Class Notes on Lambda Calculus

Chuck Liang
Hofstra University Computer Science

1 Background and Introduction

The Lambda Calculus is an abstract, mathematical basis for studying properties of programming languages just as integral calculus is used to study physics and other sciences. It was invented by Alonzo Church, who along with Alan Turing is considered one of the founders of computer science. We will study the basics of the lambda calculus for two reasons. First, most modern programming languages have direct support for "lambda expressions", and they are increasingly being used in advanced programming. Secondly, and *far more importantly,* the lambda calculus is a theoretical foundation for most of computer science: all computer algorithms can be explained by the lambda calculus (this is part of what is called the *"Church-Turing Thesis."*) It explains the behaviors of most programs and languages even when they don't use lambda expressions explicitly.

The lambda calculus has an untyped version and several typed versions. They roughly correspond to statically typed (compiler-based) and dynamically typed (interpreter-based) languages. This document will first focus on the untyped lambda calculus. Typed lambda calculi will be briefly covered at the end.

The syntax of the lambda calculus will look strange at first. Essentially it is trying to answer the question: *how do we apply a program to input?* We can study this problem in the form of *how to apply a function to an argument.* In many languages we write this as $f(t)$, where f is the function and t is the argument. In lambda calculus we write this as $(f\ t)$. Essentially, applying a program (function) to an input (argument) requires "substitution," which is to just *replace each occurrence of the formal argument with the actual argument.* The lambda calculus is the smallest and the purest programming language. It is purely mathematical in that it is not concerned about how computation is to be implemented physically, only how symbols are to be transformed.

Each language has a syntax for specifying functions. In lambda calculus functions are represented using the λ binder. So `'def f(x): return x*x'` in Python, for example, will be $f = \lambda x.(*\ x\ x)$. The $\lambda x.$ "binder" defines the (formal) argument to the function. The λ-term itself is a value, sometimes called a *nameless function,* and we are just using the symbol f to refer to it. When we apply the function to an (actual) argument, like in $(f\ 4)$, we replace each x in the body of the function with 4, which results in $4 * 4$, or $(*\ 4\ 4)$: this expression must be evaluated further using the built-in function for $*$. The evaluation continues until we reach a *"normal form,"* such as 16, which can't be evaluated any further. Sounds simple enough, but there are some complications.

Actually, in Python you can also just write `'f=lambda x:x*x'`, but even in languages that do not support lambda syntax directly, the lambda calculus is still significant. Consider the following C programs:

```
int f(int x)
{
   int y = 1;
   {
     { // starts inner scope
        int x = 2;
        y=y+x;
     }
   }
   return y+x; // which x is this referring to?
}
```

What should happen if we applied $f(4)$? We can't just replace all x's with 4: we need to recognize that there are two x's in the body, and only the outer one corresponds to the formal argument. $f(4)$ should return 7, but only if we respect the *scope* of variables. Consider another example:

```
int y = 1; // variable declared external to f (such as a global)
int f(int x)
{
   int y = 8;
   return x+y;
}
```

What should happen if we applied $f(y+1)$? You should know that it will return 10, but only because you understand that there are two y's and which y is being referred to in $x+y$. If we just blindly replaced x with $y+1$, the function would return 17.

Finally, consider the complete C program:

```
#include<stdio.h>
int x = 1;
int f(int y) { return x+y; }
int main()
{
   int x = 2;
   printf("%d\n", f(0) ); //What will this print? What SHOULD it print?
   return 0;
}
```

Which x is the x in the body of f referring to? Why?

To handle these situations correctly, we need to enhance our strategy to replace formal arguments with an actual arguments. First, we need to distinguish between local variables (called "bound variables" in the lambda calculus) and non-local variables (called "free variables").

2 Formal Definitions

An expression (term) in the lambda calculus is defined inductively as follows:

1. a variable, x, y, z etc.
2. $(A\ B)$, where A and B are both lambda terms. This is called an "application".
3. $\lambda x.A$, where x is a variable and A is a lambda term. This is called an "abstraction."
4. A known constant such as 3. Constants are not strictly necessary but we will include them in order to show more interesting examples. Constants can also be functions: for example, $*$ is used to represent the multiplication function in most languages.

For example, $\lambda x.(y\ x)$ is a lambda term because x and y are variables, so $(y\ x)$ is a term and therefore $\lambda x.(y\ x)$ is also a term.

According to this definition, (3 3) is a valid lambda term, and is read as "three applied to itself." That's fine. The lambda calculus is a purely symbolic, that is to say purely syntactic, system. Do not try to attach any *meaning* to the symbols and rules just yet.

Syntactic Conventions: Applications in lambda terms associates to the left, and applications bind tighter than abstractions. This means that $(a\ b\ c)$ is equivalent to $((a\ b)\ c)$ and **not** to $(a\ (b\ c))$. The parentheses are dropped when not needed. The term $\lambda x.x\ y$ should be understood to be equivalent to $\lambda x.(x\ y)$ and *not* to $(\lambda x.x)\ y$, because application binds tighter than abstraction: in other words, the scope of the λx binder extends to the right until it's delimited by enclosing ()s. Thus in $(\lambda x.a)\ (b\ c)$ both pairs of parentheses are needed, but in $\lambda x.((a\ b)\ c)$ both are superfluous.

Here's another way to understand the syntactic conventions. Write applications $A\ B$ as $app(A, B)$ and write abstractions $\lambda x.A$ as $abs(x, A)$. Then *application associates to the left* means that $A\ B\ C$ represents $app(app(A, B), C)$ and not $app(A, app(B, C))$. *Application binds tighter than abstraction* means that $\lambda x.A\ B$ is $abs(x, app(A, B))$ and not $app(abs(x, A), B)$.

Given an abstraction $\lambda x.M$ (where M is an arbitrary term), M is called the "body" (or *scope*) of the λ-abstraction and λx is the "binder." We can think of such an abstraction as a function that takes x then evaluates and returns M.

