambda x X)Y$, and create a term V with subterm $[Y/x]X$. Old redexes have 0 or more residuals in V. Now we peek into X: $(\lambda x X)Y$ is actually $((\lambda x(\lambda z Z))Y)W$. We can contract it into $(\lambda z[Y/x]Z)$.

5.4 The Church-Rosser Theorem

Does the order in which we perform reductions on a lambda expression matter? This is an important question. If different orders of reduction could be applied to lambda terms to achieve different results, this would have uncomfortable implications for lambda calculus as a model for computation.

Imagine that we had a term X which reduced to two separate terms X_1' and X_2' depending on the order in which reductions are applied. We must have that $X_1' \neq X_2'$, otherwise we could simply reduce one into the other. But we also have that $X = X_1'$ and $X = X_2'$. By transitivity, this implies that $X_1' = X_2'$, which contradicts our assumption. This contradiction would be a problem. The proof that this contradiction does not occur was published in 1936 by Alonzo Church and J. Barkley Rosser.

Church and Rosser's Paper

Church and Rosser published this proof as ``Some Properties of Conversion'' in the journal *Transactions of the American Mathematical Society*. It is available at `http://www.jstor.org/stable/1989762`

The Church-Rosser Theorem

If $X =_\beta Y$, then there exists a term Z such that $X \twoheadrightarrow_\beta Z$ and $Y \twoheadrightarrow_\beta Z$. Graphically, we can draw this relationship as:

$$\begin{tikzcd}
 & X_1 & & X_3 & & X_{n-2} & & X_n & \\
X \arrow[ur] \arrow[drrrr, dashed, two heads] & & X_2 \arrow[ul] \arrow[ur] & & \cdots \arrow[ul] \arrow[ur] & & X_{n-1} \arrow[ul] \arrow[ur] & & Y \arrow[ul] \arrow[dllll, dashed, two heads] \\
 & & & & Z & & & &
\end{tikzcd}$$

In the above picture, solid arrows represent existing β conversions, which might be empty (that is, they might be of the form $X \rightarrow_\beta X$.) Dashed arrows represent β conversions which are implied by the theorem.

5.4.1 Proof of Church-Rosser

Our proof of the Church-Rosser theorem is based on the intuition that if we could somehow convert ``peaks'' in our diagram into ``valleys,'' we could construct the reductions $X \twoheadrightarrow_\beta Z$ and $Y \twoheadrightarrow_\beta Z$.

We first look at techniques for creating new redexes from old redexes. We have some term U such that $\Delta := (\lambda x X)Y$ is a subterm of U. We can contract with beta reduction this term into a V such that $[Y/x]X$ is a subterm of V. Now consider another subterm Γ. There are three cases:

Case 1: $\Delta \subset \Gamma$
Γ' reappears in V as a redex we call the *residual* of Γ.

Case 2: Δ disjoint from Γ
If they are disjoint, then Γ is contained in U and V, and Γ is its own residual.

Case 3: $\Gamma \subset \Delta$
Subcase 3.1: $\Gamma \subseteq Y$
Γ may have many residuals in V. There is also the possibility that x does not occur at all in X, and there are no residuals. So Γ can have zero or more residuals in V.
Subcase 3.2: $\Gamma \subseteq \lambda x X$
Γ now looks like $[Y/x]\Gamma$. If Γ has x as a free variable, then the residual of Γ is $[Y/x]\Gamma$.

Lecture 6

6.1 Computing With λ Terms

To perform computations, we need data structures. How do we represent data structures? We can use collections of terms that look like data structures. Our target for this lecture is to define the nonnegative integers, booleans, and finite sets. Once we have described these data structures, we can define some operations on them. Specifically, we will define addition and subtraction for the integers, and the conditional operator which branches between two terms dependant on whether a boolean is true or false.

6.2 Data Structures

6.2.1 Booleans

The way that we define booleans at first may seem a little bit strange. The reason that we define them in the way that we do is motivated by the common use cases for booleans. Often, we want to use booleans inside conditional statements, to decide which of two branches to take. We might write, for example, the following program:

```
int age = keyboard.read();
if (age >= 21) {
  print("Welcome to the bar!");
}
else {
  print("Come back when you are older.");
}
```

One way to do this is to establish an application (XY) where X is the term you would like to evaluate in the case where your boolean is true, and Y is the term you would like to evaluate in the case where your boolean is false.

So, in our above example, we would have a term representing the function `print("Welcome to the bar!")` as X and `print("Come back when you are older.")` as Y. We could then represent the booelan true as the function $(\lambda xy.x)$ and the boolean false as the function $(\lambda xy.y)$. We define $K := \lambda xy.x$ as our true boolean, and $K_* := \lambda y.xy$ as our false boolean.

Typically, if we have some term T, we write $T_* := TI$. This is where our notation for K_* is derived. We can demonstrate that $K_* =_\beta KI$ by β conversion:

$$KI = (\lambda xy.x)I \to_\beta Iy = (\lambda yI) =_\beta \lambda y\lambda zz =_\beta \lambda yz.z =_\alpha \lambda xy.y = K_*$$

6.2.2 Integers

The integers are substantially more difficult to represent than booleans. We need some manner of term which can be straightforwardly added, subtracted, and so on. When you factor in the possibility of negative numbers, the difficulty becomes unnecessarily high for an introductory work. So we will restrict ourselves to the nonnegative integers $0, 1, 2, \ldots$. To avoid confusion, we will write conventional numbers as usual, and numbers which we mean to be represented by our integer data structure with an underline, as in $\underline{n}$.

A good starting point is to consider the most simple problem which we face with our integer representation: counting. If our technique for incrementing an integer by one is complicated, it is a sign that the rest of our representation will be even more so. One way that we can represent numbers which turns out to be very nice for incrementing is to take some arbitrary term a and repeat it n times. These numbers are called the **Church numerals**.

$$\underline{n} = \underbrace{a \cdot a \cdot \ldots \cdot a}_{n \text{ times}} = a^n$$

This is a nice, clean representation of the number n. One downside is that it fails fairly catastrophically for all numbers less than 1. Usually, we would write that $a^0 = 1$, but 1 does not have meaning in the context of repeating a term a. $a^{-n} = 1/a^n$ is even worse, from the standpoint of lambda expression representation. This is why we ignore negative numbers for now.

The solution to the problem of representing zero is actually fairly simple. We can just declare that $a^1 = 0, a^2 = 1$, and so on. So we in fact have that:

$$\underline{n} = \underbrace{a \cdot a \cdot \ldots \cdot a}_{n+1 \text{ times}} = a^{n+1}$$

All that we are missing now is the actual term which represents zero. For this, we will use K_*. Ordinarily, we would try to motivate this choice and explain in some way why we made the decision. For now, we will have to satisfy ourselves with the knowledge that the decision will make some complicated constructions more simple down the line.

We now know enough to write a general version of the number $\underline{n}$ as a term. Notice that our term is not precisely a term applied to itself over and over again. Our analogy to exponentiation of a number a motivates our strategy, but is not perfect.

$$\underline{n} := \lambda xy.(\underbrace{x(x(\ldots x}_{n}\, y)) \ldots)$$

The next step down our rode to addtion is a function $\underline{s}$ which takes a number $\underline{n}$ to a number $\underline{n+1}$. We call this function **successor**. In our function notation, we write that:

$$\underline{S}\ \underline{n} \twoheadrightarrow_\beta \underline{n+1}.$$

The definition of the function S follows naturally from the definition of the Church numerals. We simply want to add another x to the chain.

$$\underline{S} := \lambda nxy.x((nx)y)$$

A Few Church Numerals

$\underline{0}$	$=$	$\lambda xy.y$
$\underline{1}$	$=$	$\lambda xy.xy$
$\underline{2}$	$=$	$\lambda xy.xxy$
$\underline{3}$	$=$	$\lambda xy.xxxy$
$\underline{4}$	$=$	$\lambda xy.xxxxy$
	$\ldots$	

Example: Applying $\underline{S}$ to $\underline{0}$ to get $\underline{1}$

$$\begin{aligned}
\underline{S0} &=_\beta (\lambda nxy.x(nxy))(\lambda xy.y) =_\beta (\lambda n(\lambda x(\lambda y.(xnxy))))(\lambda xy.y) \\
&\to_\beta (\lambda x(\lambda y.(x(\lambda xy.y)xy))) \to_\beta (\lambda x(\lambda y.xy)) \\
&=_\beta \lambda xy.xy =_\beta \underline{1}
\end{aligned}$$

There is another method which we alternately could have taken to get to $\underline{S}$. Our definition adds a new x to the left side of the list of xs. But we could alternately add a new x to the right side, instead. We have access to both ends of the list of xs, but not to the middle.

6.2.3 Finite Sets

There are some times when we want to implement a function which is best described by an explicit mapping between its domain and codomain. Ideally, we would want to write a function which can map from any set into any other set. To avoid the complexity of dealing with potentially infinite terms, however, we will restrict ourselves to mappings between finite sets.

Arbitrary Mappings

This problem shows up a lot in digital systems engineering. Imagine a finite state machine with arbitrarily assigned labels, and functions which take those labels as inputs to determine the next state to transition into. As you might imagine, these sorts of problems start to look like random mappings after a while.

Take for example the finite sets $A = \{1, 2, 3\}, B = \{12, 17, 9\}$, with a function f between them such that

$$f(1) = 12 \qquad f(2) = 17 \qquad f(3) = 9$$

One way that we can look at this problem is to think of the first set as an index into the second set as a list. If we want to find out what $f(a)$ is for some general number $a \in A$, we simply take the ath element of B. Notice that we can also compute $f^{-1}(b)$ for some $b \in B_L$, where B_L is the list representation of the set B. We simply count from the start of B_L until we reach b. So if we create a function f by this process, it necessarily implies the existance of the inverse, f^{-1}.

That this inverse exists allows us to argue that we can in fact construct functions which take arbitrary finite sets to other arbitrary finite sets. Consider now the new function w such that

$$w(7) = 12 \qquad w(4) = 17 \qquad w(6) = 9$$

How can we write this function? One way is to define a function g as below, and compose it with f:

$$g(7) = 1 \qquad g(4) = 2 \qquad g(6) = 3$$

$$\begin{tikzcd}
W \arrow[r, "g^{-1}"] \arrow[dr, dashed, "w = f \circ g^{-1}"'] & A \arrow[d, "f"] \\
 & B
\end{tikzcd}$$

This demonstrates that it is possible to write functions from finite sets to other finite sets, assuming that we can actually write functions like f and f^{-1} in lambda calculus. We begin with the function which takes a set $\{1, \ldots, n\}$ and map them to list of terms of the form $X_1 \ldots X_n$.

It its simplest form, this function takes elements of the set $\{1, 2\}$ and maps them into a list of length 2. We represent 1 and 2 with the terms:

$$\begin{aligned}
K_2^1 &:= \lambda x_1 x_2.x_1 \\
K_2^2 &:= \lambda x_1 x_2.x_2
\end{aligned}$$

So our function consists of a list $\langle X_1, X_2 \rangle$, and takes either of the above functions. In general, we call terms like $\lambda x_1 x_2 . x_1$ K_i^n, where n is the number of terms in the list, and i is the index of the element we want from inside that list. We can generally define K_n^i as the term

$$K_n^i := \lambda x_1 \dots x_n . x_i$$

So now let us say that we have a finite set $\{1, \dots, n\}$, and we want to create the function $f : \{1, \dots, n\} \to \{x_1, \dots, x_n\}$. Hopefully, we can describe a term T_f such that when we apply it to a term K_i^n, it is be beta equal to $K_{f(i)}^n$.

$$(T_f K_i^n) =_\beta K_{f(i)}^n$$

This now leaves us with the question of defining T_f. An easy way is to simply repeat our table lookup function scheme:

$$T_f := \lambda a . a(K_{f(1)}^n K_{f(2)}^n \dots K_{f(n)}^n)$$

Example: Demonstration of T_f

$$T_f K_i^n \to_\beta K_i^n K_{f(1)}^n K_{f(2)}^n \dots K_{f(n)}^n \twoheadrightarrow_\beta K_{f(i)}^n$$

Any function $f : \{1, \dots, n\} \to \mathbb{S}$ can be created by this lookup table strategy. But what about functions of the form $f : \{1, \dots, n\} \times \{1, \dots, n\} \to \{1, \dots, n\}$, or $f(i, j)$? To create functions of more than one variable, we can simply curry the inputs by definining $f_i(j) := f(i, j)$, and redefining $T_f := \lambda a . a T_{f_1} T_{f_2} \dots T_{f_n}$.

We will defer discussion of the inverse functions until we have a chance to understand techniques for recusion and bounded search. For now, it is sufficient to say that they exist.

Lecture 7

7.1 Operations on Data Structures

Now that we have defined some data structures, the next natural step is to perform calculations on them. We establish a number of integer operations, and describe how to use booleans to implement branching. We complete a simple programming environment by describing a strategy for implementing recursion.

7.1.1 Integer Addition and Subtraction

What is the addition of two integers? Since we represent our integers in the form

$$\lambda xy.\underbrace{xx\ldots x}_{n}y$$

it stands to reason that we want a function $\underline{+}$ such that

$$\underline{+}(\lambda xy.\underbrace{xx\ldots x}_{n}y)(\lambda xy.\underbrace{xx\ldots x}_{m}y) \twoheadrightarrow_\beta (\lambda xy.\underbrace{xx\ldots x}_{n+m}y)$$

So our function $\underline{+}$ can simply take one number and insert it into the second. So we can define $\underline{+}$ as:

$$\underline{+} := \lambda uv.\lambda xy.(ux(vxy))$$

Example: Using $\underline{+}$

$$\underline{+}\underline{3}\underline{2} \quad =_\beta \quad (\lambda uv.(\lambda xy.(ux(vxy))))()()$$

We can see that addition will never result in a number which is not in our set of (nonnegative) integers. A problem arises, however, when we attempt to define subtraction. Defining subtraction as the inverse of addition is unsatisfactory, as many expressions involving subtraction would just not have solutions. To keep subtraction closed on the set of nonnegative integers, we define **monus subtraction** as

$$n \dot{-} m = 0 \text{ if } m \geq n \text{ else } n - m$$

Monus subtraction is not exactly the integer subtraction that we are used to, but it solves the problem of $A - B$ being undefined whenver $B > A$. In the next lecture, we will explain out how to perform monus subtraction using lambda terms.

Recall our definition of the monus subtraction function:

$$n \dot{-} m = \begin{cases} 0 & \text{if } m \geq n \\ n - m & \text{if } m < n \end{cases}$$

We can redefine this function in terms of itself, recursively:

$$n \dot{-} m = \begin{cases} n \dot{-} 0 = n \\ n \dot{-} (m+1) = \text{pred}(n \dot{-} m) \end{cases}$$

7.1.2 Integer Multiplication

In the last lecture, we introduced methods for representing nonnegative integers, and for adding such integers. A natural next step is to determine a method for computing multiplication. By this point, it should be relatively clear how to proceed.

We can define multiplication as

$$\underline{\cdot} := \lambda uvu(\underline{+}v)\underline{0}$$

We have an example:

$$\underline{\cdot}\,\underline{n}\,\underline{m} \twoheadrightarrow_\beta \underline{n}(\underline{+}\,\underline{m}) \twoheadrightarrow_\beta \underbrace{\underline{+}\,\underline{m}(\underline{+}\,\underline{m}(\ldots(\underline{+}}_{n}\ \underline{m}\,\underline{0})\ldots) \twoheadrightarrow_\beta \underline{n\cdot m}$$

There exists a simpler, alterantive definition:

$$\underline{\otimes} := \lambda uv.u(vx)$$

Again we compute an example for clarity:

$$\underline{\otimes}\underline{n}\,\underline{m} \twoheadrightarrow_\beta \lambda x(\,\underline{n}\,(\underline{m}\,x)) \twoheadrightarrow_\beta (\lambda x\lambda w\underbrace{(\,\underline{m}\,x(\ldots(\,\underline{m}}_{n}\ xw))))$$

This gives us n nested redexes of the form $\lambda yx(\ldots(xy)\ldots)(\,\_\,)$

7.1.3 The Conditional

We now return to our definition of the boolean. Recall that we want to use boolean values to determine which branch to take in a conditional statement. In general, we want $\supset KXY =_\beta X$ and $\supset K_*XY =_\beta Y$ for some definition of $\supset$, which we call the conditional. You might notice now that we have done something sneaky. Let us reduce the expression KXY.

Reduction of KXY

$$KXY =_\beta (\lambda xy.x)(XY) \rightarrow_\beta X$$

Huh. It turns out that we can simply use the identity term, I, as our conditional. Why do we need the conditional operator at all, then? The answer is that there is more than one way that we can represent K and K_*, and not every way that we choose reduces as nicely. Much later, we will find it useful to retain the same structure for terms which we expect to do similar things, so that we can more easily prove that they actually are related.

7.1.4 Basic Boolean Operators

Now that we can branch on individual booleans, we can consider cases where we want to condition on some logical combination of multiple boolean values. To do this, we can implement the basic building blocks of Boolean algebra.

There are three main operators which we should implement. The AND operator, the OR operator, and the NOT operator. In computer engineering, these operators are called **logic gates**. In logic, they are called **logical connectives**. We can think of them as functions of the form $f : B \times B \rightarrow B$, where B is the set $\{1, 0\}$ of boolean values.

Boolean Algebra

Boolean Algebra was invented by George Boole, a philosopher and logician, for use in determing the truth or falsity of statements.

We can easily define the functions in terms of **truth tables**, which are simply a way of enumerating all possible inputs to a boolean function, and listing their output. Conventionally, we notate the inputs to the function with capital letters starting with 'A', and notate the output with the letter 'S'. True is 1 and 0 is false. The truth tables for our three operators are as follows.

AND

A	B	S
0	0	0
0	1	0
1	0	0
1	1	1

OR

A	B	S
0	0	0
0	1	1
1	0	1
1	1	1

NOT

A	S
0	1
1	0

Note that these definitions are essentially similar to the English language meanings of the words and, or, and not. Take the statement: ``we need to buy eggs and we need to buy milk.'' If we do not buy both eggs and milk, we have failed in our task.

