Lecture Notes

Math 4680/5680, Topics in Logic and Computation
Peter Selinger
Dalhousie University

Fall 2007

1 Introduction

1.1 Extensional vs. intensional view of functions

What is a function? In modern mathematics, the prevalent notion is that of "functions as graphs": each function f has a fixed domain X and codomain Y, and a function $f : X \to Y$ is a set of pairs $f \subseteq X \times Y$ such that for each $x \in X$, there exists exactly one $y \in Y$ such that $(x, y) \in f$. Two functions $f, g : X \to Y$ are considered equal if they yield the same output on each input, i.e., $f(x) = g(x)$ for all $x \in X$. This is called the *extensional* view of functions, because it specifies that the only thing observable about a function is how it maps inputs to outputs.

However, before the 20th century, functions were rarely looked at in this way. An older notion of functions as that of "functions as rules". In this view, to give a function means to give a rule for how the function is to be calculated. Often, such a rule can be given by a formula, for instance, the familiar $f(x) = x^2$ or $g(x) = \sin(e^x)$ from calculus. As before, two functions are *extensionally* equal if they have the same input-output behavior; but now we can also speak of another notion of equality: two functions are *intensionally*[1] equal if they are given by (essentially) the same formula.

When we think of functions as given by formulas, it is not always necessary to know the domain and codomain of a function. Consider for instance the function

[1] Note that this word is intentionally spelled "intensionally".

$f(x) = x$. This is, of course, the identity function. We may regard it as a function $f : X \to X$ for *any* set X.

In most of mathematics, the "functions as graphs" paradigm is the most elegant and appropriate way of dealing with functions. Graphs define a more general class of functions, because it includes functions that are not necessarily given by a rule. Thus, when we prove a mathematical statement such as "any differentiable function is continuous", we really mean this is true *all* functions (in the mathematical sense), not just those functions for which a rule can be given.

On the other hand, in computer science, the "functions as rules" paradigm is often more appropriate. Think of a computer program as defining a function which maps input to output. Most computer programmers (and users) do not only care about the extensional behavior of a program (which inputs are mapped to which outputs), but also about *how* the output is calculated: How much time does it take? How much memory and disk space is used in the process? How much communication bandwidth is used? These are intensional questions having to do with the particular way in which a function was defined.

1.2 The lambda calculus

The lambda calculus is a theory of *functions as formulas*. It is a system for manipulating functions as *expressions*.

Let us begin by looking at another well-known language of expressions, namely arithmetic. Arithmetic expressions are made up from variables ($x, y, z \ldots$), numbers ($1, 2, 3, \ldots$), and operators ("$+$", "$-$", "$\times$" etc.). An expression such as $x+y$ stands for the *result* of an addition (as opposed to an *instruction* to add, or the *statement* that something is being added). The great advantage of this language is that expressions can be nested without any need to mention the intermediate results explicitly. So for instance, we write

$$A = (x + y) \times z^2,$$

and not

let $w = x + y$, then let $u = z^2$, then let $A = w \times u$.

The latter notation would be tiring and cumbersome to manipulate.

The lambda calculus extends the idea of an expression language to include functions. Where we normally write

Let f be the function $x \mapsto x^2$. Then consider $A = f(5)$, in the lambda calculus we just write

$$A = (\lambda x.x^2)(5).$$

The expression $\lambda x.x^2$ stands for the function that maps x to x^2 (as opposed to the *statement* that x is being mapped to x^2). As in arithmetic, we use parentheses to group terms.

It is understood that the variable x is a *local* variable in the term $\lambda x.x^2$. Thus, it does not make any difference if we write $\lambda y.y^2$ instead. A local variable is also called a *bound* variable.

One advantage of the lambda notation is that it allows us to easily talk about *higher-order* functions, i.e., functions whose inputs and/or outputs are themselves functions. An example is the operation $f \mapsto f \circ f$ in mathematics, which takes a function f and maps it to $f \circ f$, the composition of f with itself. In the lambda calculus, $f \circ f$ is written as

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

and the operation that maps f to $f \circ f$ is written as

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

The evaluation of higher-order functions can get somewhat complex; as an example, consider the following expression:

$$\big((\lambda f.\lambda x.f(f(x)))(\lambda y.y^2)\big)\,(5)$$

Convince yourself that this evaluates to 625. Another example is given in the following exercise:

Exercise 1.1. Evaluate the lambda-expression

$$\Big(\big(\,(\lambda f.\lambda x.f(f(f(x))))\,(\lambda g.\lambda y.g(g(y)))\,\big)(\lambda z.z+1)\Big)(0).$$

We will soon introduce some conventions for reducing the number of parentheses in such expressions.