Free and Bound Variables

Inside a given term, variables can be local or non-local. Local variables are those that occur inside the scope of a λ-binder, and free variables are those that do not. For example, in $\lambda x.(y\ x\ x)$, x is bound and and y is free. Whether a variable is free or bound depends on the exact term in question: in the entire term $\lambda x.(x\ y)$, x is bound, but in the body $(x\ y)$, x is free. The variable x that forms a part of the binder λx can also be consider bound, but it's not part of the body of the term (it declares that a variable is bound in the body). Consider the more interesting example:

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

Observe that by the syntactic conventions, this term should be read as $\lambda x.((y\ (\lambda x.x))\ x)$. In the entire term, all occurrences of x are bound and y is free. Inside the body of the term $y\ (\lambda x.x)\ x$, however, the right-most occurrence of x is free but the one inside the parentheses is bound. The right-most x refers to the outer λx binder.

Substitution and Beta-Reduction

Now we can more formally define how to apply a function in the form of a λ-abstraction to an argument. First, we define *substitution:*. Given a term M, let $M[N/x]$ represents the term obtained by replacing *all free occurrences of x in M with N*. Furthermore, we require the following condition: *the free variables of N cannot appear bound in M*.

For example, the substitution $((\lambda x.x)\ x)[N/x]$ yields the term $(\lambda x.x)\ N$: only the *free* occurrence of x in the term is replaced with N. The bound instance of x in the term represents a local variable, and should not be replaced: in other words, $\lambda x.x$ is self-contained (the proverbial "black box").

Now we define the most critical rule of Lambda Calculus, called **"beta reduction"**:

$$(\lambda x.M)\ N \quad \Rightarrow_\beta \quad M[N/x]$$

That is, to apply a λ-abstraction (function) to an argument, substitute N for x in M under the following two conditions:

1. Only the free occurrences of x in M are to be replaced (x is bound in the entire term $(\lambda x.M)$ but it may be free in the body M).
2. The free variables contained in N may not be bound in M.

A term in the form $(\lambda x.M)\ N$ is called a *beta-redex:* a place where beta-reduction can be applied. Consider some examples:

- $(\lambda x.x + x)\ y$ beta reduces ($\Rightarrow$) to $y + y$. Here, M is $x + x$ and N is y and we replace both free occurrence of x in M with y.

- $(\lambda x.y)\ z \quad \Rightarrow \quad y$. Here, M is y and N is z. There are no free x's in the body, so there's nothing to replace: $y[z/x]$ is still y. Such lambda terms correspond to functions that do not use their arguments. The λx binder here is *vacuous.*

- $(\lambda x \lambda y.y\ x)\ u\ (\lambda v.v) \quad \Rightarrow \quad u$. It is tempting to think of a term that begins with $\lambda x \lambda y \ldots$ as a "function that takes two arguments," but it is actually a function that takes one argument and returns another function, which takes the other argument. Such functions are called "Curried" functions (referring to the logician Haskell Curry). In this example, we first replaced x with u, resulting in the intermediate term $(\lambda y.y\ u)\ (\lambda v.v)$. For this beta-redex, the argument (N) is itself a lambda-term, and we replace y with $\lambda v.v$. This results in the term $(\lambda v.v)\ u$. One final reduction reduces this to u. In other words, the reduction steps are:

  $$(\lambda x \lambda y.y\ x)\ u\ (\lambda v.v) \ \Rightarrow\ (\lambda y.y\ u)\ (\lambda v.v) \ \Rightarrow\ (\lambda v.v)\ u \Rightarrow u$$

  Note: although the parentheses around $\lambda v.v$ are not required in $(\lambda y.y\ u)\ (\lambda v.v)$, it's better to place them there because after substitution its scope should be clear.

- The next term corresponds to the second C program given in the introduction:

  $$(\lambda y \lambda x.(\lambda y.x+y)\ 8)\ 4$$

  Please read the parentheses carefully: there's an inner beta-redex, which is $(\lambda y.x+y)\ 8$ where M is $x+y$ and N is 8, and an outer beta-redex, where M is $\lambda x.(\lambda y.x+y)\ 8$ and N is 4. It does not matter which redex we reduce first (this is called the *confluence property.*). Let us reduce the outer redex first: this means substitute all free occurrences of y in the body with 4, but there are actually no free occurrences of the outer y in the body: the y inside the body is bound locally. The reduction sequence is therefore:

  $$(\lambda y \lambda x.(\lambda y.x+y)\ 8)\ 4 \ \Rightarrow\ \lambda x.(\lambda y.x+y)\ 8 \ \Rightarrow\ \lambda x.x+8$$

  In the last step, we reduced the beta-redex inside the outer λx, which is $(\lambda y.x+y)\ 8$.

In each case above we reduced the term until there are no beta-redexes anywhere in the term, resulting in a *beta normal form.*

Are all terms reducible to normal forms? **NO.** The standard example is

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

Yes, this is *a term that applies its argument to itself and is itself applied to itself, and such a term reduces to itself.* Got it? Don't even try to understand it intuitively: just apply the rules of beta reduction. Both free occurrences of x in the body $x\ x$ are replaced by the argument, which is $(\lambda x.x\ x)$ (which does not contain any free variables so the restrictions on substitution do not apply). The term reduces back to itself and therefore will never be in normal form. The existence of such terms is important if we are to have repetition (loops, recursion) in our programs.