Our simplest operator is NOT. Whenever the input to NOT is true, the output to NOT is false, and vice versa. This can be implemented in lambda calculus as:

$$\lambda b. \supset bK_*K$$

We can demonstrate the correctness of this expression by exhaustive testing. As an aside, this is a good, simple example of the technique presented in the section about finite sets for constructing functions from a lookup table.

Proof of Correctness for NOT

$$\begin{aligned} \text{NOT}K &=_\beta (\lambda b. \supset bK_*K)K \to_\beta \supset KK_*K \to_\beta (\lambda xy.x)(K_*K) \to_\beta K_* \\ \text{NOT}K_* &=_\beta (\lambda b. \supset bK_*K) \to_\beta \supset K_*K_*K \to_\beta (\lambda xy.y)(K_*K) \to_\beta K \end{aligned}$$

We now present the implementations and correctness proofs for AND and OR. Understanding these types of reductions is good practice. Producing these correctness proofs on your own is a good, if quick exercise. These reductions are more explict than strictly necessary, to allow readers who have fallen behind a chance to catch up.

AND and OR

$$\begin{aligned} \text{AND}b_1b_2 &:= \lambda b_1b_2. \supset b_1b_2K_* \\ \text{OR}b_1b_2 &:= \lambda b_1b_2.b_1Kb_2 \end{aligned}$$

Proof of Correctness for AND and OR

$$\begin{aligned} \text{AND}KK &=_\beta (\lambda b_1b_2. \supset b_1b_2K_*)KK \to_\beta \supset KKK_* \to_\beta (\lambda xy.x)KK_* \to_\beta K \\ \text{AND}KK_* &=_\beta (\lambda b_1b_2. \supset b_1b_2K_*)KK_* \to_\beta \supset KK_*K_* \to_\beta (\lambda xy.x)K_*K_* \to_\beta K_* \\ \text{AND}K_*K &=_\beta (\lambda b_1b_2. \supset b_1b_2K_*)K_*K \to_\beta \supset K_*KK_* \to_\beta (\lambda xy.y)KK_* \to_\beta K_* \\ \text{AND}K_*K_* &=_\beta (\lambda b_1b_2. \supset b_1b_2K_*)K_*K_* \to_\beta \supset K_*K_*K_* \to_\beta (\lambda xy.y)K_*K_* \to_\beta K_* \end{aligned}$$

$$\begin{aligned} \text{OR}KK &=_\beta (\lambda b_1b_2. \supset b_1Kb_2)KK \to_\beta \supset KKK \to_\beta (\lambda xy.x)KK \to_\beta K \\ \text{OR}KK_* &=_\beta (\lambda b_1b_2. \supset b_1Kb_2)KK_* \to_\beta \supset KKK_* \to_\beta (\lambda xy.x)KK_* \to_\beta K \\ \text{OR}K_*K &=_\beta (\lambda b_1b_2. \supset b_1Kb_2)K_*K \to_\beta \supset K_*KK \to_\beta (\lambda xy.y)KK \to_\beta K \\ \text{OR}K_*K_* &=_\beta (\lambda b_1b_2. \supset b_1Kb_2)K_*K_* \to_\beta \supset K_*KK_* \to_\beta (\lambda xy.y)KK_* \to_\beta K_* \end{aligned}$$

7.2 Recursion

Remark: The 1^{st} definition of $\underline{\cdot}$ gives us an idea for doing recursion. Given any term F we can iterate F T times as follows:

$$F^{+} := \lambda x.xFT$$

then

$$F^{+}\underline{n} \twoheadrightarrow_{\beta} (\underbrace{F(\ldots(F}_{n}T))\ldots)$$

We can do better. We can define a more advanced form of recursion on integers:

$$F^{+}\underline{n+1}T =_{\beta} F(F^{+}\underline{n})T$$

Note that

$$F^{+}\underline{0}T =_{\beta} T$$

To make this work, we need an auxiliary function $\mathbb{N}$ $\rightarrow$Booleans. We define the sign function and its converse as

$$sg(n) = \begin{cases} \text{TRUE} & n \neq 0 \\ \text{FALSE} & n = 0 \end{cases} \qquad \overline{sg}(n) = \begin{cases} \text{FALSE} & n \neq 0 \\ \text{TRUE} & n = 0 \end{cases}$$

Now we define these functions as lambda expressions:

$$\begin{aligned} \underline{sg} &:= \lambda u.u(KK)K_{*} \\ \underline{\overline{sg}} &:= \lambda u.u(K_{*}K_{*})K \end{aligned}$$

We observe that this as well works:

$$\underline{sg}\ \underline{n} \rightarrow_{\beta} \underline{n}\,(KK)K_{*} \twoheadrightarrow_{\beta} (\underbrace{KK)(\ldots}_{n}(KK)K_{*})\ldots) \twoheadrightarrow_{\beta} K$$

7.2.1 Computing Predecessor

Start with two stacks of 1's: $0,0$. Assume that at stage t we have t,pred(t). At stage $t+1$, we have $t+1,t$. After n iterations, you have n,pred(n).

Here is another method. Represent pred by $\underline{\text{pred}}$. We need a two stack data structure:

$$\lambda a.a\ \underline{n}\ \underline{m}$$

We initialize this structure as $\lambda a.0\ 0$. We then want to apply some sort of operation to this data structure u times:

$$\lambda u.u \boxed{} (\lambda a.a\ \underline{0}\ \underline{0}\ )$$

But what function are we applying in the box?

$$\boxed{} := \lambda z.(\lambda b.b(\underline{s}\ (zk))(zk))k_{*}$$

Now we apply it:

$$\underline{\text{pred}}\ \underline{n}\ \rightarrow_{\beta} \underline{n}\ (\lambda z(\lambda b.b(\underline{s}\ (zk))(zk)))(\lambda a.a\ \underline{0}\ \underline{0})$$

Notice now that we can simulate a 2 register machine.

Remark: $\underline{\otimes}$: $\lambda uv\lambda x.u(vx) = B$. The B stands for ``beweis,'' which is German for ``proof.'' Now let us take $BFG \twoheadrightarrow_{\beta} \lambda x\ (F(Gx))$, or $(BFG)x \twoheadrightarrow_{\beta} F(Gx)$. Notice that B is a combinator.

7.3 Combinators

S stands for ``substitution.'' $C$ stands for ``commute.''

$$\begin{aligned}
I &:= \lambda x.x \\
K &:= \lambda xy.x \\
K^* &:= \lambda xy.y \\
S &:= \lambda xyz.xy(yz) \\
B &:= \lambda xyz.x(yz) \\
C &:= \lambda xyz.xzy
\end{aligned}$$

But now we want a combinator which does this:

$$\lambda x \lambda y_1 \dots y_n . x y_{\pi(1)} \dots y_{\pi(n)}$$

where π is a permutation on $\{1, \dots, n\}$.

We define one last combinator which, given a function $f : \mathbb{N}^2 \to \mathbb{N}$ gives a function $f^W(n) = f(n,n)$. This function diagonalizes across the naturals.

$$W := \lambda x \lambda y \; xyy$$

Lecture 8

8.1 Lists

The next data structure which we should construct is the list. Intuitively, it seems as though we should just be able to write out a set of lambda terms one after another, and call that a list. This intuition is not far off. We define the structure of a list as:

$$\langle X_1, \dots, X_n\rangle = \lambda a.aX_1 \dots X_n$$

To be able to construct lists within lambda calculus, we can define an operator C_* which takes a term to its list. As before, we are using the convention that for some term X, $X_* := XI$.

$$C_* := (\lambda cba.(cab))I$$

Demonstration of C_*

$$C_*X \quad =_\beta \quad (\lambda cba.(cab))IX \to_\beta \lambda ba.(Iab)X \to_\beta \lambda a.aX$$

8.1.1 Concatenation

Now that we have a method for constructing lists of length 1, we can use them to construct lists with greater length. We can think of the list of two elements as two one-element lists joined together. This joining process is called **concatenation**. We write the concatenation of two lists X and Y as $X \frown Y$. We can repeat this concatenation process multiple times to build up lists of arbitrary length.

We can use this idea of list concatenation on our list structure by defining a combinator which can take the elements of one list and append them to the second. As it turns out, the combinator B does this.

B as the list concatenation combinator

Let α and β be lists of the form $\alpha = \lambda a.a\alpha_1, \dots, \alpha_n = \langle\alpha_1, \dots, \alpha_n\rangle$ and $\beta = \lambda a.a\beta_1 \dots \beta_m = \langle\beta_1, \dots, \beta_m\rangle$.

$$\begin{aligned}
B\alpha\beta \quad &=_\beta \quad \lambda xyz.x(yz)\alpha, \beta \to_\beta \lambda z.\alpha(\beta z) \\
&=_\beta \quad \lambda z.(\lambda a.a(\alpha_1 \dots \alpha_n))((\lambda a.a(\beta_1 \dots \beta_m))z) \\
&\to_\beta \quad \lambda z.(\lambda a.a(\alpha_1 \dots \alpha_n))(z(\beta_1 \dots \beta_m)) \\
&\to_\beta \quad \lambda z.z(\beta_1 \dots \beta_m\alpha_1 \dots \alpha_n)
\end{aligned}$$

One seeming problem with this scheme is that while we defined $\langle X_1 \dots X_n \rangle \frown \langle Y_1 \dots Y_m \rangle$ as $\langle X_1 \dots X_n Y_1 \dots Y_m \rangle$, when we converted into lambda terms, we ended up with

$$B\langle X_1 \dots X_n \rangle \langle Y_1 \dots Y_m \rangle = \langle Y_1 \dots Y_m X_1 \dots X_n \rangle.$$

This looks backwards! The key word, however, is ``looks.'' Our notation here is slightly misleading. We have written lists of the form $\langle X_1 \dots X_n \rangle$ interchangeably with lists of the form $\lambda a.aX_1 \dots X_n$, but if you construct a list entirely by concatenating lists of length one, you will notice that they are constructed backwards, as $\lambda a.aX_n \dots X_1$.

Example: Constructing a List

$$\begin{aligned} \langle XYZ \rangle &= \langle X \rangle \frown \langle Y \rangle \frown \langle Z \rangle = B((B(C_*X)(C_*Y))C_*Z) \\ &\twoheadrightarrow_\beta B((\lambda a.aYX)C_*Z) \twoheadrightarrow_\beta \lambda a.aZYX \end{aligned}$$

Since the order that a list is written on the page is really just a convention, the inversion of the convention does not affect anything substative, other than allowing us to use B for concatenation instead of a more complicated combinator.

8.1.2 Length: An Inefficient Approach

This convention allows us to easily access the last element of a list. We can simply use a combinator like K to ake the first term from the list and project away the rest. But what happens if we want to access the first element?

Heads and Tails

We call the first element of a list its **head** and the second element its **tail**.

If we know that the length of the list is n, we can just remove the tail of the list n times until we are left with the first element. If we do not know the length, however, this is not possible. So one option is that we could define a new list data structure which somehow encodes its length, perhaps like this: $\langle n \langle X_1, \dots, X_n \rangle \rangle$.

The problem with this is that it is complicated. To concatenate two lists we would have to remove their lengths, add them together, concatenate the lists, and then concatenate on the new length. We could no longer use combinators like C_* and B. A better approach will be discussed after we develop fixed point combinators.

8.2 Stacks

Another way to solve the problem of needing list length is to introduce a new data structure: the stack. We define first the empty stack as E, and the stack with one element X_1 as $\lambda a.aX_1E$. If L is a stack of length n, then we can create a stack of length $n+1$ by we simply create a new list $\langle X_{n+1}, L \rangle = \lambda a\, aX_{n+1}L$. If we expand this list, we get:

$$\lambda a\, aX_{n+1}(\lambda a\, aX_n(\lambda a\, aX_{n-1}(\dots(\lambda a\, aX_1E)\dots)))$$

We call this operation ``push.'' Analogously we define a ``pop'' operation and a ``tail'' operation as

$$\begin{aligned} \langle X_{n+1}, L \rangle K &=_\beta X_{n+1} \\ \langle X_{n+1}, L \rangle K_* &=_\beta L \end{aligned}$$

But what happens if we attempt to pop from an empty list? We need to test for an empty list. Our test should take a list and if it is nonempty, return a K. We can do this with a test:

$$TEST : \lambda u\, u(\lambda zwv.K)(\lambda ab.K_*)$$

To make this happen, we can define $E = \lambda ab.K_*$.

To confirm that this works, we will test on the list $< X, L >$.

$$\begin{aligned} TEST\langle X, L\rangle &\to_\beta \langle X, L\rangle(\lambda zwv.K)(\lambda ab.K_*) \to_\beta (\lambda zwv.K)XL(\lambda ab.K_*) \twoheadrightarrow_\beta K \\ TEST\ E &\twoheadrightarrow_\beta E(\lambda xwv.K)(\lambda ab.K_*) \twoheadrightarrow_\beta K_* \end{aligned}$$

8.3 Unbounded Search

How do we search for a number? For example, say that we want to find a prime number in the list of natural numbers. In general, we write that we are looking for an n such that $\Phi(n)$ is true. We can see a pattern of cases:

$$\begin{aligned} \Phi(0) &\to 0 \\ \Phi(0), \Phi(1) &\to 1 \\ \Phi(0),\ \Phi(1), \Phi(2) &\to 2 \\ &\dots \end{aligned}$$

More generally, we can define a function:

$$f(x) = \begin{cases} n & \text{if } \Phi(n) \\ f(n+1) & \text{otherwise} \end{cases}$$

8.3.1 Gödel's Fixed Point Theorem

Gödel's Fixed Point Theorem asserts that

If a term T has a fixed point F such that $(TF) =_\beta F$, then F in some way has to depend on self application, as the statement $\exists F \mid (TF) =_\beta F$ is not true in general for all T. This implies that $T(XX) = XX$. We can write:

$$\begin{aligned} T(XX) &= (XX) \\ (\lambda x T(XX))X &= (XX) \\ X &= \lambda x\, T(XX) \\ (\lambda x\, T(XX))(\lambda x\, T(XX)) &=_\beta T((\lambda x\, T(XX))(\lambda x\, T(XX)) \\ X &=_\beta TX \end{aligned}$$

We define this as Curry's Paradoxical Combinator. It is a fixed point combinator.

$$Y_{Curry} := \lambda y\ (\lambda x\ (y(xx)))(\lambda x.y(xx))$$

We can apply Y_{Curry} to T.

$$\begin{aligned} Y_{Curry}T \to_\beta (\lambda x\, T(xx))(\lambda x\, T(xx)) \\ \to_\beta T((\lambda x\, T(xx))(\lambda x\, T(xx))) \leftarrow_\beta T(Y_{Curry}T) \end{aligned}$$

We can define as well Turing's Fixed Point Combinator:

$$Y_{Turing} = \underbrace{(\lambda xy.y(xxy))}_{\alpha}\underbrace{(\lambda xy.y(xxy))}_{\alpha}$$

What happens when we apply Y_{Turing} to T?

$$Y_{Turing}T \to_\beta (\lambda y\ y(\alpha\alpha y))T \to_\beta T(\alpha\alpha T)$$

It turns out that Y_{Turing} is much more useful, but Y_{Curry} is easier to determine.

8.4 * Practical Uses for Y, the Fixed Point Combinator

One problem which arises from our current construction of λ calculus is that we do not have a mechanism for recursion. When we usually think of recursion, we think of it as a function which calls itself. This is not possible yet. Notice that we would need some way of referring to a function from inside itself. We can do this by passing a function to itself as an argument. Y allows us to evaluate an expression with itself as an argument. Remember that

$$\text{Y g} \twoheadrightarrow_\beta \text{g (Y g)}$$

By repeatedly applying this β reduction, we can have g applied to itself as many times as we want.

Take, for example, a function for computing the factorial of a number, fac. In pseudocode, we can implement fac tail recursively as

```
def fac(n):
    if (n == 1):
        1
    else:
        n * fac(n - 1)
```

Lecture 9

9.1 Lists of Integers

Our goal now is to find the length of a list of integers.
Recall that:

$$Y_{Curry} = \lambda y((\lambda x\ y(xx))(\lambda x\ y(xx)))$$
$$Y_{Turing} = (\lambda xy.y(xxy))(\lambda xy.y(xxy))$$

$$\begin{aligned} \langle x_1, \dots, x_n \rangle &= \lambda a.aX_1 \dots X_n \\ \text{length}(\langle x_1, \dots, x_n \rangle) &= n \end{aligned}$$

If we take our representation of nonnegative integers, and define a new encoding such that $n = n' + 1$, we can get a representation of nonnegative integers using only positive integers. This leaves us with zero as a special value, which we can use to mark the end of a list. We can then encode a list of nonnegative integers $m_1, m_2, \dots, m_n$ as $m_1 + 1, m_2 + 1, \dots, m_n + 1, 0$.

$$\lambda a.a\,\underline{m_1+1}\ \dots\ \underline{m_n+1}\,\underline{0}$$

We compute the length function f.

$$Y_{Turing}(\underbrace{\lambda f \lambda xy.(\underline{sg}\ y)(f(\underline{s}x))x}_{L})\underline{0}$$

So we take $C_*(Y_{Turing}L)$ to cause application to a list to put the list in function position:

$$C_*(Y_{Turing}L\underline{0})(\lambda a.a(\underline{m_1+1}\ \dots \underline{m_n+1}, \underline{0})) \to_\beta (\lambda a.a(\underline{m_1+1}\ \dots \underline{m_n+1}, \underline{0}))(Y_{Turing}L\underline{0})$$

9.1.1 Computing the Maximum of a List

Imagine that you are sent to the grocery store to get the best head of lettuce they have. We can apply a sequential algorithm to sorting the lettuce to find the best head. Pick up two heads of lettuce. Keep the better one. Pick up the next head in the sequence and compare again. Repeat until there is no more lettuce to check.