1.3 Untyped vs. typed lambda-calculi

We have already mentioned that, when considering "functions as rules", is not always necessary to know the domain and codomain of a function ahead of time.

The simplest example is the identity function $f = \lambda x.x$, which can have any set X as its domain and codomain, as long as domain and codomain are equal. We say that f has the *type* $X \to X$. Another example is the function $g = \lambda f.\lambda x.f(f(x))$ which we encountered above. One can check that g maps any function $f : X \to X$ to a function $g(f) : X \to X$. In this case, we say that the type of g is

$$(X \to X) \to (X \to X).$$

By being flexible about domains and codomains, we are able to manipulate functions in ways that would not be possible in ordinary mathematics. For instance, if $f = \lambda x.x$ is the identity function, then we have $f(x) = x$ for *any* x. In particular, we can take $x = f$, and we get

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

Note that the equation $f(f) = f$ never makes sense in ordinary mathematics, since it is not possible (for set-theoretic reasons) for a function to be included in its own domain.

As another example, let $\omega = \lambda x.x(x)$.

Exercise 1.2. What is $\omega(\omega)$?

We have several options regarding types in the lambda calculus.

- *Untyped lambda calculus.* In the untyped lambda calculus, we never specify the type of any expression. Thus we never specify the domain or codomain of any function. This gives us maximal flexibility. It is also very unsafe, because we might run into situations where we try to apply a function to an argument that it does not understand.

- *Simply-typed lambda calculus.* In the simply-typed lambda calculus, we always completely specify the type of every expression. This is very similar to the situation in set theory. We never allow the application of a function to an argument unless the type of the argument is the same as the domain of the function. Thus, terms such as $f(f)$ are ruled out, even if f is the identity function.

- *Polymorphically typed lambda calculus.* This is an intermediate situation, where we may specify, for instance, that a term has a type of the form $X \to X$ for all X, without actually specifying X.

As we will see, each of these alternatives has dramatically different properties from the others.

1.4 Lambda calculus and computability

In the 1930's, several people were interested in the question: what does it mean for a function $f : \mathbb{N} \to \mathbb{N}$ to be *computable*? An informal definition of computability is that there should be a pencil-and-paper method allowing a trained person to calculate $f(n)$, for any given n. The concept of a pencil-and-paper method is not so easy to formalize. Three different researchers attempted to do so, resulting in the following definitions of computability:

1. **Turing** defined an idealized computer we now call a *Turing machine*, and postulated that a function is computable (in the intuitive sense) if and only if it can be computed by such a machine.
2. **Gödel** defined the class of *general recursive functions* as the smallest set of functions containing all the constant functions, the successor function, and closed under certain operations (such as compositions and recursion). He postulated that a function is computable (in the intuitive sense) if and only if it is general recursive.
3. **Church** defined an idealized programming language called the *lambda calculus*, and postulated that a function is computable (in the intuitive sense) if and only if it can be written as a lambda term.

It was proved by Church, Kleene, Rosser, and Turing that all three computational models were equivalent to each other, i.e., each model defines the same class of computable functions. Whether or not they are equivalent to the "intuitive" notion of computability is a question that cannot be answered, because there is no formal definition of "intuitive computability". The assertion that they are in fact equivalent to intuitive computability is known as the *Church-Turing thesis*.

1.5 Connections to computer science

The lambda calculus is a very idealized programming language; arguably, it is the simplest possible programming language that is Turing complete. Because of its simplicity, it is a useful tool for defining and proving properties of programs.

Many real-world programming languages can be regarded as extensions of the lambda calculus. This is true for all *functional programming languages*, a class that includes Lisp, Scheme, Haskell, and ML. Such languages combine the lambda calculus with additional features, such as data types, input/output, side effects,

udpateable memory, object orientated features, etc. The lambda calculus provides a vehicle for studying such extensions, in isolation and jointly, to see how they will affect each other, and to prove properties of programming language (such as: a well-formed program will not crash).

The lambda calculus is also a tool used in compiler construction, see e.g. [8, 9].