Bad Examples of Beta Reduction:

- $(\lambda y.\lambda x.x\ y)\ v\ \Rightarrow\ (\lambda x.x)\ v\ \Rightarrow\ v$. This is **wrong** because we misused the syntactic conventions of the lambda calculus. The first reduction results in $\lambda x.x\ v$, which is not the same as $(\lambda x.x)\ v$ because application binds tighter than abstraction: it is equivalent to $\lambda x.(x\ v)$, which is already in normal form.
- $(\lambda x.y\ x)\ (\lambda u.u)\ z\ \Rightarrow (\lambda x.y\ x)\ z\ \Rightarrow y\ z$. This is wrong because application should associate to the left: the correct reduction sequence is $(\lambda x.y\ x)\ (\lambda u.u)\ z\ \Rightarrow y\ (\lambda u.u)\ z$, which is already in normal form.
- $(\lambda y.\lambda y.y\ y)\ v\ \Rightarrow\ (\lambda y.v\ v)$. We didn't make the same mistakes as above but we made a worse one. We can only replace free occurrences of y inside the body of the outer λ, which is $\lambda y.y\ y$. There are no *free* occurrences of y in this term! The correct normal form is $\lambda y.y\ y$.
- $(\lambda y.\lambda x.x\ y)\ x\ \Rightarrow\ (\lambda x.x\ x)$. In the lambda calculus, this is as bad a mistake as you could possibly make. You violated the second restriction to beta reduction: *the free variables of N cannot appear bound in M.* N in this case is the free variable x, but inside the body of the λ-term, x is bound (a local variable). From a programmer's perspective, this restriction should be intuitive: the local (bound) variable x should not be confused with externally defined (free) variables by the same name. The wrong reduction does what's called "free variable capture." Free variables must stay free after reduction.

So how do we reduce terms like in the last example? We just *rename* the bound variable to separate it from the free variable.

Alpha Conversion

Surely `int f(int y) {return y;}` and `int f(int x) {return x;}` are the same program. Not only do they behave the same, but they are syntactically the same: the only difference is in the choice of the local (bound) variable used. In lambda calculus, terms such as $\lambda x.x$ and $\lambda y.y$ are called *alpha-equivalent.* Each term can be *alpha-converted* to the other by replacing the names of bound (local) variables consistently. Formally, a variable y is called *fresh* with respect to a term M if y does not appear anywhere in M: it's a new variable[1]. Alpha conversion is defined by:

$$\lambda x.M \quad \Rightarrow_\alpha \quad \lambda y.M[y/x]$$

where y is *fresh* with respect to M.

It is important to remember that alpha-equivalence is a purely syntactic equivalence and does not assume anything about the *meaning* of symbols. For example, $\lambda x\lambda y.x + y$ and

[1]technically, it can appear in some places but to simplify things, let's assume that it's a completely new variable not found anywhere in M.

$\lambda y.\lambda x.y + x$ are alpha-equivalent (one can be converted to the other by first replacing one of the variables with an intermediate fresh variable z). However, $\lambda x \lambda y.x + y$ and $\lambda x.\lambda y.y + x$ are **not** alpha-equivalent. First, you cannot assume that the symbol '+' represents numerical addition, which is commutative. In many languages + is used for string concatenation which is not commutative ("ab" $\neq$ "ba"). Secondly, even if + is known to be commutative the two terms are still not alpha-equivalent because they are structurally different. Alpha-equivalence is a purely syntactic property.

We need alpha-conversion to allow beta reduction to proceed correctly when the names of free and bound variables clash. In the example $(\lambda y.\lambda x.x\ y)\ x$, we can apply alpha conversion to the lambda term and rewrite it as $(\lambda y.\lambda z.z\ y)\ x$, which then reduces to $\lambda z.z\ x$: the free variable x is not illegally captured by the local binder. This example roughly corresponds to the situation found in the first C program of the introduction.

We can only change the names of **bound** variables, not free variables. It would be illegal to change the outer (rightmost) x to something else. Free variables correspond to external variables which may be in use elsewhere, not just *locally.*

Eta Equivalence

Besides α-equivalence, we briefly mention another rule called eta-conversion: m is η-equivalent to $\lambda x.(m\ x)$ where x is not free in m. Both η and α are mere syntactic equivalences: only β-reduction represents real computational steps.

3 Basic Combinators

A lambda term without any free variables (completely self-contained) is called a *combinator.* There are three famous ones:

1. $I\ =\ \lambda x.x$
2. $K\ =\ \lambda x \lambda y.x$
3. $S\ =\ \lambda x \lambda y \lambda z.x\ z\ (y\ z)$

It is know that all other combinators can be generated by combining these terms in some fashion. Let's look at how they behave. First, it should be clear, if you understood the above, that $((\lambda x.x)\ A)$ beta-reduces to A, regardless of the what kind of term A is. So we can write down an **axiom:** $IA = A$. Now consider K: this terms takes two arguments (actually one after the other), and ignores the second argument. Given any two terms A and B where y does not appear free in A (alpha-convert K otherwise), $(K\ A\ B)$ reduces to A:

$$(K\ A\ B) = (\lambda x \lambda y.x)\ A\ B\ \Rightarrow\ (\lambda y.A)\ B\ \Rightarrow\ A$$

In the last step, because y is not free in A, A is not affected by the substitution $A[B/y]$. Since K can always be alpha converted away from the free variables of A, we can write down another **axiom:** $KAB = A$.

Now consider $(K\ I\ A\ B)$ where I is (alpha-equivalent to) $\lambda u.u$ and A and B are arbitrary. Recall that by syntactic convention, this is the term $((K\ I)\ A)\ B$. First we reduce the innermost redex $(K\ I)$:

$$KI = (\lambda x \lambda y.x)I \ \Rightarrow\ (\lambda y.I) = \lambda y.\lambda v.v$$

This terms takes two arguments and this time it ignores the first argument. We should see that $KIAB = (\lambda y \lambda v.v)AB \ \Rightarrow\ (\lambda v.v)B = B$. So we've another **axiom:** $KIAB = B$. We can alpha convert KI to $\lambda x \lambda y.y$.