9.1.2 Testing our list length function

$$\begin{aligned} &(Y_{Turing}L\underline{0}\underline{m_1+1}) \\ &\qquad \dots \end{aligned}$$

9.2 Simultaneous Fixed Points

There are more kinds of fixed points, other than the ones found by Turing and Curry. We defined a simple fixed point as

$$Ff =_\beta f$$

We can define simultaneous fixed points as

$$\begin{aligned} f = Ffg = F(\langle f,g\rangle K)(\langle f,g\rangle K_*) \\ g = Gfg = G(\langle f,g\rangle K_*)(\langle f,g\rangle K_*) \end{aligned}$$

We can see that

$$\langle f,g\rangle = \lambda x \, \langle F(\langle f,g\rangle, K)(\langle f,g\rangle K_*), G(\langle f,g\rangle, K)(\langle f,g\rangle K_*)\rangle \langle f,g\rangle$$

So, we can set

$$f = \underbrace{Y_{Turing}(\lambda x \langle F(xK)(xK_*), G(xK)(xK_*)\rangle)}_{\alpha} K$$

and

$$g = Y_{Turing}(\lambda x \langle F(xK)(xK_*), G(xK)(xK_*)\rangle) K_*$$

We can apply

$$\lambda x \langle F(\langle f,g\rangle, K)(\langle f,g\rangle K_*), G(\langle f,g\rangle, K)(\langle f,g\rangle K_*) > \alpha = \langle F(\alpha K, \alpha K_*), G(\alpha K, \alpha K_*)\rangle \quad \begin{aligned} K &= f \\ K_* &= g \end{aligned}$$

Let $f = Ffg$ and $g = Gfg$. We will define $\Phi(g) = F(\Phi g)g$. If we take $\alpha = Y_{Turing}(\lambda xy.F(xy)y)$. What do we know about α? $\alpha = \Phi = \lambda y \, F(\Phi y, y)$.

What:

$$Y_{Turing}(\lambda y G(F(Y_{Turing}(\lambda uv.F(uv)v)y)y)y)$$

9.3 The SKI Combinator Calculus

9.3.1 Closing Terms with Bracket Abstraction

It seems like it should not be too difficult to take an open expression and convert it into one which is closed. One process which accomplishes this is called **bracket abstraction**. Let x, y be variables. We recursively define bracket abstraction as:

$$\begin{aligned} [x]y &:= Ky \\ [x]x &:= I \\ [x](E_1, E_2) &:= S([x]E_1)([x]E_2) \end{aligned}$$

We claim two things about the bracket abstraction: first, that if U is an applicative combination of S, K, I and free variables, then $([x]U)x \twoheadrightarrow_\beta U$, and second, that the bracket abstraction of an open term U is closed.

We can prove our first claim inductively. Let our bracket abstraction be $[x]U$. Our base case is where U is a free variable. There are then two possible subcases depending on whether $U = x$ or $U \neq x$. If $U = x$, then our bracket abstraction is $[x]x = I$, and our claim holds: $Ix \to_\beta x$. If

$U = y \neq x$, then we have $[x]y = Ky$, and we have $Kyx \rightarrow_\beta y$.

This leaves our inductive step, where $U = QV$. $[x]U$ is therefore $S([x]Q)([x]V)$. By our induction hypothesis, $[x]Qx \rightarrow_\beta Q$ and $[x]Vx \rightarrow_\beta V$. We can perform a β reduction to get

$$S([x]Q)([x]V)x \twoheadrightarrow_\beta [x]Qx([x]Vx) \twoheadrightarrow_\beta QV$$

Our second claim can also be proven inductively. We again use the bracket abstraction $[x]U$. Our base case is again that U is a free variable. If $U = x$, then we get $[x]x = I$, which is closed. If $U = y$, then we get $[x]y = Ky =_\beta \lambda x.y$, which is closed. In our inductive step, we assume by our induction hypothesis that $[x]Q$ and $[x]V$ are closed, and we apply

$$S([x]Q)([x]V) =_\beta \lambda xyz.([x]Q)([x]V) \rightarrow_\beta \lambda z.([x]Qz)([x]Vz)$$

This is a closed term as well. When taken together, we can see that the bracket abstraction is a transformation which allows us to close open terms without altering their meaning.

9.3.2 Church and Curry's Combinator Calculii

How many combinators do we really need? At this juncture, we have defined a moderately sizeable list of combinators to perform a variety of tasks. As it turns out, we can construct some of these combinators out of more simple ones. In fact, every closed term in the lambda calculus can be represented with nothing more than the combinators S, K and I. This claim is was made by Curry, and the resulting system is called the SKI combinator calculus. There is a weaker version of this theorem which was proven by Church that relies on the combinators $B, C, K, W,$ and I. We will prove Curry's version.

Recall: S, K, and I

$$\begin{array}{rcl} S & := & \lambda xyz.xz(yz) \\ K & := & \lambda xy.x \\ I & := & \lambda x.x \end{array}$$

9.3.3 Proof of Completeness for SKI

We prove that any lambda term can be written using only S, K and I by introducing a transformation $T[X] = Y$ which takes an arbitrary lambda term X and converts it into an applicative combination of S, K and I with free variables. $T[\,]$ is defined recursively as follows.

$$\begin{array}{rcl} T[x] & := & x \\ T[(XY)] & := & (T[X]T[Y]) \\ T[\lambda x.X] & := & (KT[X]) \text{ if } x \notin \text{FV}(X) \\ T[\lambda x.x] & := & I \\ T[\lambda xy.E] & := & T[\lambda x.T[\lambda y.E]] \text{ if } x \in \text{FV}(E) \\ T[\lambda x.(XY)] & := & (ST[\lambda x.X]T[\lambda x.Y]) \text{ if } x \in \text{FV}(X) \text{ or } x \in \text{FV}(Y) \end{array}$$

As should now be familiar, our proof will proceed by induction on the structure of a lambda term, U. In our base case, $U = x$ and $T[x] = x$. For our inductive step, we have our usual two cases. For our first case, consider $U = (XY)$. Since $T[(XY)] = (T[X]T[Y])$, and we know that $T[X]$ and $T[Y]$ are applicative combinations of S, K, I and free variables, the application $(T[X]T[Y])$ is also in SKI.

Finally, we have the case where $U = \lambda x.V$, which breaks down into several further subcases. If $V = x$, then we simply have the identity combinator, I. If $V = X$ and x is not in the free variables of X, then $T[\lambda x.X] = KT[X]$

9.3.4 Proof of Equivalence for $T[\,]$

9.4 Yet Smaller: The ι Calculus

If our objective is minimalism, we can do better than SKI combinator calculus by observing that there exists a combinator which can be used to construct the SKI combinators. This combinator was discovered by Chris Barker, and is called ι.

$$\iota x \quad := \quad \lambda x.xSK$$

Applying ι to itself gives us the identity combinator, I.

Deriving I from ι

$$\begin{aligned}
\iota\iota \quad &=_\beta \quad (\lambda x.xSK)(\lambda x.xSK) \to_\beta (SKSK) \\
&=_\beta \quad SK(\lambda xyz.xz(yz)K) \to_\beta SK(\lambda yz.Kzyz) \\
&=_\beta \quad S(\lambda xy.x)(\lambda yz.Kzyz) \to_\beta S(\lambda xy.x)(\lambda yz.z) \\
&=_\beta \quad SKK_* =_\beta \lambda z.Kz(K_*z) \to_\beta \lambda z.Kz \\
&=_\beta \quad \lambda z.(\lambda xy.x)z =_\beta \lambda z.z \\
&=_\beta \quad I
\end{aligned}$$

If we apply ι twice to I we get K.

Deriving K from ι and I

$$\iota\iota I \quad =_\beta \quad (\lambda x.xSK)(\iota I) \to_\beta (\iota I)$$

Lecture 10

10.1 A Review

10.1.1 Combinators

$$\begin{array}{rcl} B & := & \lambda xyz.x(yz) \\ C & := & \lambda xyz.xzy \\ K & := & \lambda xy.x \\ I & := & \lambda x.x \\ W & := & \lambda xy.xyy \\ S & := & \lambda xyz.xz(yz) \end{array}$$

10.1.2 Bracket Abstraction

$$[x]\left\{\begin{array}{l} \text{S} \\ \text{K} \\ \text{I} \\ \text{y} \end{array}\right. = \left\{\begin{array}{l} KS \\ KK \\ KI = K_* \\ Ky \end{array}\right.$$

$$[x](XY) = S([x]X)([x]Y), [x]x = I$$

$[x]X$ is defined for X an applicative combinations of S, K, I and variables.

10.2 Beta Conversion of Terms

Lemma: $([x]X)x \twoheadrightarrow_\beta x$.

Proof: by induction on the definition of bracket abstraction. The base case is X is one of S, K, I, x. So, $[x]x := Ix = x$, and we additionally have

$$\begin{array}{rcl} KSx & \rightarrow_\beta & S \\ KKx & \rightarrow_\beta & K \\ KIx & \rightarrow_\beta & I \end{array}$$

Remark: We could replace I by SKK.

$$SKKx \rightarrow_\beta (Kx)(Kx) \rightarrow_\beta x$$

Inductive step: $[x](UV)x$ is by definition $S([x]U)([x]V)x \to_\beta ([x]Ux)([x]Vx)$. By the induction hypothesis, $[x]Ux \twoheadrightarrow_\beta U$ and $[x]Vx \twoheadrightarrow_\beta V$. ■

We define for each term X an applicative combination of $S, K,$ and the free variables of X as X^{CL}, where the CL stands for Combinatory Logic. It might be better to call it combinatory algebra, but it retains the CL name for historical reasons. Dude who named them (Curry?) thought it could be a foundation for all of mathematics.

How do we formally define X^{CL}?

$$\begin{aligned} x^{CL} &= x \\ (XY)^{CL} &= (X^{CL}Y^{CL}) \\ (\lambda xX)^{CL} &= [x]X^{CL} \end{aligned}$$

Claim: $X^{CL} \twoheadrightarrow_\beta X$.

Proof: Induction on the definition of X^{CL}. Base case: clear. Induction step: Case 1: $X = (UV)$. Then $X^{CL} = (U^{CL}V^{CL})$, and by the induction hypothesis, $U^{CL} \twoheadrightarrow_\beta U$ and $V^{CL} \twoheadrightarrow_\beta V$. Case 2: $X = \lambda uU$. Then $X^{CL} = [u]U$. By induction on U, basis case: $U = V \neq u$. Then $[u]U = Kv \to_\beta \lambda u.v = \lambda u.U$. If $U = u$, then $[u]U = I = \lambda u.u = \lambda u.U$. Induction step: $U = Z_1Z_2$. $[u]U = [u](Z_1Z_2) = S([u]Z_1)([u]Z_2)$. We have now

$$S : \lambda xyu.(xu)(yu)$$

So $\lambda u([u]Z_1u)([u]Z_2u)$ which reduces by our lemma to $\twoheadrightarrow_\beta \lambda u(Z_1Z_2) = \lambda u(U)$.

Finally, $U = \lambda vV$. We will leave this case as an exercise. ■

Lecture 11

11.1 Useful Combinators

Booleans	$K := \lambda xg.x$	TRUE
	$K_* := \lambda xy.y$	FALSE
	ONE $:= \lambda abc.a$	
	TWO $:= \lambda abc.b$	
	THREE $:= \lambda abc.c$	

Equality function $$Eq\ \underline{n}\ \underline{m} = \begin{cases} K & \text{if } n = m \\ K_* & \text{if } n \neq m \end{cases}$$

Notice that we should be able to derive Eq if we have a function E such that

$$E\ \underline{n}\ \underline{m} = \begin{cases} \underline{1} & \text{if } n = m \\ \underline{0} & \text{if } n \neq m \end{cases}$$

We can define a function $E(x, y) := \overline{sg}((x \dot{-} y) + (y \dot{-} x))$, where $sg(0) = 0$ and $sg(n+1) = 1 \forall n \in N$ and $\overline{sg}$ is defined as $\overline{sg} = 1$ iff $sg = 0$ and $\overline{sg} = 0$ otherwise.

11.2 Stack Discipline

The empty stack is defined as KK_*, or THREE. If our stack is $\lambda a\ aLR$, L is the top, R is the result of popping the stack.

11.3 Trees

What will we do today?

(I) Describe the data structure

(II) Tasks:

- insertion of an element
- balancing the tree

- finding an element in the tree

(III) write an informal algorithm for finding an element in a tree

- write a lambda term which implements our algorithm

A tree is a data structure which is comprised of leaves and internal nodes, where a leaf is defined as an ordered triple $<$ TRUE$, \underline{n}, X >$ of a boolean which is true, an index number, and an object. An internal node is an ordered triple of $<$ FALSE$, L, R >$ of a false boolean, a left subtree, and a right subtree.

We provide a recursive definition of trees: $<$ TRUE$, \underline{n+1}, X >$ is a tree, and $<$ FALSE$, L, R >$ is a tree.

Now we describe our algorithm: $f(x, y, z)$ where x is a tree, z is the item we are searching for, and y is a stack of trees. If x is a leaf, then $x(TWO)$ is the index. We apply $Eq(x(TWO))z$. If this is $\underline{1}$, then we want to return x of 3, so we write:

$Y_{Turing}(x(ONE)(Eq(x(TWO))z(x(THREE))(f\,(\text{TOP}\ y)(\text{POP}\ y)z)(f(x(TWO))(\lambda a\,a(x(THREE))y))))$

We call this function as Fx(Empty stack$z$).

Lecture 12

12.1 Untitled on Board

$$((\lambda x X)Y) \to_\beta [Y/x]X$$

Step 1:

$$U_1 \to_\beta \Rightarrow U_1 V \to_\beta U_2 V \text{ and } VU_1 \to_\beta VU_2$$

Also,

$$\lambda u U_1 \to_\beta \lambda u U_2$$

Many steps:

$$U \to_\beta V \Rightarrow U \twoheadrightarrow_\beta V,\ U = V \Rightarrow U \to_\beta V.$$
$$U_1 \twoheadrightarrow_\beta U_2 \text{ and } V_1 \twoheadrightarrow_\beta V_2 \Rightarrow U_1 V_1 \twoheadrightarrow_\beta U_2 V_2$$

Additionally,

$$U \twoheadrightarrow_\beta V \Rightarrow \lambda u U \twoheadrightarrow_\beta \lambda u V$$

12.1.1 Reduction Sequence

We define a reduction sequence from U to V as a set of β reductions Δ_i such that

$$U = U_0 \overset{\Delta_0}{\to}_\beta U_1 \to_\beta \dots \overset{\Delta_{n-1}}{\to}_\beta U_n = V$$

$U =_\beta V$ is the smallest equivalence relation $\subseteq \twoheadrightarrow_\beta$. We say that $U =_\beta V \iff$ there exists a sequence of terms $U_0, U_1, \dots U_{2n}$ such that

$$\begin{tikzcd}
U_0 \arrow[dr] & & U_2 \arrow[dl] \arrow[dr] & & \dots \arrow[dl] \arrow[dr] & \\
& U_1 & & U_3 & & U_n = V
\end{tikzcd}$$

12.2 Church-Rosser Theorem

The Church-Rosser theorem states that if there exists a term $U =_\beta V$ then there exists a W such that $U \twoheadrightarrow_\beta W \twoheadleftarrow_\beta V$. We state it now without proof, but over the course of the next couple of lectures we will build up to a proof.

12.2.1 Strong Diamond Property

The proof of the Church-Rosser theorem establishes something called the ``diamond property.'' The idea is that if we can take a ``hill'' and turn it into a ``valley,'' then we can ``pave'' a sequence of beta reductions into a single valley. Formally, the diamond property states that: if

$$U \twoheadleftarrow_\beta X \twoheadrightarrow_\beta V$$

12.2.2 Notes on Diagrams

In a diagram, you see both solid lines and dotted lines. The solid lines are read ``if this line exists,'' and the dotted lines are read ``then this line exists.''

$$\begin{tikzcd} & X \arrow[dl] \arrow[dr] & \\ U \arrow[dr, dashed] & & V \arrow[dl, dashed] \\ & W & \end{tikzcd}$$

We can attempt to prove the strong diamond property (which is not true.) Imagine that we have $U \Delta_1 X \Delta_2 V$. Δ_1 and Δ_2 are disjoint. Notice that Δ_2 has a unique residual in U, and Δ_1 has a unique residual in V. We can then do:

$$U \overset{\Delta_1}{\leftarrow}_\beta D \overset{\Delta_2}{\rightarrow}_\beta V$$
$$U \overset{\Delta_2}{\rightarrow}_\beta D' \overset{\Delta_1}{\leftarrow}_\beta V$$

Notice that

$$[Y_1/x]([Y_2/z]X_2) = [Y_1/x, [Y_1/x]Y_2/z]X$$

Our final case is the degenerate case Δ_2 is in Y_1. We then have

$$\begin{tikzcd} & X \arrow[dl] \arrow[dr] & \\ U \arrow[rr, dashed, leftrightarrow] & & V \end{tikzcd}$$

So, strong diamond property does not work, but this is okay for the purposes of proving Church-Rosser. At this point in the proof, many people part ways. Church and Rosser have a proof, as do Kleene and J.J. Levy/Hindley.

Church and Rosser show that all of the redexes of Δ_2 are disjoint. So eventually we get to our original, nondegenerate case. We will take a different route.

12.2.3 Weak Diamond Property

The weak diamond property replaces the single beta reductions with beta reduction sequences:

$$\begin{tikzcd} & X \arrow[dl] \arrow[dr] & \\ U \arrow[dr, dashed] & & V \arrow[dl, dashed] \\ & W & \end{tikzcd}$$

Unfortunately, the weak diamond property is not enough on its own. We cannot prove Church-Rosser using weak-diamond. Explaining why uses surjective pairings and is beyond the scope of this course.

12.2.4 A Brief Combinatorial Fact: König's Lemma

The German mathematician König had a theorem, called König's Lemma: a finitely branching tree with no infinite path is finite. The contrapositive is that if we have an infinite tree, then we have an infinite path.