1.6 Connections to logic

In the 19th and early 20th centuries, there was a philosophical dispute among mathematicians about what a proof is. The so-called *constructivists*, such as Brower and Heyting, believed that to prove that a mathematical object exists, one must be able to construct it explicitly. *Classical logicians*, such as Hilbert, held that it is sufficient to derive a contradiction from the assumption that it doesn't exist.

Ironically, one of the better-known examples of a non-constructive proof is Brower's proof of his own fixpoint theorem, which states that every continuous function on the unit disc has a fixpoint. The proof is by contradiction and does not give any information on the location of the fixpoint.

The connection between lambda calculus and constructive logics is via the "proofs-as-programs" paradigm. To a constructivist, a proof (of an existence statement) must be a "construction", i.e., a program. The lambda calculus is a notation for such programs, and it can also be used as a notion for (constuctive) proofs.

For the most part, constructivism has not prevailed as a philosophy in mainstream mathematics. However, there has been renewed interest in constructivism in the second half of the 20th century. The reason is that constructive proofs give more information than classical ones, and in particular, they allow one to compute solutions to problems (as opposed to merely knowing the existence of a solution). The resulting algorithms can be useful in computational mathematics, for instance in computer algebra systems.

1.7 Connections to mathematics

One way to study the lambda calculus is to give mathematical models of it, i.e., to provide spaces in which lambda terms can be given meaning. Such models are constructed using methods from algebra, partially ordered sets, topology, category theory, and other areas of mathematics.

1.8 Bibliography

Here are some textbooks and other books on the lambda calculus. None of them are required reading for the course, but you may nevertheless find it interesting or helpful to browse them. I will try to put them on reserve in the library, to the extent that they are available.

[1] is a standard reference handbook on the lambda calculus. [2]–[4] are textbooks on the lambda calculus. [5]–[7] are textbooks on the semantics of programming languages. Finally, [8]–[9] are textbooks on writing compilers for functional programming languages. They show how the lambda calculus can be useful in a more practical context.

[1] H. P. Barendregt. *The Lambda Calculus, its Syntax and Semantics*. North-Holland, 2nd edition, 1984.

[2] J.-Y. Girard, Y. Lafont, and P. Taylor. *Proofs and Types*. Cambridge University Press, 1989.

[3] J.-L. Krivine. *Lambda-Calculus, Types and Models*. Masson, 1993.

[4] G. E. Révész. *Lambda-Calculus, Combinators and Functional Programming*. Cambridge University Press, 1988.

[5] G. Winskel. *The Formal Semantics of Programming Languages. An Introduction*. MIT Press, London, 1993.

[6] J. C. Mitchell. *Foundations for Programming Languages*. MIT Press, London, 1996.

[7] C. A. Gunter. *Semantics of Programming Languages*. MIT Press, 1992.

[8] S. L. Peyton Jones. *The Implementation of Functional Programming Languages*. Prentice-Hall, 1987.

[9] A. W. Appel. *Compiling with Continuations*. Cambridge University Press, 1992.

2 The untyped lambda calculus

2.1 Syntax

The lambda calculus is a *formal language*. The expressions of the language are called *lambda terms*, and we will give rules for manipulating them.

Definition. Assume given an infinite set $\mathcal{V}$ of *variables*, denoted by x, y, z etc. The set of lambda terms is given by the following Backus-Naur Form:

$$\text{Lambda terms:} \quad M, N ::= x \bigm| (MN) \bigm| (\lambda x.M)$$

The above Backus-Naur Form (BNF) is a convenient abbreviation for the following equivalent, more traditionally mathematical definition:

Definition. Assume given an infinite set $\mathcal{V}$ of variables. Let A be an alphabet consisting of the elements of $\mathcal{V}$, and the special symbols "(", ")", "λ", and ".". Let A^* be the set of strings (finite sequences) over the alphabet A. The set of lambda terms is the smallest subset $\Lambda \subseteq A^*$ such that:

- Whenever $x \in \mathcal{V}$ then $x \in \Lambda$.
- Whenever $M, N \in \Lambda$ then $(MN) \in \Lambda$.
- Whenever $x \in \mathcal{V}$ and $M \in \Lambda$ then $(\lambda x.M) \in \Lambda$.

Comparing the two equivalent definitions, we see that the Backus-Naur Form is a convenient notation because: (1) the definition of the alphabet can be left implicit, (2) the use of distinct meta-symbols for different syntactic classes (x, y, z for variables and M, N for terms) eliminates the need to explicitly quantify over the sets $\mathcal{V}$ and Λ. In the future, we will always present syntactic definitions in the BNF style.