It may appear that terms (such as K and KI), which ignore some of their arguments are pointless. But in fact, they correspond to something you're already very familiar as programmers. Given two choices A and B, K *chooses* A and KI chooses B. We are now beginning to see how lambda terms correspond to constructs found in every programming language, in this case Boolean expressions and **if-else.** But let's first look at an example involving S:

$$SKI = (\lambda x \lambda y \lambda z.x\ z\ (y\ z))\ K\ I\ \Rightarrow (\lambda y \lambda z.K\ z\ (y\ z))\ I\ \Rightarrow \lambda z.K\ z\ (I\ z)\ \Rightarrow \lambda z.z$$

In the last step, we applied the axiom $KAB = A$ where A is z and B is $(I\ z)$. Alternatively, we can first reduce $(I\ z)$ to z with the axiom $IA = A$, but the result will be the same.

4 More Powerful Combinators

When Church invented the lambda calculus, he wasn't just thinking about computing. There was a hypothesis (called "Hilbert's Program") that all mathematical problems can be algorithmically solved. Eventually, this hypothesis was proven to be false by Kurt Godel and his *incompleteness theorems.* But many (though not all) mathematical properties can be described by the lambda calculus. The most basic computational operations are adding and multiplying numbers ...

4.1 Church Numerals

Church formulated a representation of natural numbers as lambda terms: zero is $\lambda f \lambda x.x$, one is $\lambda f \lambda x.f\ x$, two is is $\lambda f \lambda x.f\ (f\ x)$, three is $\lambda f \lambda x.f\ (f\ (f\ x))$, and so on. Why did he write numbers this way? Because he wanted to study the *algorithmic* properties of numbers. The "meaning" of a number is, to him, defined by what we do with them, i.e., arithmetics. With this representation of numbers, basic arithmetic operations can be defined as lambda terms:

- $PLUS = \lambda m \lambda n \lambda f \lambda x.m\ f\ (n\ f\ x)$
- $TIMES = \lambda m \lambda n \lambda f \lambda x.m\ (n\ f)\ x$

Other operations are also possible (the one for subtraction is pretty hard). For example, we can show that $1 + 2 = 3$ using beta-reduction:

$$(PLUS\ 1\ 2) = \lambda f \lambda x.1\ f\ (2\ f\ x)\ \Rightarrow \lambda f \lambda x.1\ f\ (f\ (f\ x))\ \Rightarrow \lambda f \lambda x.f\ (f\ (f\ x)) = 3$$

I first substituted 1 and 2 for m and n respectively, then reduced $(2\ f\ x)$ to $f\ (f\ x)$. Plus works by substituting one number into the end of the other number. Times works by replacing each f with more $f's$. As an exercise, you can verify that $(TIMES\ 2\ 3) = 6$.

Note that, under η equivalence, $TIMES$ can also be written as $\lambda m \lambda n \lambda f.m\ (n\ f)$.

We won't go further into Church numerals except to say that all basic arithmetic operations can be represented as pure lambda calculus. Even though in actual languages we don't use lambda terms to represent numbers, lambda calculus provides a theoretical unity to different kinds of computation.

4.2 Booleans and If-Else

A programming language needs more than just numbers and arithmetic. Church's philosophical approach to thinking about numbers extends to truth and falsehood. What is the significance of true and false from a purely computational standpoint? It depends on what we can do with true and false. As programmers, you certainly know what to do with these booleans: we use them to make decisions. From this algorithmic standpoint, true and false just represent two opposing choices:

- $TRUE = K = \lambda x \lambda y.x$
- $FALSE = KI = \lambda x \lambda y.y$

Then the familiar *if-else* construct can be defined by:

- $IFELSE = \lambda c \lambda a \lambda b.(c\ a\ b)$

IF-ELSE takes three (Curried) arguments: a boolean expression c that should reduce to $TRUE$ or $FALSE$, and two choices a and b. Since $Kab = a$ and $KIab = b$, true *chooses* a and false *chooses* b. Thus $(IFELSE\ TRUE\ 1\ 2)$ will beta-reduce to 1 and $(IFELSE\ FALSE\ 1\ 2)$ will beta-reduce to 2.

The lambda calculus should start to look less strange to you now ...

What about the boolean operations and, or and not? All of these can be defined in terms of if-else. For example:

- $NOT = \lambda a.(IFELSE\ a\ FALSE\ TRUE)$
- $AND = \lambda a \lambda b.(IFELSE\ a\ b\ FALSE)$

That is, NOT is defined as a function that takes a as an argument and has body `if (a) return false; else return true`. AND is a function that takes a and then b with body `if (a) return b; else return false`. For example, $(AND\,TRUE\,FALSE)$ beta-reduces to (IFELSE TRUE FALSE FALSE), which reduces to FALSE. If you're not convinced, do truth tables to see that these pure lambda terms do indeed have the expected behaviors of the boolean operations.

As an exercise, define OR.

4.3 Data Structures and Encapsulation

How do we *construct* a single structure from multiple components, and how do we *destruct* (break down) such a structure into its components? This sounds like a challenge to lambda calculus, but all we need to do is to represent a pair of values, which we normally write as (a, b). Then, using nested pairs, we can construct data structures of any complexity. For example, a linked list is just a sequence of nested pairs ending in some designated value representing the empty list: $(a, (b, (c, (d, null))))$. For historical reasons the three operations we define are called *cons, car* and *cdr:*

- $CONS = \lambda a \lambda b \lambda c.(c\ a\ b)$
- $CAR = \lambda p.(p\ TRUE)$
- $CDR = \lambda p.(p\ FALSE)$

Given two terms A and B, $(CONS\ A\ B)$ returns (beta reduces to) a pair represented by $\lambda c.(c\ A\ B)$. *"This is not a pair, it's a function!"* says you. Yes, but this function has all the behaviors we want from a pair. The argument c is expected to be a boolean value and recall that $(c\ a\ b)$ is equivalent to $(IFELSE\ c\ a\ b)$: true selects a and false selects b. Given such a "pair" represented by $P = \lambda c.(c\ A\ B)$, $(CAR\ P)$ reduces to $(P\ TRUE)$, which then reduces to $(TRUE\ A\ B)$. Thus $(CAR\ P)$ returns A. $(CDR\ P)$ will similarly return B. $CONS$ constructs the structure and CAR, CDR destruct it.