Proof: Suppose that T is an infinite, finitely branching tree. We find our infinite path in the following way. Consider all the subtrees of T: $T_1, \dots T_n$. By the pidgeonhole principle, at least one of these subtrees must be infinite. Take the edge to that subtree. This finds us an infinite path. ■

Lecture 13

13.1 Normalization

A question which arises now is how we determine whether two terms are equivalent, for some definition of equivalence. Clearly, for instance, the terms $\lambda x.X$ and $\lambda y.X$ share some common structure which the terms $\lambda xy.X$ and $\lambda z.Z$ do not. But what do we say about the terms $\lambda xy.X$ and X? Since $\lambda xy.X \to_\beta X$, it makes some sort of sense to say that $\lambda xy.X =_\beta X$ as well. But what about terms with more complicated structure? It is nice, for the purposes of comparison, if we can put terms into some sort of normal form.

We say that a term X is **normal** if it cannot be contracted by β reduction. This seems like a natural place to make comparisons from. We have contracted the term as far as we can contract it, and now it cannnot be contracted anymore. Recall that a term is contractable by β reduction if it has a β redex as a subterm. So we contract all of the β redexes to reach normal form. Obviously, this is not possible in all cases; simply consider trying to find a normal form for Ω!

X is said to be **normalizable** if there exists a term N in normal form such that $X \twoheadrightarrow_\beta N$. We will eventually prove that this implies some sort of uniqueness about the term X. X is said to be **strong normalizable** if every reduction sequence of X ultimately ends in a normal form term. We indicate that X is strongly normalizable by $X \in$ SN.

To understand why the distinction between normalizable and strong normalizable is necessary, consider the term $Kx\Omega$. There are two possible β reductions:

$$\begin{aligned} Kx\Omega &\to_\beta x \\ Kx\Omega &\to_\beta Kx(\Omega\Omega) \end{aligned}$$

Clearly, one of these reduction sequences terminates in the normal term x. The term $Kx(\Omega\Omega)$, however, is in the same situation as $Kx\Omega$. By reducing the K first, we can return to our normal form x. However areduction sequence which consists of the repeated reduction of Ω will never terminate in a normal form. It is worth noting as well that if a reduction sequence of $Kx\Omega$ ever does terminate, it terminates in the form x. You might recall that this is simply our weak diamond property.

$$\begin{tikzcd} & Kx\Omega \arrow[dl] \arrow[dr] & \\ x \arrow[dr, dashed] & & Kx\Omega\Omega \arrow[dl, dashed] \\ & x & \end{tikzcd}$$

13.1.1 Reduction Sequences of Strongly Normalizable Terms

We claim that if X is strongly normalizable, then the entire reduction tree of X is finite, and the tree ends in a unique normal form W.

$$\begin{tikzcd}
& X \arrow[dl, two heads] \arrow[dr, two heads] & \\
U \arrow[dr, dashed, two heads] & & V \arrow[dl, dashed, two heads] \\
& W &
\end{tikzcd}$$

We can prove the first claim by appealing to König's lemma. A finitely long term can only have a finite number of possible reductions. Since the term is strongly normalizable, we know that by definition every reduction sequence terminates in a normal form. That they terminate implies that there are no infinite paths.

Our second claim is more sophisticated. We can prove it by induction on the length of the reduction sequences of X. The trivial case is when X is already normal, and the length of all β reduction sequences of X are 0. In our inductive step, consider reduction sequences R_1 and R_2 from X to U and V:

$$R_1 : X \twoheadrightarrow_\beta U$$
$$R_2 : X \twoheadrightarrow_\beta V$$

If the lengths of R_1 or R_2 are not one (or zero), then we can look at the first reductions of R_1 and R_2. If one of R_1 or R_2 are zero, then one reduction could be the pseudo-β reduction of a term to itself.

$$\begin{tikzcd}
& & X \arrow[dl] \arrow[dr] & & \\
& X_1 \arrow[dl, two heads] \arrow[dr, dashed, two heads] & & X_2 \arrow[dl, dashed, two heads] \arrow[dr, two heads] & \\
U & & Y & & V
\end{tikzcd}$$

Observe that X_1 has a smaller reduction tree than X. We can then use the induction hypothesis to see that U and Y, and V and Y reduce to some terms U_1 and V_1:

$$\begin{tikzcd}
& & X \arrow[dl] \arrow[dr] & & \\
& X_1 \arrow[dl, two heads] \arrow[dr, dashed, two heads] & & X_2 \arrow[dl, dashed, two heads] \arrow[dr, two heads] & \\
U \arrow[dr, dashed, two heads] & & Y \arrow[dl, dashed, two heads] \arrow[dr, dashed, two heads] & & V \arrow[dl, dashed, two heads] \\
& U_1 & & V_1 &
\end{tikzcd}$$

We now have U_1 and V_1, which by the inductive hypothesis eventually reduce to some term W in a finite number of reductions. Additionally, that number of reductions is fewer than the number of reductions for X, so by the induction hypothesis we can claim that there exist reductions $U_1 \twoheadrightarrow_\beta W$ and $V_1 \twoheadrightarrow_\beta W$.

$$\begin{tikzcd}
& & X \arrow[dl] \arrow[dr] & & \\
& X_1 \arrow[dl] \arrow[dr, dashed] & & X_2 \arrow[dl, dashed] \arrow[dr] & \\
U \arrow[dr, dashed] & & Y \arrow[dl, dashed] \arrow[dr, dashed] & & V \arrow[dl, dashed] \\
& U_1 \arrow[dr, dashed] & & V_1 \arrow[dl, dashed] & \\
& & W & &
\end{tikzcd}$$

This demonstrates that all reduction sequences of X eventually terminate at some term W, which proves our claim. So all redution sequences of strongly normalizable terms are finite, and terminate in the same term.

13.2 Simulating Strong Normalization

Digression: ∞ reduction paths (or perpetual reduction paths). For some $X \notin$ SN, find a principle redex in X, called Δ such that the contraction of Δ results in a term $Y \notin$ SN.

Possible terms:

1. The principle redex of λuU is the principle redex of U.
2. $X = X_1, \ldots X_n$ has a principle redex of X_i such that i is the smallest such i such that a redex exists.
3. $X = (\lambda x X_0)X_1, \ldots, X_n$. If the head redex is contracted, then it is principle.

The principle redex redex is

a. if $x \notin FV(X_0)$, then the principle redex of X is the principle redex of X_1, if it exists.

b. else $(\lambda x X_0)X_1$ is the principle redex.

Corollary: If X is not strongly normalizable, and Y results by contracting the principle redex of X, then $Y \notin$ SN.

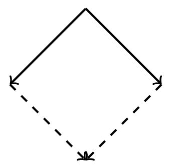
Definition: Comparing redexes in X. Standardizations says that you should always contract a redex able on to the left of the anchor.

13.2.1 On Coloring Redexes (A Simulation of SN)

Given a term X, we color some (strict) subset of the redexes in X green. An uncolored redex is red. If X reduces to Y in a single step by some green redex Δ, then the redexes in Y which are residuals of those in X which are green get colored green. We call this a **Colored Reduction**.

If $X \to_{\beta,colored} Y$, and Y has no colored redexes, then Y is said to be a complete development of green redexes in X.

Theorem (Hindley): Every colored reduction sequence terminates in a unique normal form. Note that we have weak diamond for colored reductions.



Proof: $X \mapsto \#X$ and a colored reduction to Y gives us $Y \mapsto \#Y$ for some $\#$ which associates a nonnegative integer with terms.

Lecture 14

14.1 Valuation on a Term X

During the previous lecture we mentioned a function which mapped terms to nonnegative integers. Now we formalize that idea as the valuation of a term, and give some restrictions on what forms a valuation can take. A valuation on a term is a map $\mathcal{V}$: subterms$(X) \rightarrow \mathbb{N}^+$ satisfying two conditions:

- The value $\mathcal{V}(x) > \sum_T$ to the right of $_x$ $\mathcal{V}(T)$ in a redex in R. So for $(\lambda uU)V$, $\mathcal{V}(u) > \mathcal{V}(U)$.
- $\mathcal{V}((UV)) >$ max$(\mathcal{V}(U), \mathcal{V}(V))$ and $\mathcal{V}(\lambda uU) > \mathcal{V}(U)$

Valuations exist for any term. Assume by recursion that

- All leaves to the right of x have values
- All subterms all of value variable occurance have values themselves have values. (What?)

Given a valuation of X,

$$\#X = \sum_{Y \text{ a subterm of } X} \mathcal{V}(Y)$$

14.1.1 Colored Reductions

Select some subset R of the set of redexes in a given term X and color them green. Color all others red. We defined last time colored reductions $X \rightarrow X_o \overset{\Delta_1}{\rightarrow_\beta} X_1 \overset{\Delta_2}{\rightarrow_\beta} \ldots$ where $\Delta_1 \in R$, and $\Delta_2 \in$ residuals of R.

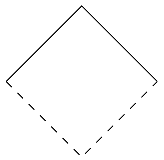
Claim: There exists a valuation on X_1 induced by the reduction $X_0 \overset{\Delta_1}{\rightarrow_\beta}$ such that $\#X_1 < \#X_0$.

Proof: Consider a redex $\Delta_1 \in R$...

Conclusion: Every colored reduction sequence terminates. This gives us that every colored reduction sequence terminates in a unique ``normal form'' with no colored redexes. We call these colored reductions ``complete development.''

14.1.2 The Lemma of Parallel Moves

This is also known as the Strip Lemma. If we have a reduction of the form



Remark: Church-Rosser follows by induction on the conversion. With many and one, we can repeatedly apply until we get many and many.

14.1.3 Proof of the Strip Lemma

Lecture 15

15.1 Review

The finiteness of developments results in a unique ``normal form.''

Strip Lemma (the Lemma of Parallel Moves): A single reduction Δ and a β reduction followed by a complete reduction of all the residuals of Δ can be reached by β reducing the term after applying Δ.

15.2 Cofinal Reduction Strategies

Assign to each term a reduction sequence

$$x = x_1 \to_\beta x_2 \to_\beta x_3 \to_\beta \ldots$$

So any Y such that $X =_\beta Y$ is β reduceable to some term in the sequence.

15.2.1 Klop's Theorem

Theorem (Jan Willem Klop): A reduction sequence $x = x_0 \to_\beta x_1 \to_\beta \ldots$ is cofinal if for any Δ redex in any x_i, eventually there is a $j > i$ such that Δ has no residual in x_j.

Jan Willem Klop

Klop was born in 1945 in Gorinchem, Netherlands. He works on the Algebra of Communicating Processes at Vrije Universiteit, in Amsterdam.

Correlary: The following sequence is cofinal:

$x_1 \twoheadrightarrow_\beta$ complete reduction of all redexes in x_1 $x_2 \twoheadrightarrow_\beta$ complete reduction of all redexes in x_2 $\cdots$

Proof: Suppose that the reduction sequence

$$x = x_0 \to_\beta x_1 \to_\beta \ldots$$

has the quality that for a redex Δ in x_i there exists some x_j, $j > i$ such that Δ has no residual in X_j. Therefore, there exists some $k > i$ such that there are no redexes in x_i have residuals in x_k. Now we prove that this reduction sequence is cofinite. Given $Y =_\beta x =_\beta x_0$, then by Church-Rosser, there exists a Z such that $X \twoheadrightarrow_\beta Z \twoheadleftarrow_\beta Y$. We will show that there exists an i such that $Z \twoheadrightarrow_\beta X_i$. We have that

$$Z \overset{\mathscr{R}}{\twoheadleftarrow}_\beta x_0 \to x_1 \to \ldots \to x_i \to \ldots$$

We induct on the length of the reduction sequence $\mathscr{R}$ from X to Z. Basis: take the one redex Δ in $\mathscr{R}$. Let $i = 0$. Then there exists $j > 0$ such that X_j has no residuals of Δ by the strip lemma.

We now consider for our induction step that $\mathscr{R}$ has a length greater than 1.

$$\ldots \leftarrow_\beta x_i \leftarrow_\beta \ldots \leftarrow_\beta x_1 \leftarrow_\beta x_0 \twoheadrightarrow_\beta U \to_\beta (\Delta)\ Z$$

Take the last reduction, Δ in $\mathscr{R}$. By the induction hypothesis, there exists an n such that $U \twoheadrightarrow X_n$. Let C be the set of residuals of Δ in X_n. There exists a $k > n$ such that no ersidual of C has X_k a residual in X_n.

15.3 A Method to Prove that $X \neq_\beta Y$

We will generalize colored reductions. The idea, due to J. J. Levy, is called Levy labels.

The general idea of Levy labels is that if we have a term X, we assign nonnegative integers to the subterms of X, representing the ``potential to reduce.'' Colored reduction simply assigned all ``red'' redexes 0, and ``green'' redexes 1, saying they had the potential for one reduction.

For our purposes, we will instead use finite sequences of integers. These sequences are nonempty, and the integers themselves are nonnegative. These sequences additionally obey the following property: $n_{i+i} = n_i \pm 1$.

$$n_1, n_2, n_3, \ldots n_k$$

- $n_i \geq 0$
- $n_{i+i} = n_i \pm 1$
- $k > 0$

If Y is a subterm of X, we will write $Y = Y^\alpha$, where $\alpha = n_1 \ldots n_k$. We require that all occurances of a given variable X should always have the same first element in the sequence $x = x^{n\alpha}$. The αs may vary. Applications look like:

$$(U^{\alpha,n+1}, V^{\beta,n})^{n,\gamma}$$

Abstractions look like:

$$(\lambda x^n X^{\alpha,n})^{n+1,\gamma}$$

Chapter 16

16.1 * The Book on Levy Labels

TODO: find a copy of the textbook.

16.2 Potential of a Term

The potential function maps from subterms (occurances) to sequences of nonnegative integers $\alpha = n_1 n_2 \dots n_k$ such that $n_{i+1} = n_i \pm 1$.

We write $Y = Y^\alpha$. The superscript is for the potential of Y. This satisfies

- (Uniformity) For every variable occurance of x the potential begins with the same integer $x^{n\alpha}$.
- (Construction)
  - Application: $(U^{\alpha,n+1} V^{\beta,n})^{n,\gamma}$
  - Abstraction: $(\lambda u^{n,} U^{\beta,n})^{n+1,\gamma}$

Example:

$$(\lambda x (x^3 x^{3,2})^{2,3})^4$$

Proposition: Every term has at least one pootential assignment where all potentials are $\geq$ any nonegative integer.

Proof: Let some integer N be nonnegative. We assign to each variable x the potential $N+1$. If we have $U = U^{\alpha,N+1}$ and $V = V^{\beta,N+1}$. Then

$$(U^{\alpha,N+1} V^{\beta,N+1,N})^{N,N+1}$$
$$(\lambda u^{N+1} U^{\alpha,N+1})^{N+2,N+1}$$

Beta reduction looks like:

$$((\lambda x^{\alpha,} X^{\alpha,n})^{n+1 Y^{\beta,n}})^{n,\gamma}$$

Most generally, we could have

$$((\lambda x^{n,} X^{\alpha,n})^{n+1,\delta,n+1} Y^{\beta,n})^{n,\gamma}$$

but we do not do this. We can simply do

$$((\lambda x^{\alpha,} X^{\alpha,n})^{n+1 Y^{\beta,n}})^{n,\gamma} \to [Y^{\beta,n} / x^{n,}] X^{\alpha,n,\gamma}$$

16.2.1 Potential Reduction

$$(U^{\alpha,n+2,n+1}V^{\beta,n})^{n,\gamma} \to (U^{\alpha,n+2}V^{\beta,n,n+1})^{n+1,n,\gamma}$$
$$(U^{\alpha,n+1,n+2}V^{\beta,n+1})^{n+1,\gamma} \to (U^{\alpha,n+1}V^{\beta,n+1,n})^{n,n+1,\gamma}$$

Example: let
$$\omega^4 = (\lambda x^3(x^3x^{3,2})^{2,3})^4.$$
Then if we want to reduce $\Omega = \omega\omega$, we do

$$\begin{aligned}\omega^4\omega^{4,3} &\to (\omega^{4,3}\omega^{4,3,2})^{2,3,4} \to \omega^4\omega^{4,3,2,3} \to \omega^{4,3,2,3}\omega^{4,3,2,3,2} \to \omega^{4,3,2}\omega^{4,3,2,3,2,1}\\ &\to \omega^4\omega^{4,3,2,3,2,1,2,3} \to \omega^{4,3,2,3,2,1,2,3}\omega^{4,3,2,3,2,1,2,3,2}\\ &\to \omega^{4,3,2,3,2,1}\omega^{4,3,2,3,2,1,2,3,2,1,0}\end{aligned}$$

We now have two kinds of reductions: beta reductions and potential reductions.

Definition: We say that a term with potential is strongly normalizable if every alternating sequence of beta reductions and potential reductions terminates.

Theorem: Every term is strongly normalizable.

Lecture 17

17.1 Reduction of Terms with Potential

Recal that a potential is a sequence of nonnegative integers $n_1, n_2, \ldots, n_k$ such that $n_{i+1} = n_i \pm 1$. We now have three kinds of reductions we can perform:

$$\begin{array}{rl}
\underline{\text{Beta:}} & (1)\ ((\lambda x^{n,} X^{\alpha,n})^{n+1} Y^{\beta,n})^{n,\gamma} \to_\beta [Y^{\beta,n}/x^{n,}]X^{\alpha,n,\gamma} \\
\underline{\text{Potential:}} & (2)\ (U^{\alpha,n+2,n+1} V^{\beta,n})^{n,\gamma} \to (U^{\alpha,n+2} V^{\beta,n,n+1})^{n+1,n,\gamma} \\
 & (3)\ (U^{\alpha,n+1,n+2} V^{\beta,n+1})^{n+1,\gamma} \to (U^{\alpha,n+1} V^{\beta,n+1,n})^{n,n+1,\gamma}
\end{array}$$

The above potential reductions are oftentimes referred to as ``rotations."