The following are some examples of lambda terms:

$$(\lambda x.x) \qquad ((\lambda x.(xx))(\lambda y.(yy))) \qquad (\lambda f.(\lambda x.(f(fx))))$$

Note that in the definition of lambda terms, we have built in enough mandatory parentheses to ensure that every term $M \in \Lambda$ can be uniquely decomposed into subterms. This means, each term $M \in \Lambda$ is of precisely one of the forms x, (MN), $(\lambda x.M)$. Terms of these three forms are called *variables*, *applications*, and *lambda abstractions*, respectively.

We use the notation (MN), rather than $M(N)$, to denote the application of a function M to an argument N. Thus, in the lambda calculus, we write (fx) instead of the more traditional $f(x)$. This allows us to economize more efficiently on the use of parentheses. To avoid having to write an excessive number of parentheses, we establish the following conventions for writing lambda terms:

Convention.

- We omit outermost parentheses. For instance, we write MN instead of (MN).
- Applications associate to the left; thus, MNP means $(MN)P$. This is convenient when applying a function to a number of arguments, as in $fxyz$, which means $((fx)y)z$.
- The body of a lambda abstraction (the part after the dot) extends as far to the right as possible. In particular, $\lambda x.MN$ means $\lambda x.(MN)$, and not $(\lambda x.M)N$.
- Multiple lambda abstractions can be contracted; thus $\lambda xyz.M$ will abbreviate $\lambda x.\lambda y.\lambda z.M$.

It is important to note that this convention is only for notational convenience; it does not affect the "official" definition of lambda terms.

2.2 Free and bound variables, α-equivalence

In our informal discussion of lambda terms, we have already pointed out that the terms $\lambda x.x$ and $\lambda y.y$, which differ only in the name of their bound variable, are essentially the same. We will say that such terms are α-equivalent, and we write $M =_\alpha N$. In the rare event that we want to say that two terms are precisely equal, symbol for symbol, we say that M and N are *identical* and we write $M \equiv N$. We reserve "$=$" as a generic symbol used for different purposes.

An occurrence of a variable x inside a term of the form $\lambda x.N$ is said to be *bound*. The corresponding λx is called a *binder*, and we say that the subterm N is the *scope* of the binder. A variable occurrence that is not bound is *free*. Thus, for example, in the term

$$M \equiv (\lambda x.xy)(\lambda y.yz),$$

x is bound, but z is free. The variable y has both a free and a bound occurrence. The set of free variables of M is $\{y, z\}$.

More generally, the set of free variables of a term M is denoted $FV(M)$, and it is defined formally as follows:

$$\begin{array}{lcl} FV(x) & = & \{x\}, \\ FV(MN) & = & FV(M) \cup FV(N), \\ FV(\lambda x.M) & = & FV(M) \setminus \{x\}. \end{array}$$

This definition is an example of a definition by recursion on terms. In other words, in defining $FV(M)$, we assume that we have already defined $FV(N)$ for all subterms of M. We will often encounter such recursive definitions, as well as inductive proofs.

Before we can formally define α-equivalence, we need to define what it means to *rename* a variable in a term. If x, y are variables, and M is a term, we write $M\{y/x\}$ for the result of renaming x as y in M. Renaming is formally defined as follows:

$$\begin{array}{lcll} x\{y/x\} & \equiv & y, & \\ z\{y/x\} & \equiv & z, & \text{if } x \neq z, \\ (MN)\{y/x\} & \equiv & (M\{y/x\})(N\{y/x\}), & \\ (\lambda x.M)\{y/x\} & \equiv & \lambda y.(M\{y/x\}), & \\ (\lambda z.M)\{y/x\} & \equiv & \lambda z.(M\{y/x\}), & \text{if } x \neq z. \end{array}$$

Note that this kind of renaming replaces all occurrences of x by y, whether free, bound, or binding. We will only apply it in cases where y does not already occur in M.

Finally, we are in a position to formally define what it means for two terms to be "the same up to renaming of bound variables":

Definition. We define *α-equivalence* to be the smallest congruence relation $=_\alpha$ on lambda terms, such that for all terms M and all variables y that do not occur in M,

$$\lambda x.M =_\alpha \lambda y.(M\{y/x\}).$$

Recall that a relation on lambda terms is an equivalence relation if it satisfies rules (*refl*), (*symm*), and (*trans*). It is a congruence if it also satisfies rules (*cong*) and (ξ). Thus, by definition, α-equivalence is the smallest relation on lambda terms satisfying the six rules in Table 1.