To represent linked lists, we also need to choose a term for the empty list (typically called *null* or *nil:*) as well as a way to determine if a pair is empty or not.

- $NIL = \lambda x.\lambda y.y$ (same as $FALSE$ and zero).
- $ISNIL = \lambda p.p\ (\lambda x \lambda y.\lambda z.FALSE)\ TRUE$

$(ISNIL\ NIL)$ reduces to $NIL\ (\lambda x \lambda y.\lambda z.FALSE)\ TRUE$, which reduces to TRUE because NIL ignores the first argument and returns the second ($KIAB = B$). But given a non-empty pair $P = \lambda c.(c\ A\ B)$, $(ISNIL\ P)$ reduces to:

$$P\ (\lambda x \lambda y.\lambda z.FALSE)\ TRUE\ \Rightarrow (\lambda x \lambda y.\lambda z.FALSE)\ A\ B\ TRUE\ \Rightarrow FALSE.$$

Now we can define a linked list as in

$$M = (CONS\ 2\ (CONS\ 3\ (CONS\ 5\ (CONS\ 7\ NIL))))$$

and write expressions such as $(CAR\ (CDR\ M))$, which extracts the second value from the list (verify that it reduces to 3). We can also write $IFELSE\ (ISNIL\ M)\ A\ B$. Add "syntactic sugar" to make it look like your favorite language.

Once we have pairs we can build other structures of arbitrary complexity. For example, binary tree nodes can be represented by $(CONS\ X\ (CONS\ LEFT\ RIGHT))$.

Representing data structures using functions is not so strange once you realize that this is exactly what we want when constructing an *"abstract data type."* For example, we often construct a *class* that contains private values and public access functions. The public functions form the *interface* to the data type: the values that make up the data structure are thus *encapsulated.*

4.4 Repetition (Recursion)

We need another component to create a basic programming language: recursion. The usual loop constructs of ordinary languages can be seen as special cases of recursion. The idea is to create a *fixed point operator* fix with the behavior that $fix\ M$ reduces to $M\ (fix\ M)$. This will allow us to repeat arbitrarily many Ms, or form loops that repeatedly evaluates M. Combined with $IFELSE$, we can terminate the loop when some condition is reached.

- $FIX = \lambda m.(\lambda x.m\ (x\ x))\ (\lambda x.m\ (x\ x))$

This term is similar to the term we saw earlier that reduces to itself and has no normal form. We verify that:

$$(FIX\ M) \Rightarrow (\lambda x.M\ (x\ x))\ (\lambda x.M\ (x\ x))\ \Rightarrow\ M\ ((\lambda x.M\ (x\ x))\ (\lambda x.M\ (x\ x))) = M\ (FIX\ M)$$

If it ever seemed strange to you that you can define a function that refers to itself, the FIX operator doesn't actually do that: it's applied to a function that *names* the recursive function using a λ-binder. The following recursive program computes the sum of all numbers in a linked list:

$$SUM = FIX\ (\lambda f.\lambda n.IFELSE\ \ (ISNIL\ n)\ \ ZERO\ \ (PLUS\ (CAR\ n)\ (f\ (CDR\ n)))$$

The recursive function is called by the λ-bound variable f. To see how recursion works in more detail, here M is $\lambda f.\lambda n.IFELSE\ (ISNIL\, n)\ ZERO\ (PLUS\ (CAR\, n)\ (f\ (CDR\, n))$ and $SUM = FIX\ M \Rightarrow M\ (FIX\ M)$. For example,

$$SUM\ (CONS\ 2\ NIL) \Rightarrow M\ (FIX\ M)\ (CONS\ 2\ NIL) \Rightarrow$$

$$IFELSE\ (ISNIL\ (CONS\ 2\ NIL))\ ZERO\ (PLUS\ 2\ (FIX\ M\ NIL)) \Rightarrow$$

$$(PLUS\ 2\ (FIX\ M\ NIL)) \Rightarrow (PLUS\ 2\ (M\ (FIX\ M)\ NIL)) \Rightarrow (PLUS\ 2\ 0) \Rightarrow 2$$

The recursion stoped and returned 0 when $(ISNIL\ NIL)$ reduced to $TRUE$. We note that all elements of this program are definable in pure lambda calculus.

5 Properties of Programming Languages

The previous section showed how common programming language constructs can be represented by lambda terms. Towards building a complete programming language, we must also address several additional issues.

5.1 Order of Evaluation

As examples such as SKI show, there are often multiple beta-redexes inside a term and there are different strategies for selecting which one to reduce first. Most popular programming languages use *call-by value* evaluation, which means that the actual arguments to a function are fully evaluated before they're passed to the function. The alternative evaluation method is *call-by-name*, which always reduces the outermost beta redex first. Consider $S(KI)K$. If we reduced (KI) first, it would be call-by-value, and if we first substituted (KI) into S, it would be call-by-name. Either strategy will give us the same normal form $\lambda z.\lambda y.z$ (which is alpha-equivalent to K again - verify this normal form as an exercise).

The Church-Rosser Theorem: *If $A \Rightarrow_\beta B$ and $A \Rightarrow_\beta C$, then there is a term D such that $C \Rightarrow_\beta D$ and $B \Rightarrow_\beta D$.*

In particular, if a term can be reduced to a normal form, then it must be unique. This property is also called *confluence* and it says that all reduction strategies are valid, though not equally efficient: some strategies may require more steps than others. Consider $\lambda x.(x\ x\ x\ x\ x\ x\ x)\ (SKI)$. Using call-by-value, we only have to reduce SKI to a normal form once, but using call-by-name, we will have to reduce it six times. This example suggests that call-by-value is more efficient. But that's not always true: consider

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

Using call-by-name, the term reduces in one step to z, because y is a vacuous binder. How many steps will call-by-value take? Find the answer yourself, but don't take forever :).