Recall that we introduced potentials as a generalization of the idea of colored reductions. This raises a question: what corresponds to the case of colored redexes? We define every variable x to have a potential of zero: x^0. Every instance of lambda abstraction gives us then $(\lambda x^0 X^0)^{1,0}$. Our reductions stall, as this gives us $U^{0,1}V^2$.

If we assign a variable a potential of 1, then our reductions do not stall. So this corresponds to a colored reduction. We have U^1, V^1.

17.1.1 Weak Diamond with Potential Reduction

We now consider a case where we have two terms U and V which both can be reduced using *either* beta reduction or potential reduction from some third term X. The weak diamond property also holds in this case.

$$\begin{tikzcd}
& X \arrow[dl, "\text{Beta/Potential}"'] \arrow[dr, "\text{Beta/Potential}"] & \\
U \arrow[dr, dashed] & & V \arrow[dl, dashed] \\
& Z &
\end{tikzcd}$$

Proof: Consider Δ_1 and Δ_2 as β redexes. There are four possible cases:

$$\begin{tikzcd}
& X \arrow[dl, "\Delta_1"'] \arrow[dr, "\Delta_2"] & \\
U \arrow[dr, dashed] & & V \arrow[dl, dashed] \\
& Z &
\end{tikzcd}$$

(1) Δ_1 and Δ_2 are disjoint. This case is equivalent to the one without potentials.

(2) Δ_1 is contained in Δ_2

 (a) Δ_1 is contained in argument
 (b) Δ_1 is contained in abstraction terms

(3) Rotation/Rotation with $\Delta_1 \subseteq \Delta_2$. The diagram can be completed as in the case without potential.

(4) Rotation/Beta. There can be no conflict between a rotation and a beta reduction.

 Definition: We say that X^α is strongly normalizable SN if every reduction sequence terminates.

17.2 Perpetual Strategy

We define a principle redex Δ such that if $X = X_0 \to X_1 \to X_2 \to \ldots \to X_n \to \infty$ sequence, then there is one with Δ as the first redex.

The principle redex of a term X is

(1) $X = (\lambda y^{n,} Y^{\beta,n})^{n+1,\alpha}$, then the principle redex is $Y^{\beta,n}$.

(2) $X = ((x^{\alpha,n+1} X_1^{\beta,n,})^{n_1} X_2^{\beta_2,N_1-1}) \ldots X_n^{\boxed{}}$, then

 (a) The first i such that X_i has a principle redex.
 (b) Else the principle redex is $x^{\alpha,n+1} X_1^{\beta,n}$ if (1) or (2) apply

Lecture 18

18.1 Lexicographic Orderings

A problem which sometimes arises is the problem of ordering tuples of the form $(x_0, \ldots, x_n)$ where the individual elements x_i have some ordering such that we can say $x_i = x_j, x_i < x_j$ or $x_i > x_j$. There are a number of ways that we could go about this, so we start by making a list of properties that we wish lexicographic ordering to have. Ideally, our ordering will tell us that identical lists are equal and anything else is not, and be transitive.

Alphabetic Ordering

The lexicographic ordering is a generalization of the idea of alphabetic ordering. Alphabetic ordering is just a lexicographic ordering where the tuple is the letters of the strings and the ordering is based on where in the alphabet letters occur.

An obvious first approach would be to simply say that for two tuples X and Y we will compare the first two elments x_0 and y_0. If $x_0 > y_0$, then $X > Y$, and so forth. We have to decide, however, what to do in a case where the length of X and the length of Y differ, as in the case where $X = (x_0, x_1, x_2)$ and $Y = (y_0, y_1)$. We can pad one of the tuples until it is equal in length to the other with elements which are either greater or smaller than the elements of the two lists. So if the elements of X and Y are integers, we might say that

$$(x_0, x_1, x_2) \overset{?}{=} (y_0, y_1) \iff (x_0, x_1, x_2) \overset{?}{=} (y_0, y_1, \infty)$$

18.2 Definition of Principle Redex

We claim that if an infinite reduction sequence exists for some term, we can find it by repeatedly contracting the principle redex of the term. The principle redex is defined in the following way:

Definition of Principle Redex

(1) $(\lambda x^{n}. X^{\alpha,n})^{n+1,\beta}$ is the principle redex of $X^{\alpha,n}$

(2) $(U^{\alpha,n+1} V^{\beta,n})^{n,\gamma}$ is the principle redex of

   (a) $U^{\alpha,n+1}$ if U is a variable or an application and the principle redex exists, else the principle redex of $V^{\beta,n}$

   (b) if $U^{\alpha,n+1} = (\lambda y^{m} Y^{\delta,m})^{m+1,\alpha',n+1}$ then

      (i) α' contains 0, then the principle redex of $Y^{\delta,m}$ if it exists, else

      (ii) $y \notin \mathrm{FV}(Y)$ then the principle redex of V if it exists, else Y.

At this point [(1)(a)(ii)] we have a term of the form:

$$((\lambda u^{n}. U^{\alpha,n})^{n+1,\beta,m+1} V^{\gamma,m})^{m,\delta}$$

(1) 0 in β

(2) no 0 in β, $u \notin \mathrm{FV}(U)$ principlal redex to be the one for V if it exists.

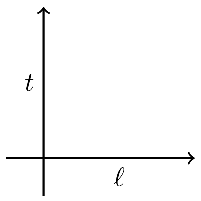
(3) Potential

18.2.1 Proof of Infinte Reduction of Principle Redex

We want to demonstrate that the perpetual reduction strategy always contracts the principle redex.

Lemma: if $X^\alpha \in$ SN and $Y^{\beta,n} \in$ SN where $y^n \in$ FV(X). It then makes sense to apply $[Y^{\beta,n}/y^n]X^\alpha$.

Proof: We first look at the special case of $Y^{\beta,n} = (xY_1 \dots Y_m)^{\beta,n}$. X^α is strongly normalizable, so the reduction tree of X^α is finite and of size t. X^α also has a length, ℓ. We can induct on pairs (t,ℓ).



We establish a lexicographic ordering on the points whereby $(t,\ell) > (t',\ell')$ if $t > t'$ or if $t = t'$ and $\ell > \ell'$.

If $X = y$, then $\alpha = n, \alpha'$. So when we perform the substitution, we get

$$[Y^{\alpha,n}/y^n]X^\alpha = (xY_1 \dots Y_m)^{\beta,n,\alpha'}$$

Note that $Y^{\beta,n} \in$ SN $\Leftrightarrow Y_1, \dots, Y_m \in$ SN.

Induction Step: We have the following cases:

- $X = zX_1 \dots X_k$ and $z \neq y$.
- $[Y/y]X = z([Y/y]X_1) \dots ([Y/y]X_{1_2})$ use ℓ.
- $X = (\lambda z X_0)X_1 \dots X_k$.
- $[Y/y]X = (\lambda z[Y/y]X_0)([Y/y]X_1) \dots ([Y/y]X_k)$.

We now again have several cases: if 0 is in the potential, then $z \notin$ FV(X_0) principle redex cases for X_1.

Lecture 19

19.1 TODO: Find notes on this lecture

Lecture 20

20.1 Closure of Strong Normalizable Terms Under Substitution

We would like to show that for any strongly normalizable terms X, Y, the substitution $[Y/y]X$ is also strongly normalizable.

20.1.1 Proof

We can prove the claim by induction on the tuple (p, r, t, ℓ), where

p	is the last integer in the potential of Y
r	is the size of the reduction tree of Y
t	is the size of the reduction tree of X
ℓ	is the size of X

Lexicographic Ordering

Remember our definition of a lexicographic ordering from before. It allows us to induct on the tuple (p, r, t, ℓ) by giving us a formal description of when the tuple is getting ``smaller.''

There are three options for the term X. It can either start with a variable z which is not being substituted, a lambda term, or a variable y which is being substituted.

$$\begin{aligned} X &= zX_1 \dots X_n \\ X &= (\lambda zZ)X_1 \dots X_n \\ X &= yX_1 \dots X_n \end{aligned}$$

For our third case, we have $[Y/y](X) = Y([Y/y]X_1)\dots([Y/y]X_n)$. These $n+1$ are in SN. We find the principle redex of Δ if Δ is a subterm of the first occurance of Y. We contract Δ in Y to give Y', so $\omega([Y/y]X_1)\dots([Y/y]X_n)$, which is in SN by induction hypothesis on r. Note that this operation might increase ℓ or t, but it definitely reduces r and does not affect p.

We have one more case: $Y = \lambda zZ$. We then have

$$x = (y^{p,\alpha,m} X_1^{\beta,m})^{m,\gamma} \dots Y \quad = \quad (\lambda z^s Z^{\delta,s})^{s+1,\mu,p}$$

let $[Y/y]X_1^{\beta,m}$ be $Z_1^{\beta,m}$. Then

$$(\lambda z^s Z^{\delta,s})^{s+1,\mu,p,\alpha,m+1} Z_1^{\beta,m}$$

We know that Z_1 is in SN by the induction hypothesis. Suppose $s+1 = p$. Then we are substituting something with a last potential $p-1$. Which would be nice, if it were true.

In case the potential $s+1, \mu, p, \alpha, m+1$ has a 0 in it, it is not a principle redex. Then we get strong normalization from the induction hypothesis. Similarly, if $z \notin \text{FV}(Z)$ we also get strong normalization.

Definition: Shift and Rotate are funtions from potentials without zeros to potentials. The point of these functions is that: $(U^{p,\alpha}V^{\beta})^{\gamma}$ potential reduces to $(U^{\beta}V^{\delta})^{\mu}$ where $\delta = \beta$ rotate(p,α) and $\mu =$ shift$(p,\alpha)\gamma$.

$$\begin{aligned}
\text{rotate}(p) &= \text{empty sequence}\\
\text{shift}(p) &= \text{empty sequence}\\
\text{rotate}(\alpha, q+2, q+1) &= q+1, \text{rotate}(\alpha, q+2)\\
\text{rotate}(\alpha, q+1, q+2) &= q, \text{rotate}(\alpha, q+1)\\
\text{shift}(\alpha, q+2, q+1) &= \text{shift}(\alpha, q+2)q+1\\
\text{shift}(\alpha, q+1, q+2) &= \text{shift}(\alpha, q+1)q
\end{aligned}$$

Fact: $(U^{p,\alpha}V^{\beta})^{\gamma}$ potential reduces to

$$(U^{p}V^{\beta,\text{rotate}(p,\alpha)})^{\text{shift}(p,\alpha),\gamma}$$

assuming 0 is not in p, α.

Back to the main thread of our proof, we have

$$((\lambda z^{s} Z^{\delta,s})^{s+1,\mu,p,\alpha,m+1} Z_1^{\beta,m})^{m,\gamma}$$
$$((\lambda z^{s} Z^{s})^{s+1,\mu,p} Z_1^{\beta,m,\text{rotate}(p,\alpha,m+1)})^{m,\gamma,\text{shift}(p,\alpha,m+1)}$$

Consider a new variable w of potential $p-1$,rotate$(s+1,\mu,p)$. Consider the term $[w^{p-1,\text{rotate}(s+1,\mu,p)}/z^{s}]Z^{s}$. We know that this is in SN by the first part of the lemma, our original special case, or by induction hypothesis on p. ■

Lecture 21

21.1 TODO: Find notes on this lecture

Lecture 22

22.1 TODO: Find notes on this lecture

Lecture 23

23.1 Plan

We have spent a while on theory, so now we spend some time on applications of fixed point combinators. Additionally, we have been talking a lot about β reduction, but not about η reduction, which we will now add. Furthermore, we will talk about Bohm's theorem, which answers the following question: For which sets of terms S do we have a conditional C i.e. if $X, Y \in S$ then $X = Y \Rightarrow CXY = K$ else if $X \neq Y \Rightarrow CXY = K_*$. Recall that $K = \lambda ab.a$ and $K_* = \lambda ab.b$. Put another way, this is the question of which sets of terms have a decidable equality.

Böhm's Theorem: If S is a set of normal forms, then there is a conditional C for equality being beta-eta converstion.

23.2 Demonstrating that Two Terms are Not Equal

Curry and Turing's Fixed Point Combinators

$$\begin{aligned} Y_C &= \lambda y.(\lambda xy(xx))(\lambda xy(xx)) \\ Y_T &= (\lambda x.(\lambda yy(xxy)))(\lambda x.(\lambda yy(xyy))) \end{aligned}$$

Can we show that $Y_C \neq Y_T$? The strategy for this is to pick a cofinal reduction sequence for one of the two terms. We will arbitrarily pick Y_C. If we have such a cofinal reduction sequence, and Y_T reduces to some point in the cofinal reduction sequence of Y_C.

$$\begin{tikzcd} Y_C \arrow[d, dashed, two heads] & Y_T \arrow[dl, dashed, two heads] \\ {} & \end{tikzcd}$$

We see that we have a cofinal reduction sequence for Y_C:

$$\lambda(\lambda xy(xx))\underbrace{(\lambda xy(xx))}_{\alpha} \to \lambda yy(\alpha\alpha) \to \lambda yy(y(\alpha\alpha)) \twoheadrightarrow \ldots \twoheadrightarrow \lambda y\, \underbrace{y(\ldots y}_{n}(\alpha\alpha))\ldots) \twoheadrightarrow \ldots$$

By Klop's theorem, this sequence is cofinal. Also, notice that there is only redex to contract, so this reduction sequence is unique. We also can write a cofinal reduction sequence for Y_T:

$$\underbrace{\lambda x(\lambda yy(xxy))}_{\beta}\,\lambda x(\lambda y(xxy)) \to \lambda yy(\beta\beta y) \twoheadrightarrow \lambda y(y(y(\beta\beta y))) \to \ldots \to \lambda y\, \underbrace{y(y(\ldots * y}_{k}(\beta\beta y))\ldots)$$

We could alternately write from the point $\lambda y.y(y(\beta\beta y))$:

$$\lambda y.y(y(\beta\beta y)) \twoheadrightarrow \lambda y\, \underbrace{y(y(\ldots y}_{k}(\lambda zz(\beta\beta z))y))\ldots)$$

Notice that both reduction sequences continue with internal reductions. We can see that the internal reduction sequences are of terms which look like $y(\beta\beta y)$ and $(\lambda zz(\beta\beta z))y$. Neither of those can look like $(y(\alpha\alpha))$, so the terms Y_C and Y_T cannot be equal.

23.2.1 Generalization by Corrado Böhm

We define

$$Y_B = \underbrace{(\lambda x.(\lambda zyy(xxzy)))}_{\alpha}(\lambda x\lambda zy.y(xxzy))$$

by adding an extra parameter to Y_C. Recall that

$$\underline{n} := \lambda xy.\underbrace{x(\ldots(x}_{n}\, y)\ldots)$$

We then define

$$Y_n = Y_B^{\underline{n}} \to (\lambda zy.y(\alpha\alpha zy))\underline{n} \to \lambda y.y(\alpha\alpha\underline{n}y) \twoheadrightarrow \lambda y.y(y(\alpha\alpha\underline{n}y)) \twoheadrightarrow \ldots$$

We can now observe that if $Y_n \neq_\beta Y_m$ for some $n \neq m$, we have infinitely many fixed point combinators.

Remark: F is a fixed point combinator if $\Longleftrightarrow$ $Fx = x(Fx)$.

Exercise: Show that $Fx =_\beta x(Fx)$ is equivalent to $F =_\beta (\lambda y\lambda x(yx))F$. Start by abstracting both sides with a λ. Note: $(\lambda y\lambda x(yx))$ is called the owl combinator, so named by Ray Smullyan.

23.3 Eta reduction

Remember that if $x \notin$FV(X), then $\lambda x(Xx) \to_\eta X$. This is eta reduction. We also can define eta expansion:

$$X \to \lambda x(Xx) \text{ for } x \notin \text{FV}(X)$$

(1) Every eta reduction sequence terminates because lengths decrease

(2) Eta reduction has the diamond property.

$$\begin{tikzcd}
& \circ \arrow[dl, "\Delta_1"'] \arrow[dr, "\Delta_2"] & \\
{} \arrow[dr] & & {} \arrow[dl] \\
& \circ &
\end{tikzcd}$$

Proof of (2): Consider how Δ_1 and Δ_2 overlap. There can be no conflict.

(3) Eta expansion has strong diamond

(3) Eta expansion has Church-Rosser

23.3.1 Eta expansion of terms with potential

$$U^{\alpha,n+1,n+2,\beta} \xrightarrow{\text{eta expansion}} (\lambda u^{n+1}(U^{\alpha,n+1}u^{n+1,n})^{n,n+1})^{n+2,\beta}$$

Lecture 24

24.1 Diamond Property for Reductions

24.1.1 Beta Reduction

24.1.2 Eta Expansion

$$\begin{aligned} U^{\alpha,n+1,n+2,\beta} &\rightarrow (\lambda u^{n+1,n}(U^{\alpha,n+1}u^{n+1,n})^{n,n+1})^{n+2} \\ U^{\alpha,n+2,n+1,\beta} &\rightarrow (\lambda u^{n,n+1}(U^{\alpha,n+2}u^{n,n+1})^{n+1,n})^{n+1} \end{aligned}$$

24.1.3 Potential Reductions

$$\begin{aligned} (U^{\alpha,n+1,n+2}V^{\beta,n+1})^{n+1,\gamma} &\rightarrow (U^{\alpha,n+1}V^{\beta,n+1,n})^{n,n+1,\gamma} \\ (U^{\alpha,n+2,n+1}V^{\beta,n})^{n,\gamma} &\rightarrow (U^{\alpha,n+2}V^{\beta,n,n+1})^{n+1,n,\gamma} \\ (\lambda u^{n,}U^{\alpha,n})^{n+1,n+2,\gamma} &\rightarrow (\lambda u^{n+1,n}U^{\alpha,n,n+1})^{n+2,\gamma} \\ (\lambda u^{n+1}U^{\alpha,n+1})^{n+2,n+1,\gamma} &\rightarrow (\lambda u^{n,n+1}U^{\alpha,n+1,n})^{n+1} \end{aligned}$$

To prove the diamond property for reductions, any reduction tree without potential can be given potential.