It is easy to prove by induction that any lambda term is α-equivalent to another term in which the names of all bound variables are distinct from each other and from any free variables. Thus, when we manipulate lambda terms in theory and

$$
\begin{array}{llll}
(\mathit{refl}) & \dfrac{}{M = M} & (\mathit{cong}) & \dfrac{M = M' \qquad N = N'}{MN = M'N'} \\[2ex]
(\mathit{symm}) & \dfrac{M = N}{N = M} & (\xi) & \dfrac{M = M'}{\lambda x.M = \lambda x.M'} \\[2ex]
(\mathit{trans}) & \dfrac{M = N \qquad N = P}{M = P} & (\alpha) & \dfrac{y \not\in M}{\lambda x.M = \lambda y.(M\{y/x\})}
\end{array}
$$

Table 1: The rules for alpha-equivalence

in practice, we can (and will) always assume without loss of generality that bound variables have been renamed to be distinct. This convention is called *Barendregt's variable convention*.

As a remark, the notions of free and bound variables and α-equivalence are of course not particular to the lambda calculus; they appear in many standard mathematical notations, as well as in computer science. Here are four examples where the variable x is bound.

$\int_0^1 x^2\, dx$

$\sum_{x=1}^{10} \frac{1}{x}$

$\lim_{x\to\infty} e^{-x}$

```
int succ(int x) { return x+1; }
```

2.3 Substitution

In the previous section, we defined a renaming operation, which allowed us to replace a variable by another variable in a lambda term. Now we turn to a less trivial operation, called *substitution*, which allows us to replace a variable by a lambda term. We will write $M[N/x]$ for the result of replacing x by N in M. The definition of substitution is complicated by two circumstances:

1. We should only replace *free* variables. This is because the names of bound variables are considered immaterial, and should not affect the result of a substitution. Thus, $x(\lambda xy.x)[N/x]$ is $N(\lambda xy.x)$, and not $N(\lambda xy.N)$.

2. We need to avoid unintended "capture" of free variables. Consider for example the term $M \equiv \lambda x.yx$, and let $N \equiv \lambda z.xz$. Note that x is free in N and bound in M. What should be the result of substituting N for y in M? If we do this naively, we get

$$M[N/y] = (\lambda x.yx)[N/y] = \lambda x.Nx = \lambda x.(\lambda z.xz)x.$$

However, this is not what we intended, since the variable x was free in N, and during the substitution, it got bound. We need to account for the fact that the x that was bound in M was not the "same" x as the one that was free in N. The proper thing to do is to rename the bound variable *before* the substitution:

$$M[N/y] = (\lambda x'.yx')[N/y] = \lambda x'.Nx' = \lambda x'.(\lambda z.xz)x'.$$

Thus, the operation of substitution forces us to sometimes rename a bound variable. In this case, it is best to pick a variable from $\mathcal{V}$ that has not been used yet as the new name of the bound variable. A variable that is currently unused is called *fresh*. The reason we stipulated that the set $\mathcal{V}$ is infinite was to make sure a fresh variable is always available when we need one.

Definition. The (capture-avoiding) *substitution* of N for free occurrences of x in M, in symbols $M[N/x]$, is defined as follows:

$$
\begin{array}{lcll}
x[N/x] & \equiv & N, & \\
y[N/x] & \equiv & y, & \text{if } x \neq y, \\
(MP)[N/x] & \equiv & (M[N/x])(P[N/x]), & \\
(\lambda x.M)[N/x] & \equiv & \lambda x.M, & \\
(\lambda y.M)[N/x] & \equiv & \lambda y.(M[N/x]), & \text{if } x \neq y \text{ and } y \not\in FV(N), \\
(\lambda y.M)[N/x] & \equiv & \lambda y'.(M\{y'/y\}[N/x]), & \text{if } x \neq y,\ y \in FV(N), \text{ and } y' \text{ fresh.}
\end{array}
$$

This definition has one technical flaw: in the last clause, we did not specify which fresh variable to pick, and thus, technically, substitution is not well-defined. One way to solve this problem is to declare all lambda terms to be identified up to α-equivalence, and to prove that substitution is in fact well-defined modulo α-equivalence. Another way would be to specify which variable y' to choose: for instance, assume that there is a well-ordering on the set $\mathcal{V}$ of variables, and stipulate that y' should be chosen to be the least variable which does not occur in either M or N.