Theorem: If a term can be reduced to a normal form, the normal form can be reached using the call-by-name strategy.

Redundant computations in call-by-name can also be avoided with more sophisticated implementations: if we used directed acyclic graphs *(DAGs)* instead of *trees* to represent terms, then redundancies such as in $\lambda x.(x\ x\ x\ x\ x\ x\ x)$ can be avoided. The main difficulty in implementing call-by-name is that we must *delay* computation until it's actually needed. This means we can't just pass a value to a function, but must pass unexecuted *code* to a function. Furthermore, the code must carry the environment under which it's defined in order to be executed correctly: it's called a *closure.*

Most conventional languages use call-by-value. A notable exception is Haskell. Some newer languages (Scala) now allow functions to be defined to use either call-by-value or call-by-name. Most conventional languages also do not reduce terms that are hidden inside λ abstractions: that is, in $(\lambda x.(\lambda y.y)\ x)A$, the beta-redex inside the λx term, $(\lambda y.y)\ x$ is not evaluated until the outer redex is reduced first. In other words, we don't evaluate a function until it's passed all of its arguments. Also, certain built-in constructs must use call-by-name. If we think of $ifelse$ as a function of three arguments then under call-by-value $(ifelse\ c\ a\ b)$ will evaluate both a and b regardless of the truth value of a, and it is clearly not how we want if-else to behave. Thus, from a practical perspective, the $IFELSE$ combinator we defined earlier can only be used with call-by-name. Thus in most conventional languages, constructs such as if-else (and short-circuited booleans) are implemented as special constructs and not just as built-in functions.

To simulate the effect of call-by-name in the call-by-value setting of conventional languages, we have to encapsulate b and c, the terms that cannot be evaluated eagerly, inside vacuous λ-abstractions:

- $ifelse = \lambda c \lambda a \lambda b.(c\ a\ b)\ I$

Instead of calling $(IFELSE\ C\ A\ B)$, we call $(ifelse\ C\ (\lambda x.A)\ (\lambda x.B))$. Here, x is assumed to not appear free in A or B (alpha-convert otherwise). The dummy (vacuous) λx delays the evaluation of A and B. They are not evaluated until $(c\ a\ b)$ is applied to I.

The FIX combinator requires a similar treatment. The FIX combinator is also called the Y combinator:

- *Call-by-value FIX combinator:* $Y = \lambda m.(\lambda x.m\ (\lambda y.x\ x\ y))\ (\lambda x.m\ (\lambda y.x\ x\ y))$

We will need these versions of the combinators in our implementation of pure lambda calculus in real programming languages, such as Python and Perl.

5.2 Static Scoping

A critical subject we now address in lambda calculus concerns the third C program in the introduction. This program shows the difference between static (or lexical) scoping and dynamic scoping. C, like most languages, is *statically* scoped. The third C program prints 1 because the free variable x in the body of the function f refers to the static context in which f was *defined.* With dynamic scoping, f would refer to the nearest declared x in its *runtime* environment, which would be the x declared in main. Why is C (and most languages) statically scoped? There are theoretical and practical ways to explain this. Theoretically, the lambda calculus shows that it should definitely be statically scoped.

When we reduce $(\lambda x.M)\ N$, x is "bound" to N and then evaluated in M. We can define a *let* construct that binds variables to values in this way:

- $(let\ x = A\ in\ B)\ =\ (\lambda x.B)\ A$

We can now rewrite the third C program as follows:

$$let\ x = 1\ in\ (let\ f = (\lambda y.x + y)\ in\ (let\ x = 2\ in\ (f\ 0)))$$

This translates into the λ-term

$$(\lambda x.(\lambda f.(\lambda x.f\ 0)\ 2)\ (\lambda y.x + y))\ 1$$

Clearly, the x inside $\lambda y.x + y$ refers to the outermost λx. The inner λx is a vacuous abstraction: this x is not used anywhere. This term beta-reduces to 1 (verify this yourself). The scoping rules of lambda calculus are those of static scoping.

5.3 The Undecidability of the Halting Problem

To complete this introduction to untyped lambda calculus, we prove in pure lambda calculus the most important theoretical result in computer science: the Halting Problem. Programs can take other programs as input and attempt to analyze their behavior: this is called *static analysis.* Examples of such programs include compilers and malware detectors. The Halting Problem is: does there exists a program that, given the source code of any program *as well as its input,* will return true if that program terminates (halts) on that input, and false if that program does not terminate. The problem is *undecidable* because there cannot be such a program: the very existence of such a program will lead to a logical contradiction.

The Halting problem can be formulated in terms of Turing Machines as well as the (untyped) lambda calculus and several other equivalent formal systems. The *Church-Turing Thesis* is that all algorithms can be formulated as Turing machines, or as pure lambda terms. Therefore, the following proof is valid for all languages that claim to be "Turing Complete:"

The proof is by contradiction (*reductio ad absurdum*): we make an assumption and show that the assumption leads to an absurdity or paradox.

1. Assume that there is a pure lambda term $HALT$ such that, given any lambda terms P and A, $(HALT\ P\ A)$ reduces to $TRUE$ if $(P\ A)$ is reducible to a beta normal form, and that $(HALT\ P\ A)$ reduces to $FALSE$ if $(P\ A)$ cannot be reduced to a normal form.