(1) Every β reduction sequence terminates. (add clockwise rotation)

(2) Every eta expansion sequence terminates.

(3) Every potential reduction sequence terminates.

24.2 Tools for Proving Termination

24.2.1 Multiset Ordering

A multiset is a set with multiplicity. Vaguely, a multiset is a finite set of natural numbers where each integer can occur more than once. The simplest example would be the set $\{n, n\}$, where the integer $n \in \mathbb{Z}$. A multiset is an assignment of nonnegative integers to the nonnegative integers, i.e. a function $f : \mathbb{N} \mapsto \mathbb{N}$ such that there exists $m \in \mathbb{N}$ such that for all $n \geq m$, $f(m) = 0$. A third definition: A multiset is a finite list of nonnegative integers modulo order.

We can order multisets as follows: $X > Y$ if there is an element $x \in X$ and a multiset Z such that for each $z \in Z$, $z < x$ and $Y = X \setminus \{x\}$. That is to say, we say $X = \{\ldots, x, \ldots\}$ and $Z = \{z_1, \ldots, z_k\}$ for some $k \in \mathbb{N}$ and then do $X = \{\ldots, z_1, \ldots, z_k \ldots\}$.

Theorem: There is no infinite sequence of multisets $X_0 > X_1 > X_2 > \ldots > X_k > \ldots$. Recall König's Lemma: A tree which is finitely branching and infinite has an infinite path.

Suppose that we have an infinite descending sequence X_n of multisets. A finitely branching tree of nonnegative integers. At any given stage the current $X_i =$ the leaves of this tree. Note that this tree has infinite nodes. However by König's lemma, this finitely branching tree must have an infinite path. This is a contradiction.

Lecture 25

25.1 Localization

There are two types of local rotations, one for beta and one for eta.

(1) (Eta) Given an eta expansion $U \to_\eta \lambda u(Uu)$ followed by any sequence of rotations, then the rotaions can be done first on the term containing just U. Except for local clockwise rotations. Those are of the following form, for example:

$$\begin{aligned}(\lambda u^{n,}(U^{\alpha,m+1,m+2}u^{n,\beta,m+1})^{\square})^{\diamond}\\ ((U^{\alpha,m+1}u^{n,\beta,m+1,m})^{m,\square})^{\diamond}\end{aligned}$$

Local means a clockwise rotation of the particular application $U \to_\eta \lambda u(Uu)$.

(2) (beta) Given beta reduction $(\lambda uU)V \ \to_\beta \ [V/u]U.$ preceded by a seqnece of rotations. Then all the rotations can be done last, except local clockwise rotations of the redex.

$$\begin{aligned}(\lambda u^{n,}U^{\alpha,n})^{n+1,n+2}V^{\delta,n+1}\\ \to (\lambda u^{n+1,n}U^{\alpha,n,n+1})^{n+1}V^{\delta,n+1}\\ = [V^{\delta,m+1}/u^{n+1,n}]U^{\alpha,n,n+1}\\ \dots\\ \to (\lambda u^{n,}U^{\alpha,n})^{n+1}V^{\delta,n+1,n}\\ = [V^{\delta,n+1,n}/u^{n,}]U^{\alpha,n}\end{aligned}$$

25.1.1 Eta Postponement

If we have an eta expansion followed by a local clockwise rotation followed by a beta reduction, then these can be done in the ``opposite'' order: a local clockwise reduction followed by a beta reduction followed by a sequence of eta expansions.

$$(\lambda uU)V \to [V/u]U$$

Say that we have an eta expansion inside a term, $U \to \lambda uUu$, followed by some number of rotations local to the eta, followed by a beta reduction $(\lambda yY)z \to [Z/y]Y$.

Case 1: Δ_1, Δ_2 are disjoint.

Case 2: Overlap: 2(a) $\Delta_1 \subseteq \Delta_2$ if $\Delta_1 = \Delta_2$ then we are clearly done in opposite order. If $\Delta_1 \subset \Delta_2$ and $\Delta_1 \subseteq Y$ $\Delta_1 \subseteq Z$ maybe many eta expansions in $[Z/y]Y$. $\Delta_1 = \lambda yY$ i.e. $UZ, \lambda u(Uu)ZmUZ$. Then simply keep UZ.

2(b) $\Delta_2 \subseteq \Delta_1$. There are several subcases: either $\Delta_2 = Uu$ or $\Delta_2 \subseteq U$ (This is another easy case). Now if $\Delta_2 = Uu$, then we get

$$\begin{aligned} &\lambda y Y \\ \lambda u((\lambda yY)u)\ &\text{clockwise rotations of } \lambda yY \text{ potentials onto } u. \\ \lambda u([u/y]Y)\ &\text{net effect is a counterclockwise rotation} \end{aligned}$$

Proof of Termination: We need to show that:

(1) Every alternating sequence of beta reduction and clockwise rotations terminates.

(2) Every sequence of rotations terminates.

(3) Every sequence of eta expansions and clockwise rotations terminates.

$$U^{\alpha,n+2,n+1,\beta} \to (\lambda u^{n,}(U^{\alpha,n+2}u^{n,n+1})^{n+1,n})^{n+1,\beta}$$

We assign for each subterm the sum of the values of the potential. The value of the term equals the multiset of those sums.

Suppose we have an infinite sequence of betas, etas, and rotations. Take the first beta, and permute it to the beginning. You now have a number of local rotations, followed by a beta. We repeat this process until we have a sequence of local rotations, followed by a sequence of betas, followed by etas and rotations. This terminates, obviously. For the full, formal proof, see the appendix.

Lecture 26

26.1 Head Normal Form

Recall: The general form of a term:

$$\lambda x_1 \ldots x_n . x_1 X_1 \ldots X_m$$

with $n \geq 0, m \geq 0$ and $i \in \mathbb{N}$. This is called head normal form. The head variable is x_i.

$$\lambda x_1 \ldots x_n (\underbrace{(\lambda x X_0) X_1}_{\text{head redex}} \ldots X_m)$$

with $n \geq 0$ and $m \geq 1$.

A term X is said to have a head normal form Y if $X =_\beta Y$ and Y is in head normal form. This is highly nonunique.

Suppose that Y is a head normal form, and $Y \twoheadrightarrow_\beta Z$. THen we know that Z is also a normal form. It has the same lambda prefix (n) and the same head variable (i), and the same number of components m. Therefore if $X =_\beta Y$, Y is in head normal form by church rosser.

$$X \twoheadrightarrow_\beta Z \twoheadleftarrow Y$$

Therefore, $X \twoheadrightarrow_\beta$ to a head normal form.

But $X \twoheadrightarrow_\beta Z$, where Z is in head noraml form. In the head reduction part, $\lambda y_1, \ldots, y_r((\lambda y Y_0)Y_1 \ldots Y_s)$. $X \twoheadrightarrow_\beta W \twoheadrightarrow_\beta Z$.

Theorem: X has a head normal form if and only if the head reduction sequence beginning with X terminates.

26.2 Solvability

Definition: X (a closed term) is said to be solvable if there exists terms $N_1 \ldots N_n$ such that $X N_1 \ldots N_n =_\beta I$.

$$\begin{aligned} X N_1 \ldots N_n K &= K \\ X N_1 \ldots N_n &= K \\ X N_1 \ldots N_n I I &= I \end{aligned}$$

Theorem (Wadsworth): A closed term M is solvable if and only if M has a head normal form.

Proof: M has a head normal form $M \twoheadrightarrow_\beta \lambda x_1 \dots x_n.x_i X_1 \dots X_m$. Then

$$M \underbrace{K_*^m \dots K_*^m}_{m \text{ times}} \twoheadrightarrow_\beta (\lambda x_1 \dots x_n(x_1 X_1 \dots X_m))K_*^m \dots K_*^m \twoheadrightarrow_\beta K_*^m([K_*^m/x_1, \dots, K_*^m/x_n]) \dots \twoheadrightarrow_\beta I$$

Conversely, suppose that M is solvable. $MN_1 \dots N_k \twoheadrightarrow_{\text{head reduction } \beta} I$. We cannot reduce any subterms in such a way that M is not in the head position.

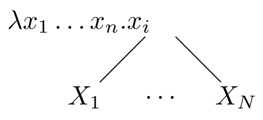
Remark: M is solvable if and only if there exists a term F such that $FM = K_*$ and for some other term N, $FN =_\beta K$.

Proof: M is solvable just like before. $F = \lambda u \dots uK_*^m \dots K_*^m FM \twoheadrightarrow_\beta K_*$. $FM = K_*$ and $FN = K$. We need two new variables x, y so that $FMxy =_\beta Y$. y is normal, so there is a head reduction $FMxy \twoheadrightarrow_\beta y$. $MW_1 \dots W_k \twoheadrightarrow_\beta y$.

Why does this prove that M has a head normal form? Just put in I for x and I for y and you get $M([I/x, I/y]W_1) \dots ([I/x, I/y]W_k) \twoheadrightarrow_\beta I$.

26.2.1 Properties of Unsolvable Terms

Consider the following procedure on the possibly open term X. Perform the head reduction sequence. Either it does not terminate, in which case we have an open term with no head normal form. We call this $\perp$, or bottom. The option is that we get $\lambda x_1 \dots x_n.x_i$ We break x_i into $X_1 \dots X_m$ and repeat in parallel, creating a finitely branching tree.



This is called the Bohm tree of the term.

26.3 Bohm Trees

Facts and examples:

(1) if X has a noraml form, then the Bohm tree of X is finite.

(2) any unsolvable term has a finite Bohm tree $= \perp$. For example, take $Y_T K$.

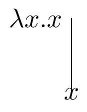
Example: computing the Bohm tree of $Y_T K$.

$$Y_T K \twoheadrightarrow_\beta K(Y_T K) \rightarrow_\beta \lambda x(Y_T K) \twoheadrightarrow_\beta \lambda x_1 \dots x_n(Y_T k)$$

We can see that this cannot have a head normal form, since we can have an arbitrarily large number of lambda prefixes. The Bohm Tree $BT(Y_T K) = \perp$.

Remark: $Y_T K$ is unsolvable, but it is not like Ω. Ω cannot be reduced even to something beginning with a lambda. $\Omega \not\twoheadrightarrow_\beta \lambda x X$. Ω is still $\perp$.

A good Bohm tree, for example, looks like this: $M := \lambda x.Y_T(\lambda y.xy) \twoheadrightarrow_\beta \lambda x.x(Y_T/\lambda y.xy)$. Repeating this many times gives you $\twoheadrightarrow_\beta \lambda x.x(\dots(x(Y_T(\lambda y.xy))))$. So the Bohm tree $BT(M)$ is:



Lecture 27

27.1 Böhm Trees

Definition: A Böhm tree is a finitely branching tree whose nodes are labeled

(1) With a λ prefix and a head variable

$$\lambda x_1 \dots x_n . x_i \;\; i \in \mathbb{N}$$

(2) If the tree is a leaf, then

$$\perp$$

We associate to each term X a Böhm tree. If X is unsolvable (that is, it has no head-normal form - by Wadsworth's Theorem), then the Böhm tree of X is simply $\perp$. However if X has a head normal form $\lambda x_1 \dots x_n . x_i X_1 \dots X_m$, then the Böhm tree of X should be:

$$\begin{tikzcd}
 & \lambda x_1 \dots x_n . x_i \arrow[dl, no head] \arrow[d, no head] \arrow[dr, no head] & \\
BT(X_1) & \dots & BT(X_m)
\end{tikzcd}$$

Claim: If $T =$ BT(X), then T has finitely many free variables.

Barendreight's theorem: If a finitely bounded Böhm tree has finitely many free variables then it is the Böhm tree of a term. TODO: FIGURE OUT WHAT THIS IS CALLED AND FILL IT OUT.

27.1.1 A Useful Ordering of Böhm Trees

We say that $T_1 \sqsubseteq T_2$ if you can take T_1 and replace (possibly infintely many) $\perp$ nodes with arbitrary other Böhm trees.

Remark: $\sqsubseteq$ is a partial ordering. It is reflexive: $T_1 \sqsubseteq T_1$. It is transitive.

Remark: T is finite.

Definition: T_1 and T_2 are **compatible** if there exists $T_1 \sqsubseteq T_3 \sqsupseteq T_2$.

Suppose that we have closed terms M, N and a closed term F such that $FM =_\beta K$ and $FN =_\beta K_*$. Then there exists a term with finite Böhm tree M_1 and a term N_1 with finite Böhm tree such that $BT(M_1) \sqsubseteq BT(M)$. We say that M_1 is an **approximation** of M. The same is true of N: $BT(N_1) \sqsubseteq BT(N)$.

Stated another way: IF there exists a term F that can differentiate between M and N, then it only needs to use a finite amount of the Böhm trees of M and N to do so. This idea is called

continuity. This in particular is a weak form of continuity. Theoretically, we can put anything in place of K and K_*, but if they have different Böhm trees, the theorem gets really hard to prove.

We have $FM =_\beta K$ and $FN =_\beta K_*$, so

$$FMxy \overset{\text{head reduction}}{\twoheadrightarrow} x$$
$$FNxy \overset{\text{head reduction}}{\twoheadrightarrow} y$$

Definition: The levels of a Böhm tree. The root of a Böhm tree is level 1. Every subBöhm tree is of level 2.

Proof: Suppose that $FM =_\beta K$. THen $FMxy =_\beta x$. So $FMxy \twoheadrightarrow_\beta x$. Then there is an assignment of potentials $(FMxy)^\alpha \twoheadrightarrow_\beta x^{\alpha'}$. This is a normal form. By our theorem of Strong Normalization and uniqueness of normal form, we know that any reduction sequence of terms with potential beginning with $(FMxy)^\alpha$ must terminate in the term $x^{\alpha'}$. We can pick the reduction sequence which first puts M in normal form as a term with potential. We call this new term M'.

$$M' = \lambda x_1 \ldots x_n. \begin{cases} x_i X_1 \ldots X_m \\ \text{or} \\ (\lambda x X)^0 X_1 \ldots X_m \end{cases}$$

If we have our first case case, the redex must be projected. In our second case we have a term with potential 0 so, we simply replace it with $\perp$, as not it nor its residual can be constracted.

So M_1 is the result of replacing terms of the second kind with bottom. The Böhm tree of M_1 is therefore clearly less than or equal to the Böhm tree of M. $BT(M_1) \sqsubseteq BT(M)$.

It is then sufficient to consider terms with finite Böhm trees. ■

Next time we will codify properties of unsolvable terms.

Lecture 28

28.1 Continuity

Recall: Suppose M is closed and $XM =_\beta x$, a free variable. Then there exists a term N with finite Böhm tree such that $BT(N) \sqsubseteq BT(M)$, and $XN =_\beta x$.

Claim: Suppose that $XN =_\beta x$ and the Böhm tree of N is finite. Then whenever we have any term M such that $BT(M) \sqsupseteq BT(N)$, then $XM =_\beta x$.

Proof: The Böhm tree of N is obtained by repeatedly applying the standardization theorem until leaves of the tree are obtained. Once those leaves have been obtained, certain unsolvable terms are replaced by $\perp$. If we skip the final step, we have a reduct of N, say $N*$. We have $XN* \twoheadrightarrow_\beta x$. We can then replace those unsolvable leaves with $\Omega = (\lambda x.xx)(\lambda x.xx)$. The unsolvable terms are usually in the form $\lambda x_1 \ldots x_n(\lambda x X)X_1 \ldots X_m$. We now have a new term N^+ such that $XN^+ \twoheadrightarrow_\beta x$.

Clearly, then, if $BT(M) \sqsupseteq BT(N)$, then $XM \twoheadrightarrow_\beta x$.

Corollary: If $FM_1 =_\beta K$, and $FM_2 =_\beta K_*$, then M_1 and M_2 have incompatible Böhm trees. That is to say, there is no term L such that $BT(M_1) \sqsubseteq BT(L) \sqsupseteq BT(M_2)$.

Proof: $FM_1xy =_\beta x$ and $X = \lambda u(Tuxy)XM_1 =_\beta x$. $FM_2xy =_\beta y$ $XM_2 =_\beta x$. Apply claim since we have $BT(L) \sqsupseteq BT(M_1) \sqsupseteq BT(M_2)$. $BT(M) \sqsupseteq BT(N)$

$XN_1* \twoheadrightarrow_\beta x \quad BT(L) \sqsupseteq BT(N_1*)$
$XN_2* \twoheadrightarrow_\beta y \quad BT(L) \sqsupseteq BT(N_2*)$
Then there exists a finite term $L*$ such that $BT(L*) \sqsupseteq BT(N_1*)$ and $BT(L*) \sqsupseteq BT(N_2*)$. So we have that $XL* \twoheadrightarrow_\beta x$ and $XL* \twoheadrightarrow_\beta y$. This violates Church-Rosser.

Remark: Converse is false. For example, and 2 terms which eta convert cannot be ``split'' by such an F ie if $M =_\beta N$ then you cannot have an F such that $FM =_\beta K$ and $FN =_\beta K_*$. So, for example $\lambda xx =_\eta \lambda xy(xy)$. The Böhm tree of $\lambda x.x$ is $\lambda x.x$ and the Böhm tree of $\lambda(xy).(xy)$ is $\lambda xy.x - - - y$.

Eta is the only counterexample. Two finite Böhm trees are separable (splittable) only if one is not an eta conversion of the other.

28.2 Properties of Unsolvables

We say that a possibly open term X is unsolvable if X has no head normal form.

Remark: If X is solvable then not necessarily the case that $M_1 \dots M_m X M_1 \dots M_m =_\beta I$. What is true is that there is a substitution $\Theta = [N_1/x_1, \dots, N_n/x_n]$ where $FV(X) \subseteq \{X_1 \dots X_n\}$.