2.4 Introduction to β-reduction

Convention. From now on, unless stated otherwise, we identify lambda terms up to α-equivalence. This means, when we speak of lambda terms being "equal", we mean that they are α-equivalent. Formally, we regard lambda terms as equivalence classes modulo α-equivalence. We will often use the ordinary equality symbol $M = N$ to denote α-equivalence.

The process of evaluating lambda terms by "plugging arguments into functions" is called *β-reduction*. A term of the form $(\lambda x.M)N$, which consists of a lambda abstraction applied to another term, is called a *β-redex*. We say that it *reduces* to $M[N/x]$, and we call the latter term the *reduct*. We reduce lambda terms by finding a subterm that is a redex, and then replacing that redex by its reduct. We repeat this as many times as we like, or until there are no more redexes left to reduce. A lambda term without any β-redexes is said to be in *β-normal form*.

For example, the lambda term $(\lambda x.y)((\lambda z.zz)(\lambda w.w))$ can be reduced as follows. Here, we underline each redex just before reducing it:

$$\begin{array}{rcl}(\lambda x.y)(\underline{(\lambda z.zz)(\lambda w.w)}) & \to_\beta & (\lambda x.y)(\underline{(\lambda w.w)(\lambda w.w)}) \\ & \to_\beta & \underline{(\lambda x.y)(\lambda w.w)} \\ & \to_\beta & y.\end{array}$$

The last term, y, has no redexes and is thus in normal form. We could reduce the same term differently, by choosing the redexes in a different order:

$$\underline{(\lambda x.y)((\lambda z.zz)(\lambda w.w))} \quad \to_\beta \quad y.$$

As we can see from this example:

- reducing a redex can create new redexes,
- reducing a redex can delete some other redexes,
- the number of steps that it takes to reach a normal form can vary, depending on the order in which the redexes are reduced.

We can also see that the final result, y, does not seem to depend on the order in which the redexes are reduced. In fact, this is true in general, as we will prove later.

If M and M' are terms such that $M \twoheadrightarrow_\beta M'$, and if M' is in normal form, then we say that M *evaluates* to M'.

Not every term evaluates to something; some terms can be reduces forever without rachine a normal form. The following is an example:

$$\begin{array}{rcl}(\lambda x.xx)(\lambda y.yyy) & \to_\beta & (\lambda y.yyy)(\lambda y.yyy) \\ & \to_\beta & (\lambda y.yyy)(\lambda y.yyy)(\lambda y.yyy) \\ & \to_\beta & \ldots\end{array}$$

This example also shows that the size of a lambda term need not decrease during reduction; it can increase, or remain the same. The term $(\lambda x.xx)(\lambda x.xx)$, which we encountered in Section 1, is another example of a lambda term which does not reach a normal form.

2.5 Formal definitions of β-reduction and β-equivalence

The concept of β-reduction can be defined formally as follows:

Definition. We define *single-step β-reduction* to be the smallest relation $\to_\beta$ on terms satisfying:

$$\begin{array}{ll}(\beta) & \dfrac{}{(\lambda x.M)N \to_\beta M[N/x]} \\ (\mathit{cong}_1) & \dfrac{M \to_\beta M'}{MN \to_\beta M'N} \\ (\mathit{cong}_2) & \dfrac{N \to_\beta N'}{MN \to_\beta MN'} \\ (\xi) & \dfrac{M \to_\beta M'}{\lambda x.M \to_\beta \lambda x.M'}\end{array}$$

Thus, $M \to_\beta M'$ iff M' is obtained from M by reducing a *single* β-redex of M.

Definition. We write $M \twoheadrightarrow_\beta M'$ if M reduces to M' in zero or more steps. Formally, $\twoheadrightarrow_\beta$ is defined to be the reflexive transitive closure of $\to_\beta$, i.e., the smallest reflexive transitive relation containing $\to_\beta$.

Finally, β-equivalence is obtained by allowing reduction steps as well as inverse reduction steps, i.e., by making $\to_\beta$ symmetric:

Definition. We write $M =_\beta M'$ if M can be transformed into M' by zero or more reduction steps and/or inverse reduction steps. Formally, $=_\beta$ is defined to be the reflexive symmetric transitive closure of $\to_\beta$, i.e., the smallest equivalence relation containing $\to_\beta$.