2. Let $INF = (\lambda x.x\ x)\ (\lambda x.x\ x)$. Recall that this term has no normal form: it does not halt.

3. Let $Q = \lambda p.(IFELSE\ (HALT\ p\ p)\ \ INF\ \ I)$. That is, if p halts on itself as input, go into an infinite loop with INF, otherwise, return a normal form with I.

4. Consider the question *DOES* $(Q\ Q)$ *HALT?*. That is, is $(Q\ Q)$ reducible to a normal form? Based on the assumption that $(HALT\ Q\ Q)$ returns $TRUE$ if $(Q\ Q)$ halts and $FALSE$ otherwise, we see from the definition of Q that

   if $(Q\ Q)$ *halts then* $(Q\ Q)$ *does not halt, and if* $(Q\ Q)$ *does not halt then it halts.*

5. We've reached a contradiction in that $(Q\ Q)$ halts and does not halt. The terms for $IFELSE$, $TRUE$, $FALSE$, I and INF are all pure lambda terms. The only other term we used was $HALT$ and its assumed behavior. Therefore, the assumption that $HALT$ exists must be false.

The Halting Problem defines the limits of computation based on the Church-Turing model. Many other undecidable problems are proved by reducing them to the halting problem. In practical terms, the undecidibility of this problem means that we cannot always determine how a program will behave *even if we know its input* without actually running the program. But running the program could result in an infinite loop. This means we can't always experimentally determine if there are errors in our programs. Obviously, these results are relevant to the design and use of programming languages.

To conclude our presentation of the untyped lambda calculus, find a Python program in Appendix A that implements most of the contents of these notes.

6 Typed Lambda Calculus

Types are critically important in high level programming. An expression such as `"abc"/2.5` it is considered nonsense because it does not *type check.* Virtually all languages rely on type information to some degree. Even at the machine level, where all values are binary numbers, one can argue that types exist in the form of the sizes of data in bytes. Certainly all higher level languages recognize types. In addition to rejecting nonsense, types allow compilers to generate optimized code, and is an important reason why compiled languages perform better than interpreted ones. Even languages such as Python should not be called "untyped:" the difference is that they only check types at *runtime* (aka dynamic typing).

Formally, there is a close relationship between type theory and mathematical logic. A logically consistent program is *well typed.* There are various versions of *"typed lambda calculi."* The most basic one was also invented by Church and is known as ...

6.1 The Simply Typed Lambda Calculus

The type of a lambda-abstraction $\lambda x.M$ is of the form $a \to b$ where a is the type of the argument (λ-bound variable x) and b is the type of the body M. For example, $I = \lambda x.x$ has type $a \to a$ where a can be replaced by any type (int, string, etc). $K = \lambda x \lambda y.x$ has type $a \to b \to a$, with the $\to$ operator on types associating to the right. Variables are associated with types that they're assumed to have, using notation such as $x : int$. From a set of assumptions we derive a type on the right of the $\vdash$ symbol using the following rules:

$$\frac{}{x : a, \ldots \vdash x : a} \qquad \frac{x : a, \ldots \vdash M : b}{\ldots \vdash \lambda x.M : a \to b} \qquad \frac{\ldots \vdash M : a \to b \qquad \ldots \vdash N : a}{\ldots \vdash (M\ N) : b}$$

You should be able to derive the types for I and K using the first two rules. The last rule correspond to the classical syllogism *modus ponens* and establishes the correspondence between types and logic. That is, if $\to$ can be read as *implies* then the type of every combinator must be a propositional *tautology* because the rules of typing are isomorphic to the rules of logic (this is called the *Curry-Howard Isomorphism*). The following is a sample type derivation using the rules:

$$\cfrac{\cfrac{\cfrac{\cfrac{}{y : a \to b \vdash y : a \to b} \qquad \cfrac{}{x : a \vdash x : a}}{x : a, y : a \to b \vdash (y\ x) : b}}{x : a \vdash \lambda y : y\ x : (a \to b) \to b}}{\vdash \lambda x \lambda y.y\ x : a \to (a \to b) \to b}$$

You can verify that the derived type is a tautology by constructing a truth table.

The typing rules can be used for both *type checking* as well as *type inference.* Advanced languages can infer types even when type information is missing in the source code.

An important consequence of introducing types is that beta-reduction (and alpha/eta conversion) must preserve the types of terms:

Theorem: Type Soundness. *If $\vdash s : A$ is a valid type derivation for lambda term s and $s \Rightarrow_\beta t$, then $\vdash t : A$ is also a valid type derivation.*

This core result (also referred to as *subject reduction*) is what we mean by a computation being *type safe:* 3+1 is an expression of type int, and 4, the normal form, is also an int. So type checking can be done before, during, or after beta-reduction. If it's done before reduction, we can call it *static typing.* If done during reduction, it can be called *dynamic typing.*

An example of a term that cannot be typed is a free variable x by itself, not bound at any level. Another untypable term is $\lambda x.(x\ x)$. The type inference rules imply that the bound variable x must have *both* type a and type $a \to a$. The type expression itself would have to be infinitely long, which is not allowed.

Although beta reduction, alpha conversion, and the static scoping of free and bound variables remain the same, we cannot use some of the same constructs from the untyped lambda calculus to build a programming language. For example, $\lambda x \lambda y.x$ has type $a \to b \to a$ but $\lambda x \lambda y.y$ has type $a \to b \to b$. These are distinct types: we cannot call both "bool". Forcing them to be of the same type will restrict both to $a \to a \to a$, but this means, for example, we would not be able to construct pairs containing different types of values such as nested pairs (the *car* and the *cdr* must be of the same type).

Simply typed, and many other typed lambda calculi also contain the following property:

Theorem: Strong Normalization. *There are no infinite beta-reduction sequences.*

Combined with the Church-Rosser property (which still holds), this means that every reduction must terminate in the same normal form.