Recall that for all terms are either, $\lambda x_1 \dots x_n.x_i X_1 \dots X_m$, $n \geq 0, m \geq 0, i \in \mathbb{N}$ (this is the head normal form) or $\lambda x_1 \dots x_n (\lambda x X) X_1 \dots X_m$ with $n \geq 0, m \geq 0$, which is not in head normal form.

(1) Unsolvables are enumerable under reduction.

(2) If X is unsolvable, then so is ΘX for any Θ

Proof: If the head reduction sequence of $X = X_1 \to_\beta X_2 \to_\beta \dots \to_\beta X_k \to_\beta \dots$ then the head reduction sequence of ΘX is $\Theta X_1 \to_\beta \Theta X_2 \to_\beta \dots \to_\beta \Theta X_k \to_\beta \dots$

(1) 3 If and only if X is unsolvable, then so is $\lambda x.X$.

(2) 4 If X is unsolvable, and Y is any term, then XY is unsolvable.

Proof: Let X be unsolvable, and Y a term. Consider a reductino sequence $XY \to_\beta X_0 \to_\beta X_1 \to_\beta X_2 \to_\beta \dots \to_\beta X_k \to_\beta \dots$. There are zero or more head reductions of X.

Case 1: X does not head reduce to something beginning with a λ. Then this sequence is an infinite sequence.

Case 2: Consider ther first time a λ comes to the head of X. We get $X = X_0$ and $X_0 Y \to_\beta X_1 Y \to_\beta \dots \to_\beta X_k Y \to_\beta (\lambda u U)Y \to_\beta [Y/u]U \to_\beta \dots$. By property 3, $\lambda u U$ is unsolvable, so U is unsolvable. By condition 2, $[Y/u]U$ is unsolvable, therefore $[Y/u]U$ from above is unsolvable and the following reduction is its head reduction sequence.

28.2.1 Congruence

Definition: Congruence: We define an -equivalence- congruence relation such that one subset of the equivalence relation is all unsolvables. We additionally want this equivalence relation to be closed under beta conversion. So $X \subseteq B, Y =_\beta X \Rightarrow Y \in B$. We partition into blocks closed modulo beta conversion: $X, Y \in B, U, V \in B \twoheadrightarrow XU, YV \in B$.

Lecture 29

29.1 Congruence

Definition: A binary relation $\sim$ on closed terms is said to be a **congruence** if

(1) $\sim$ is an equivalence relation

(2) closed under β (and η). i.e. if $M =_\beta N \Rightarrow M \sim N$.

(3) $M \sim N$ and $P \sim Q \Rightarrow MP \sim NQ$

Examples:

(i) for all M, N $M \sim N$

(ii) $\{N : M \sim N\} = \{N : M =_\beta N\}$

(iii) $\{N : M \sim N\} = \{N : M =_{\beta\eta} N\}$

We know that example iii is not example i because of the Church-Rosser theorem: distinct normal forms don't $\beta\eta$ convert. iii and ii are different because β conversion is not as exansive as $\beta\eta$ conversion.

Example: $\lambda xy.xy =_\eta \lambda x.x$ are in the same class for iii but have unique beta forms, so they are not in the same class for ii. The separation of i and ii is trivial. Take any two terms which do not β convert.

Suppose that S is a set of closed terms which are closed under β conversion. Then there is a congruence relation $\sim$ such that if $M, N \in S$ then $M \sim N$. Indeed there is a smallest such $\sim$. i.e. for any other congruence $\backsim$, we have $M \sim N \Rightarrow M \backsim N$.Why? Because the intersection of congruences is a congruence.

29.2 Jacopini's Theorem

When are M and N related by the smallest congruence containing S as a class?

29.2.1 Jacopini Tableaus

A **tableau** relating M and N consists of a sequence of terms $M_1, M_2, \dots M_m$ and pairs of terms $(P_1, Q_1), (P_2, Q_2), \dots, (P_m, Q_m)$ and the β conversions

$$\begin{aligned} M &=_\beta M_1P_1Q_1 \\ M_1P_1Q_1 &=_\beta M_2P_2Q_2 \\ M_2P_2Q_2 &=_\beta M_3P_3Q_3 \\ &\vdots \\ M_{m-1}P_{m-1}Q_{m-1} &=_\beta M_mP_mQ_m \\ M_mP_mQ_m &=_\beta N \end{aligned}$$

where $P_i, Q_i \in S$ such that $i = 1, \dots, m$ and $m \geq 1$. An example is $S = \{Q : Q =_\beta P\}$.

Let J be defined by $J(M, N)$ if and only if there is a Jacopini tableau linking M and N. We now claim that J is a congruence, and indeed is the smallest congruence.

Theorem:

(1) J is a congruence

(2) If $\sim$ is a congruence containing S as a class $J(M, N)$, then $M \sim N$

Corellary: Let $\sim$ be the smallest congruence containing S = the set of unsolvable terms as a class. Then if $M \sim N$ then $BT(M) = BT(N)$.

Example: Ω and $\Omega\Omega$ are both in S, but are β different. So $\Omega \sim \Omega\Omega$. They have the same Böhm tree. In fact, $\lambda x.x\Omega \sim \lambda x.x(\Omega\Omega)$, and they have the same Böhm tree.

Proof: $M \sim N \Leftrightarrow$ there is a Jacopini tableau. Compute the Böhm tree $TB(M_i, P_i, Q_i)$. Compute a head normal form, if it exists.

Remark: Jacopini's Theorem: We can set S to be the β closure of $\{\Omega, M\}$, then the smallest $\sim$ containing S as a class is not our first example from above.

Lecture 30

30.1 More on Jacopini

For some S a nonempty set of closed terms closed under β equivalence, recall that $J(M,N) \Leftrightarrow$ there exist $M_1,\dots,M_m$ which are closed, $P_1,\dots P_m \in S$, $Q_1 \dots Q_m \in S$ such that

$$\begin{aligned} M &=_\beta M_1P_1Q_1 \\ M_1P_1Q_1 &=_\beta M_2Q_2P_2 \\ &\vdots \\ M_mP_mQ_m &=_\beta N \end{aligned}$$

NOTICE THAT THE Ps AND Qs SWITCH PLACES FROM ONE SIDE TO ANOTHER. Last time we introduced the Jacopini relation $J(M,N)$, which is a congruence which is closed under $=_\beta$. Today we show that it is the smallest.

Suppose that we are given $\sim$, a congruence such that if $P,Q \in S$, then $P \sim Q$. We want to show that if $J(M,N)$, then $M \sim N$.

Consider a case $M_iP_iQ_i = M_{i+1}P_{i+1}Q_{i+1}$. We have $J(M, M_1P_1Q_1)$ and $M \sim M_1P_1Q_1$. Why? $\sim$ is a congruence class, and the congruence classes are closed under β conversion. Similarly, $J(M_iQ_iP_i, M_{i+1}Q_{i+1}P_{i+1})$ implies that $M_iQ_iP_i \sim M_{i+1}Q_{i+1}P_{i+1}$.

So we have some $M_iP_iQ_i$, now $M_iP_i \sim M_iP_i$, because congruence relations are reflexive. Secondly, we have $Q_i, P_{i+1} \in S$ by definition of tableau. Therefore $Q_i \sim P_{i+1}$. Therefore $M_iP_iQ_i \sim M_iP_iP_{i+1}$.

30.1.1 Let us start over...

We want to show that $M_iP_iQ_i \sim M_iQ_iP_i =_\beta M_iP_iQ_i \sim M_iP_iP_i$ since $Q_iP_i \in S$. We know that

$$\lambda x(M_ixQ_i)P_i \dots$$

30.1.2 Let us start over...

We want to show that $M_iP_iQ_i \sim M_iQ_iP_i =_\beta M_iP_iQ_i \sim M_iP_iP_i$. We know that

$$M_iP_iQ_i \sim M_iP_iP_i =_\beta (\lambda x M_ixP_i) \sim (\lambda x M_ixP_i)P_iQ_i \in S$$

We are trying to prove that every step in the tableau equivalences remain in S.

We have a problem. If $S = \{K, K_*\}$. If we have $K \sim K_* \to_\beta M \sim N \Rightarrow KMN \sim K_*MN \to_\beta M \sim N$.

30.2 Böhm's Theorem

We will prove a bare bones version: If M and N are closed terms in normal form and they do not η convert, then there exists terms $P_1 \ldots P_n$ such that $MP_1 \ldots P_m =_\beta K$ and $NP_1 \ldots P_m =_\beta K_*$.

Remark: This is true for terms with finite Böhm trees.

30.2.1 Proof

The terms M and N look like:

$$\lambda x_1 \ldots x_n . x_i$$

where x_i has subterms $X_1 \ldots X_m$. n and m can both be zero. There are three different ways in which a node in M can differ from one in N. Different length lambda prefix, different length bound variables, ???.

Eta Expansion Example: $\lambda xy.xX \to_\eta \lambda xy.(\lambda z(xX)z)$ We could alternately reduce $\lambda xy.xX \to_\eta \lambda x \lambda z(\lambda y.xX)z$.

Notice that when we eta convert, the length of the lambda prefix and the number of bound variables both change. But they both change in such a way that $N - M$ is invariant.

We assume that M, N are $M \neq_\eta N$ normal form with the same tree. Find the first place at which the Böhm trees are different. For example, suppose that at the root of the tree we have $M = \lambda x_1 \ldots x_n . x_i X_1 \ldots X_n$ and $N = \lambda y_1 \ldots y_k . y_j Y_1 \ldots Y_k$. There are two cases.

Case 1: $n \neq k$, $n > k$ For each $P_1 \ldots P_n$, $MP_1 \ldots P_n = P_i X_1^+ \ldots X_n^+$ and $NP_1 \ldots P_n = P_j Y_1^+ \ldots Y_m^+ P_{k+1} \ldots P_n$. How do we ensure that $P_i = P_j$? We set for all $r = 1, \ldots, n$ $P_r = \lambda z_1 \ldots z_n I$. So $MP_1 \ldots P_n =_\beta I$

Lecture 31

31.1 Proof of Böhm's Theorem

Recall: M,N in β-normal form. $M \neq_\beta N$.

Claim: There exists $P_1 \dots P_\ell$ such that

$$\begin{aligned} MP_1 \dots P_\ell K \\ NP_1 \dots P_\ell K_* \end{aligned}$$

Remark: By η expansions we can assume that the Böhm trees of M and N are the same ``abstract'' tree. That is, the trees are the same without labels.

Proof: The proof is by induction on the depth of the first place where these two Böhm trees actually differ.

The basis case is when the two Böhm trees differ at the root, i.e. $M = \lambda x_1 \dots x_m . x_i X_1 \dots X_t$ and $\lambda y_1 \dots Y_n . y_j Y_1 \dots Y_t$.

There are two cases.

(1) $n \neq m$ so we assume without loss of generality that $m > n$

We define

$$K_*^k := \lambda z_1 \dots z_k I$$

so that $K_*^1 = K_*$. We then write for $r \geq t$

$$M \underbrace{K_*^r \dots K_*^r}_{m} =_\beta K_*^r X_1^+ \dots X_t^+ =_\beta K_*^{r-t}$$

And

$$N \underbrace{K_*^r \dots K_*^r}_{m} =_\beta K_*^r Y_1^+ \dots Y_t^+ K_*^r \dots K_*^r \to_\beta K_*^{r-t} \underbrace{K^r \dots K^r}_{m-n} =_\beta K_*^{r-t+m-n}$$

Pick r such that these two terms reduce to terms of the form K_*^α and K_*^β where $\alpha \neq \beta$.

Assume $\alpha > \beta$. It suffices to find $Q_1 \dots Q_s$ such that $K_*^\alpha Q_1 \dots Q_s = K$ and $K_*^\beta Q_1 \dots Q_s = K_*$. Explicitly, these are $\lambda x_1 \dots x_\alpha . I Q_1 \dots Q_s$.

Induction step: Let

$$\begin{aligned} M &= \lambda x_1 \dots x_n . x_i X_1 \dots X_t \\ N &= \lambda x_1 \dots x_m . x_i Y_1 \dots Y_t \end{aligned}$$

Let r be the smallest $1 \leq r \leq t$ such that the first place at which they differ is in the pair $X_r Y_r$. Consider $x_i X_1 \ldots X_t$ and $x_i Y_1 \ldots Y_t$. Pick an integer ℓ which is very large, i.e. $\ell >> t + t$. Substitute $\lambda y_1 \ldots y_\ell \lambda a.a y_1 \ldots y_\ell$ for x_1.

Lecture 32

32.1 Gödel Numbering (without numbers)

The idea is to assign to each lambda term M at least one object, a Gödel ``number'' (or code) which is another term $\ulcorner M \urcorner$ (which is not necessarily unique) such that there exists a lambda term E, the evaluator, where

$$E \ulcorner M \urcorner =_\beta M$$

Note that this gives us a fixed point theorem (oftentimes called the Kleene fixed point theorem). For each M there exists an N such that $N = M \ulcorner N \urcorner$.

Corollary: There exists N such that $MN =_\beta N$.

Since we can apply Kleene's fixed point to the term $\lambda x.M(Ex)$, we get an N such that

$$MN =_\beta M(E \ulcorner N \urcorner) =_\beta (\lambda x.M(Ex)) \ulcorner N \urcorner = N$$

32.1.1 Proof of Kleene's Fixed Point

Given M, consider the term

$$P = \lambda x.M(Exx)$$

Then

$$P \ulcorner P \urcorner =_\beta (\lambda x.M(Exx)) \ulcorner P \urcorner =_\beta M(E \ulcorner P \urcorner \ulcorner P \urcorner) = M(P \ulcorner P \urcorner)$$

32.1.2 Codes

A code is a quadruple $\langle M, N, P, Q \rangle = \lambda a.aMNPQ$. There are three possible cases. Variable, application, or abstraction. M tells us which we are in. It is one of the following:

$$M = \begin{array}{ll} \lambda abc.a & \text{variable} \\ \lambda abc.b & \text{application} \\ \lambda abc.c & \text{abstraction} \end{array}$$

N is a comment. We do not care what it is at present. P is a placeholder which may be used later.

Q is a church numeral $\underline{n}$ which tells us which variable it is. So, for example,

$$\begin{aligned} \langle \lambda abc.a, N, P, \underline{n} \rangle &= \ulcorner X_n \urcorner \\ \langle \lambda abc.b, N, \ulcorner X \urcorner, \ulcorner Y \urcorner \rangle &= \ulcorner (X, Y) \urcorner \\ \langle \lambda abc.c, N, \underline{n}, \ulcorner X \urcorner \rangle = \ulcorner \lambda x_n . X \urcorner & \end{aligned}$$

This defines the set of codes of terms.

Remark 1: Many functions on codes of terms will be defined by recursion. For example, we define the depth function.

Recall: there is a term MAX such that MAX $\underline{n}\,\underline{m} = \underline{\text{max}(n,m)}$. We define depth$\ulcorner M \urcorner$ = a code of M such that the comments are the depth of the corresponding subterm. So

$$\text{depth}(M) = \begin{array}{cl} 1 & \text{if } M \text{ is a variable} \\ 1 + \text{max}(\text{depth}X, \text{depth}Y) & \text{if } M = XY \\ 1 + \text{depth}(X) & M = \lambda x.X \end{array}$$

32.1.3 A Pidgeon Lambda Calculus Language

We write $\ulcorner M \urcorner = \langle P_0, P_1, P_2, P_3 \rangle$. If $P_0 = \lambda abc.a$, then $D\ulcorner M \urcorner = \langle P_0, 1, P_2, P_3 \rangle$. If $P_0 = \lambda abc.b$, then $\langle P_0, \underline{+}\ \underline{1}$ MAX$(D\ulcorner P_2 \urcorner(\lambda xyzw.y))(D\ulcorner P_3 \urcorner(\lambda xyzw.y)), DP_2, DP_3\rangle$. IF $P_0 = \lambda abc.c$, then $D\ulcorner M \urcorner = \langle P_0, (DP_2(\lambda abced.b) + 1), P_2, P_3 \rangle$.

To make this a lambda term, we need to do two things. Replace the conditionals by projections, and then solve for D using a fixed point combinator.

Exercise 1

33.1 Problem Statement

We define a **pseudoterm** in the following manner:

- A variable x is a pseudoterm.
- If X is a pseudoterm and x is a variable, then $(\lambda x X)$ is a pseudoterm.
- If $X_1, \dots X_n$ are pseudoterms and x is a variable, then $(x\ X_1 \dots X_n)$ is a pseudoterm.
- If $X_2, \dots X_n$ are pseudoterms and $(\lambda x X)$ is a pseudoterm then $((\lambda x X) X_1 \dots X_n)$ is a pseudoterm.

Show that every pseudoterm results from a term by deleting parens around applications in function position. Note that this is slightly different from the formulation in class (deleting parens around applications not in argument position) because we are not using Church's dot notation. With dot notation, for example $(\lambda x(x X_1 \dots X_n))$ becomes $(\lambda x.x X_1 \dots X_n)$.

33.2 Proof

We consider the four cases of a pseudoterm enumerated in the problem statement. We will prove that each form of a pseudoterm can be constructed by removing parens around applications in function position from a term.

Case 1: x
We have a pseudoterm x which we wish to construct by removing parens from a term. We take the term x, and remove no parens. This gives us the pseudoterm x. $\square$

Case 2: $(\lambda x X)$
Again, we start with a term $(\lambda x X)$ and remove no parens to get the pseudoterm $(\lambda x X)$. $\square$

Case 3: $(x X_1 \dots X_n)$
Start with the case where $n = 1$: the pseudoterm $(x X_1)$. We can derive this directly from the term $(x X_1)$ by removing no parenthesis. Now consider that we have some strategy for deriving a pseudoterm $(x X_1 \dots X_n)$ from some term $(x \overline{X})$. We can then derive a pseudoterm $(x X_1 \dots X_n X_{n+1})$ from a term $((x\overline{X}) X_{n+1})$ by removing the parenthesis of the term in function position, $(x\overline{X})$. $\square$

Case 4: $((\lambda x X) X_1 \dots X_n)$
We again start with the case where $n = 1$. This is the pseudoterm $((\lambda x X) X_1)$. This is also a term, so we are done. Now we assume that we have some way to derive a pseudoterm $((\lambda x X) X_1 \dots X_n)$ from some term $((\lambda x X)\overline{X})$. We can then derive the pseudoterm $((\lambda x X) X_1 \dots X_n X_{n+1})$ from the term $(((\lambda x X)\overline{X}) X_{n+1})$ by removing the parenthesis around $((\lambda x X)\overline{X})$, which is in function position. $\blacksquare$

Exercise 2

34.1 Problem Statement

A elementary alpha conversion is an alpha conversion which changes only the variable at one occurrence of lambda, and, of course, at all the occurrences it binds. So, for example, the alpha conversion from $\lambda xy.(xy)$ to $\lambda yx.(yx)$ is not elementary but it can achieved by the sequence $\lambda xy.(xy)$, $\lambda xz.(xz)$, $\lambda yz.(yz)$, $\lambda yx.(yx)$ of elementary alpha conversions. Show that every alpha conversion from X to Y can be achieved by a sequence of elementary alpha conversions which uses at most one variable neither in X nor in Y.

34.2 Proof

We first consider the simple case when X and Y each contain a single variable, which is different. In this case, it is clear that a single elementary alpha conversion, changing that one differing variable, converts X to Y or vise versa.

Now consider X and Y such that

Exercise 3

35.1 Problem Statement

Let X be an applicative combination of S, K and variables. Show that $[x]X$ beta reduces to λxX. Conclude that for any term Y, Y^{CL} beta reduces to Y.

35.2 Proof

Exercise 4

36.1 Problem Statement

(i) Suppose that we have colored some of the redexes in X green and we have a colored reduction $X \twoheadrightarrow_\beta Y$. Let R be a green redex in X; show that no residual of R is a subterm of another residual of R (we say they are not "nested") in Y.[Hint; use the existence of a valuation on X; the value of any free variable of R must be less than the value of R]

(ii) Show that in a colored reduction all the alpha conversions can be done at the beginning. Is this true for all reductions?

36.2 Proof

Exercise 5

37.1 Problem Statement

Recall that we can reduce potentials in two ways:

(i) $(U^{a,n+1,n+2}V^{b,n+1})^{n+1,c}$ reduces to $(U^{a,n+1}V^{b,n+1,n})^{n,n+1,c}$

(ii) $(U^{a,n+2,n+1}V^{b,n})^{n,c}$ reduces to $(U^{a,n+2}V^{b,n,n+1})^{n+1,n,c}$

Show that every sequence of such reductions must terminate. [Hint; given a term X with potential, ignore lambdas to think of X as a binary tree with sequences of non-negative integers at its nodes. Let b be a positive integer bigger than the sum of all the previously mentioned integers $+3$. For a given subterm Y of X we have the potential $n_1, ..., n_k$ and the number of moves m to the left in the path from the root of X to Y. So for example in $(\lambda x.xx)(\lambda x.xx)$, where we have omitted potentials, the m for the 1st occurrence of x is 2, for the second occurrence of x is 1, for the third is 1, and for the 4th is 0. Now to this subterm of X assign the value $(n_1 + ... + n_k) * b^m$ and let $\#X$ be the sum of the values of the subterms of X. Show that potential reductions reduce $\#X$]

37.2 Proof

The Mumbles Proof

The following is copied pretty much verbatim from Professor Statman's writings on the subject. Basically, all I did was LaTeX it and make it look pretty.

Here is a proof that every reduction sequence of Levy labeled terms terminates based on Barendregt's perpetual reduction strategy.

I have called the proof the "Mumbles proof" because the proof originated at a pub in Mumbles, Wales in September of 1974 (Swansea mini-conference). I was there with Diederick vanDaalen, Roel deVrijer, and Jean-Jaques Levy, and as I remember it the idea of the proof is mostly mine (others might remember this differently).

Proposition: If the perpetual reduction strategy terminates on X then X is strongly normalizable.

Proof: Suppose that the perpetual reduction strategy terminates on X. Let p be the number of steps in the reduction strategy applied to X before a normal form results, and let q be the length of X. The proof is by induction on (p, q) ordered lexicographically. We distinguish two cases:

Case 1: X has no head redex or a head redex of rank 0. This case follows from the induction hypothesis on the second coordinate (or possibly the first coordinate).

Case 2: X has a head redex of positive Levy rank r. Let

$$X = \lambda x_1 \ldots x_s.(\lambda z Z)X_1 \ldots X_t$$

We distinguish two subcases:

Subcase 1: z does not occur free in Z. We suppose that there is an infinite reduction beginning with X. If no residual of the head redex is ever contracted in this reduction then either there is an infinite reduction sequence beginning with X_1 or there is an infinite reduction sequence beginning with

$$\lambda x_1 \ldots x_s.ZX_2 \ldots X_t$$

Either of these alternatives contradicts the induction hypothesis applied to p. If some residual of the head redex is contracted then there is an infinite reduction sequence beginning with

$$\lambda x_1 \ldots x_s.ZX_2 \ldots X_t$$

which contradicts the induction hypothesis on p.

Subcase 2: z occurs free in Z. Now

$$X' = \lambda x_1 ... x_s.([X_1/z]Z)X_2 ... X_t$$

is strongly normalizable by induction hypothesis on the 1st coordinate therefore each X_i is strongly normalizable. If an infinite reduction sequence begins with X then it either

(i) never contracts the unique residual of the head redex $(\lambda z Z)X_1$ or

(ii) contracts this redex at some stage.

The first case is impossible by the previous remark that the X_i are strongly normalizable. In the second case we have

$$\begin{aligned} \lambda z Z &\twoheadrightarrow_\beta \lambda z Z' \\ X_i &\twoheadrightarrow_\beta X'_i \text{ for } i = 1, \ldots, t \end{aligned}$$

and there is an infinite reduction sequence beginning with

$$\lambda x_1 \ldots x_s.([X'_1/z]Z')X'_2 \ldots X'_t$$

but this is impossible since X' is strongly normalizable and

$$X' \twoheadrightarrow_\beta \lambda x_1 \ldots x_s.([X'_1/z]Z')X'_2 \ldots X'_t. \ \blacksquare$$

Proposition: If the perpetual reduction strategy terminates on X and Y then it terminates on $[Y/x]X$.

Proof: If the perpetual reduction strategy terminates on X and Y then X and Y are strongly normalizable. Let X have reduction tree of size n and Y a reduction tree of size m. Moreover, let the Levy label of Y be l, and the length of X be o. The proof is by induction on the 4-tuple (l, m, n, o) ordered lexicographically. We distinguish several cases:

<u>Case 1:</u> X has a head redex with positive Levy rank. Here

$$X = \lambda x_1 \ldots x_r.(\lambda z Z)X_1 \ldots X_s,$$

and we set

$$X' = \lambda x_1 ... x_r.([X_1/z]Z)X_2 ... X_s.$$

In case z is free in Z the first step of the perpetual strategy on $[Y/x]X$ is

$$[Y/x]X \rightarrow_\beta [Y/x]X'$$

and the triple for $[Y/x]X'$ is $(l, m, k, -)$ where $k < n$. Thus the induction hypothesis applies and the perpetual reduction strategy terminates on $[Y/x]X'$. Thus it terminates on $[Y/x]X$. In case z is not free in Z.

<u>Case 2:</u> X has a head redex with Levy rank 0 or with a head variable different from x. This case follows easily from the induction hypothesis on the 4th coordinate.

<u>Case 3:</u> x is the head variable of X. Here

$$X = \lambda x_1 \ldots x_r.x X_1 \ldots X_s$$

where we assume xX_1 has Levy label k.

<u>Subcase 1:</u> Y begins with lambda with positive Levy label. Now the case where $s = 0$ is trivial. Set $X'_i = [Y/x]X_i$.Let x' be a new variable with Levy label k By the induction hypothesis on the 4th co-ordinate

$$\lambda x_1 \ldots x_r.x' X'_2 \ldots X'_s X'_1$$

are strongly normalizable. Let $Y = (\lambda z Z)^l$.By induction hypothesis on the first coordinate $[X_1^{l-1}/z]Z^{l-1}$ is strongly normalizable. In case z occurs in Z the perpetual strategy terminates by induction hypothesis on the first coordinate for $[[X_1^{l-1}/z]Z^{l-1}/x']\lambda x_1 \ldots x_r.x' X'_2 \ldots X'_s$, and in case z does not occur we must add that it terminates for X'_1.

Subcase 2: Y begins with a head redex with positive rank. Set $X'_i = [Y/x]X_i$. Let x' be a new variable with Levy label k. By the induction hypothesis on the 4th co-ordinate

$$X' = \lambda x_1 \dots x_r . x' X'_1 \dots X'_s$$

is strongly normalizable. Now the next step in the perpetual strategy on $[Y/x']X'$ is the next step on the head occurrence of Y in

$$\lambda x_1 \dots x_r . Y X'_1 \dots X'_s .$$

Let the result on Y be Y'. Then the perpetual strategy on $[Y'/x']X'$ terminates by the induction hypothesis on the second coordinate (or possibly the 1st coordinate if the Levy label of Y' drops from Y).

Subcase 3: Y begins with a head variable or a head redex of Levy rank 0, or a lambda with Levy label 0. This subcase follows from the induction hypothesis on the 4th coordinate. ■

Eta Expansion Proof

Here we outline the steps in the proof that every alternating sequence of beta reductions, eta expansions and potential rotations terminates. First we list the rules:

(beta)	$((\lambda y^{n,} Y^{A,n})^{n+1} Z^{B,n})^{n,C}$	$\rightarrow$	$[Z^{A,n}/y] Y^{B,n,C}$
(eta)	$Y^{A,n+1,n+2,B}$	$\rightarrow$	$(\lambda y^{n+1,} Y^{A,n+1} y^{n+1,n})^{n+2,B}$
(eta)	$Y^{A,n+2,n+1,B}$	$\rightarrow$	$(\lambda y^{n,n+1} Y^{A,n+2} y^{n,n+1})^{n+1,B}$
(clockwise rotation)	$(Y^{B,n+1,n+2} Z^{C,n+1})^{n+1,A}$	$\rightarrow$	$(Y^{B,n+1} Z^{C,n+1,n})^{n,n+1,A}$
(clockwise)	$(Y^{B,n+2,n+1} Z^{C,n})^{n,A}$	$\rightarrow$	$(Y^{B,n+2} Z^{C,n,n+1})^{n+1,n,A}$
(counter-clockwise rotation)	$(\lambda y^{n+1,} Y^{B,n+1})^{n+2,n+1,C}$	$\rightarrow$	$(\lambda y^{n,n+1,} Y^{B,n+1,n})^{n+1,C}$
(counter-clockwise)	$(\lambda y^{n,} Y^{B,n})^{n+1,n+2,C}$	$\rightarrow$	$(\lambda y^{n+1,n,} Y^{B,n,n+1})^{n+2,C}$

The first two steps involve the very important notion of localization.

(1) Localization w.r.t. eta:
Any eta expansion $U \rightarrow \lambda u.Uu$ followed by a sequence of rotations can be done in the opposite order; a sequence of rotations followed by the eta expansion followed by local clockwise rotations.

$$(\lambda u^{A,n+1})(U^{B,n+1,n+2} u^{A,n+1})^{n+1,C})^{D} \rightarrow (\lambda u^{A,n+1})(U^{B,n+1} u^{A,n+1,n})^{n,n+1,C})^{D}$$
$$(\lambda u^{A,n})(U^{B,n+2,n+1} u^{A,n})^{n,C})^{D} \rightarrow (\lambda u^{A,n})(U^{B,n+2} u^{A,n,n+1})^{n+1,n,C})^{D}$$

(2) Localization w.r.t. beta:
Any sequence of rotations followed by a beta reduction $(\lambda u U)V \rightarrow [V/u]U$ can be done in the opposite order; a sequence of local clockwise rotations followed by the beta reduction followed by a sequence of rotations.

$$((\lambda u^{n,} U^{A,n})^{n+1,B,m+1,m+2} V^{C,m+1})^{m+1,D} \rightarrow ((\lambda u^{n,} U^{A,n})^{n+1,B,m+1} V^{C,m+1,m})^{m,m+1,D}$$
$$((\lambda u^{n,} U^{A,n})^{n+1,B,m+2,m+1} V^{C,m})^{m,m+1,D} \rightarrow ((\lambda u^{n,} U^{A,n})^{n+1,B,m+1} V^{C,m+1,m})^{m,m+1,D}$$

The next step involves the permutation of beta and eta. It is like the similar step in our textbook except we are using eta expansion, not reduction.

(3) Eta postponement:
An eta expansion followed by a clockwise rotation local to a beta redex which is then contracted can be done in the opposite order; a local clockwise rotation followed by a beta reduction followed by a sequence of eta expansions. There may be no beta reduction and in this case there is a rotation.

We already know the following:

(4) Every alternating sequence of beta reductions and clockwise rotations eventually terminates.

Next,

(5) Every alternating sequence of eta expansions and local clockwise rotations eventually terminates.

Consider an eta expansion followed by a local clockwise rotation. For each subterm, sum the integers in its potential and take the multiset of results; one for each subterm. What happens to the multiset after the expansion followed by the rotation?

We already know the following for clockwise rotations alone.

(6) Every sequence of rotations eventually terminates.

We can define the multiset ordering for pairs of nonnegative integers, ordered lexicographically, just like the multiset order for integers, and show that there are no infinite descending sequences. Now we apply this to prove (6). The subterm U^A is given a multiset defined as follows. Let n be the sum of the integers in A, let m be the number of moves to the left in the path from the root of the tree to U^A, and let k be the number of lambdas in the term that are NOT on this path to U^A. The multiset assigned to U^A consists of n copies of (k, m). The multiset assigned to the whole term is the multiset union of the multisets assigned to its subterms. Any rotation reduces this multiset in the multiset order.

If we combine (2) and (4) we get the following.

(7) Every alternating sequence of beta reductions and rotations terminates.

If we combine (1) and (5) we get the following.

(8) Every alternating sequence of eta expansions and rotations eventually terminates.

Suppose that we have an infinite alternating sequence of beta reductions, eta expansions and rotations. We We may assume that there are infinitely many beta reductions and infinitely many eta expansions (why?). We shall alter this sequence as follows. First we move successive beta reductions to the beginning of the sequence. If two beta's are separated by an alternating sequence of rotations and eta expansions then all rotations not clockwise and local to the eta expansions can be permuted to be before all these etas. Then by eta postponemnet the last beta can be permuted to a position before the local rotations and these eta expansions, unless it dissappears as in two cases of eta postponement. Thus there must come a time when in every case the next beta eventually disappears in the permutation process.

(9) There is no infinite alternating sequence of beta reductions, eta expansions and rotations where in every case the next beta eventually disappears in the permutation process.

By (8), for any term, the entire tree of eta expansions and rotations is finite. Suppose that (9) is false and consider a term X, with smallest tree, that begins a counterexample to (9). In such a counterexample the next term after X in the sequence is not the result of a rotation for this would contradict the choice of X. Thus the next term in the sequence must be the result of an eta expansion and this is followed by a sequence of clockwise rotations local to the eta expansion. Now there must be at least one beta reduction which vanishes from permutation with the first eta expansion; for,again, otherwise the choice of X is contradicted. But the first such permutation effects a non trivial rotation, either clockwise or counterclockwise on X which contradicts the choice of X as one with smallest tree.

Thus by (9) we get the end result.

(10) Every alternating sequence of beta reductions, eta expansions and rotations eventually terminates.

Table of Named Lambda Te

$$
\begin{aligned}
I &:= (\lambda x x) \\
S &:= (\lambda x(\lambda y(\lambda z((xz)(yz))))) = \lambda xyz.xz(yz) \\
\omega &:= \lambda x(xx) \\
\Omega &:= (\omega\omega) = (\lambda x(xx))(\lambda x(xx)) \\
\Omega^{+} &:= (\lambda x(xxx))(\lambda x(xxx)) \\
K &:= (\lambda xy.x) \\
K^{*} &:= KI = (\lambda xy.x)I \\
B &:= \lambda xyz.x(yz) \\
C &:= \lambda xyz.xzy \\
W &:= \lambda x\lambda y\, xyy \\
K_i^n X_1 \ldots X_n &:= X_i
\end{aligned}
$$

Application

An application of a term X to another term Y is written XY.

Argument Position

In a term (UV), the V is in function position.

Boolean

A boolean can take one of two values, either true or false. We conventionally refer to true as $K := (\lambda xy.x)$ and false as $K_* := KI$.

Closed

A term X such that $FV(X) = \emptyset$. Alternately: a term which does not have any free variables. See **combinator**.

Combinator

A term X such that $FV(X) = \emptyset$. Alternately: a term which does not have any free variables. See **closed**.

Diagonalize

A function diagonalizes across its inputs by setting all of its inputs to be equal.

Fixed Point

A fixed point of a function is an element of the function's domain which is mapped to itself by that function. For example, if $f(x) = x$, then x is a fixed point of f.

Function Position

In a term (UV), the U is in function position.

Left Association

When we delete every pair of parens around a subterm which is not in argument position, we can restore them by using left association (TODO: IMPROVE THIS).

Monus Subtraction

Notated $\dot{-}$, monus subtraction is defined as usual subtraction with the additional rule that if the subtraction results in a number lower than zero, it is defined to be 0 instead.

Program

Any term in lambda calculus is referred to as a ``program.''.

Redex

In a reduction, the redex is the initial term. It is replaced by the **reductum**.

Reductum

In a reduction, the reductum is derived from the **redex**.

Ring

An algebraic structure comprised of a set and operations which generalize multiplication and addition to the elements of the set.

Tail Recursion

A tail recursive function is a function which calls itself at the end of its excecution.

Term

A term in λcalculus consists of constants and variables x, y, z and other terms, X, Y, Z, arranged in one of the following ways. First, as a variable x. Second as (XY), read ``X applied to Y.'' Third, as a lambda term $(\lambda x X)$.