This result implies we cannot write programs with loops (or recursion) without certain additional constructs. The fixed pointer operator, with property $FIX\ M = M\ (FIX\ M)$, must be imported as an extra constant. If we are to give FIX a type, it would have to be $(a \to a) \to a$. This is a *not a tautology,* so no pure lambda term can have this type.

Thus, the simply typed lambda calculus is insufficient for creating a programming language. A more elaborate type system is required, one with additional terms that allow us to define data structures and recursive programs.

6.2 Algebraic Types

We choose to present a type system close to that of many typed, functional programming languages. In addition to $\to$, this type system allows several other *type constructors* including $*$ for cartesian product (pairs) and $|$ for disjoint union (sums). The system also allows the definition of recursive types via a μ constructor, though this constructor is usually omitted syntactically. These types constructors allow us to define data structures that are referred to as "algebraic data types."

For example, the following is a possible type definition for binary trees containing integers, using the syntax of $F^\sharp$:

$$type\ Bintree\ =\ Nil \mid Node\ of\ int * Bintree * Bintree$$

Here, the | operator states that a bintree is constructed in one of two ways: an empty tree called Nil, or a $Node$. A Node is defined as a product (tuple) containing an integer and a pair of other Bintrees. The definition is implicitly recursive (using the hidden μ operator, which is like a fixedpoint operator for types). The lambda calculus is extended with terms that can represent such types. A sample term of type Bintree is

$$Node(5, Node(2, Nil, Nil), Node(8, Nil, Nil))$$

This is a tree with 5 at the root, 2 to the left and 8 to the right.

Algebraic types define precisely how data can be constructed as well as *deconstructed,* leading to a programming feature called *pattern matching.*

6.3 Quantification over Types.

Types can also be abstracted over (made generic). Most programmers are already familiar with templates/generics. In terms of logic, a generic type variable is a kind of universally quantified ($\forall$) variable. We sometimes write $\Pi a.a \rightarrow a$ for the type of a function of generic type. Most languages use the angle-bracket syntax for quantified type variables:

```
  type Bintree<T> = Nil | Node of T * Bintree<T> * Bintree<T>
```

Note that all type variables must be bound (quantified), so

```
  type Bintree<T> = Nil | Node of T * Bintree<A> * Bintree<B>
```

is not a logically acceptable definition: A and B are not bound.

6.4 Subtyping

A term cannot have conflicting types: something can't be both a circle and a rectangle. However, a circle is also a "shape." This is called subtyping. The typical notation is $A : B$ meaning that A is a subtype of B. Subtyping establishes a "is-a" relationship between types and their terms: a circle *is a* shape. Subtyping is most typically found as inheritance in object oriented programming languages. Its benefits are debatable but that's a story for another day ...

APPENDIX

A Pure Untyped Lambda Calculus in Python

Most non-statically typed scripting languages support untyped lambda terms directly. We summarize the creation of a basic programming language from pure lambda calculus by implementing it using pure lambda terms in Python.

```
# Pure Lambda Calculus in Python (works in both Python2.2+ and Python3)

## The syntax of Python differs from traditional lambda calculus in that
# application is written f(x) instead of (f x), and a : is used for .

I = lambda x:x
K = lambda x:lambda y:x
S = lambda x:lambda y:lambda z:x(z)(y(z))

# Note: it would not be correct to define K as lambda x,y:x, because that's
# using python's built-in pairs.  Every lambda-abstraction takes exactly
# one argument, although that argument can be a tuple.  To be faithful to
# the build-from-scratch approach, we should use Curried functions and
# not rely on built-in features as much as possible: f(a,b) is not the same
# as f(a)(b).

true = K
false = K(I)
ifelse = lambda c:lambda a:lambda b:c(a)(b)
if_else = lambda c:lambda a:lambda b:c(a)(b)(I) # simulates call-by-name
NOT = lambda a:a(false)(true)

print(ifelse(NOT(true))(1)(2)) ### prints 2

# Church numerals
ZERO = false
ONE = lambda f:lambda x:f(x)
PLUS = lambda m:lambda n:lambda f:lambda x:m(f)(n(f)(x))
TIMES = lambda m:lambda n:lambda f:lambda x:m(n(f))(x)

# linked lists
CONS = lambda a:lambda b:lambda c:c(a)(b)
CAR = lambda p:p(true)
CDR = lambda p:p(false)
NIL = ZERO
```

```
ISNIL = lambda p:p(lambda x:lambda y:lambda z:false)(true)

# applicative order FIX combinator for recursive definitions
FIX = lambda m:(lambda x:m(lambda y:x(x)(y)))(lambda x:(m(lambda y:x(x)(y))))

Primes = CONS(2)(CONS(3)(CONS(5)(CONS(7)(CONS(11)(NIL)))))
print(CAR(CDR(Primes)))  ### prints 3, the second prime

### functions to convert Python numerals to Church numerals:

def Church(n):  # converts non-neg integer n to Church numeral
    if (n==0): return ZERO
    elif n>0: return PLUS(ONE)(Church(n-1))
#
def Unchurch(c):  # converts Church numeral c to Python integer:
    return c(lambda y:1+y)(0)
#
## for example, Unchurch(TWO) beta-reduces to
# (lambda f:lambda x:f(f(x)))(lambda y:1+y)(0), which reduces to 1+1+0=2

# Unfortunately, Python cannot print intermediate lambda terms: it does
# does not support "higher order abstract syntax".

C25 = Church(25)  # pure lambda representation of 25
print(Unchurch(C25))  #### prints 25

## pure lambda function to add all numbers in a linked list:
SUM = FIX(lambda f:lambda n:if_else(ISNIL(n))(lambda y:ZERO)(
    lambda y:PLUS(CAR(n))(f(CDR(n)))))

L = CONS(Church(1))(CONS(Church(2))(CONS(Church(4))(CONS(Church(8))(NIL))))

ANSWER = SUM(L)
print(Unchurch(ANSWER))  ### prints 15
# end of program
```