Exercise 2.1. This definition of β-equivalence is slightly different from the one given in class. Prove that they are in fact the same.

3 Programming in the untyped lambda calculus

One of the amazing facts about the untyped lambda calculus is that we can use it to encode data, such as booleans and natural numbers, as well as programs that operate on the data. This can be done purely within the lambda calculus, without adding any additional syntax or axioms.

We will often have occasion to give names to particular lambda terms; we will usually use boldface letters for such names.

3.1 Booleans

We begin by defining two lambda terms to encode the truth values "true" and "false":

$$\begin{array}{rcl} \mathbf{T} & = & \lambda xy.x \\ \mathbf{F} & = & \lambda xy.y \end{array}$$

Let **and** be the term $\lambda ab.ab\mathbf{F}$. Verify the following:

$$\begin{array}{rcl} \textbf{and}\ \mathbf{TT} & \twoheadrightarrow_\beta & \mathbf{T} \\ \textbf{and}\ \mathbf{TF} & \twoheadrightarrow_\beta & \mathbf{F} \\ \textbf{and}\ \mathbf{FT} & \twoheadrightarrow_\beta & \mathbf{F} \\ \textbf{and}\ \mathbf{FF} & \twoheadrightarrow_\beta & \mathbf{F} \end{array}$$

Note that $\mathbf{T}$ and $\mathbf{F}$ are normal forms, so we can really say that a term such as **and** $\mathbf{TT}$ *evaluates* to $\mathbf{T}$. We say that **and** *encodes* the boolean function "and". It is understood that this coding is with respect to the particular coding of "true" and "false". We don't claim that **and** MN evaluates to anything meaningful if M or N are terms other than $\mathbf{T}$ and $\mathbf{F}$.

Incidentially, there is nothing unique about the term $\lambda ab.ab\mathbf{F}$. It is one of many possible ways of encoding the "and" function. Another possibility is $\lambda ab.bab$.

Exercise 3.1. Find lambda terms **or** and **not** which encode the boolean functions "or" and "not". Can you find more than one term?

Moreover, we define the term **if_then_else** $= \lambda x.x$. This term behaves like an "if-then-else" function — specifically, we have

$$\begin{array}{rcl} \textbf{if\_then\_else}\ \mathbf{T}MN & \twoheadrightarrow_\beta & M \\ \textbf{if\_then\_else}\ \mathbf{F}MN & \twoheadrightarrow_\beta & N \end{array}$$

for all lambda terms M, N.

3.2 Natural numbers

If f and x are lambda terms, and $n \geqslant 0$ a natural number, write $f^n x$ for the term $f(f(\ldots(fx)\ldots))$, where f occurs n times. For each natural number n, we define a lambda term $\overline{n}$, called the *nth Church numeral*, as $\overline{n} = \lambda fx.f^n x$. Here are the first few Church numerals:

$$\begin{array}{rcl} \overline{0} & = & \lambda fx.x \\ \overline{1} & = & \lambda fx.fx \\ \overline{2} & = & \lambda fx.f(fx) \\ \overline{3} & = & \lambda fx.f(f(fx)) \\ \ldots & & \end{array}$$

This particular way of encoding the natural numbers is due to Alonzo Church, who was also the inventor of the lambda calculus. Note that $\overline{0}$ is in fact the same term as $\mathbf{F}$; thus, when interpreting a lambda term, we should know ahead of time whether to interpret the result as a boolean or a numeral.

The successor function can be defined as follows: **succ** $= \lambda nfx.f(nfx)$. What does this term compute when applied to a numeral?

$$\begin{array}{rcl} \textbf{succ}\ \overline{n} & = & (\lambda nfx.f(nfx))(\lambda fx.f^n x) \\ & \to_\beta & \lambda fx.f((\lambda fx.f^n x)fx) \\ & \to_\beta & \lambda fx.f(f^n x) \\ & = & \lambda fx.f^{n+1}x \\ & = & \overline{n+1} \end{array}$$

Thus, we have proved that the term **succ** does indeed encode the successor function, when applied to a numeral. Here are possible definitions of addition and multiplication:

$$\begin{array}{rcl} \textbf{add} & = & \lambda nmfx.nf(mfx) \\ \textbf{mult} & = & \lambda nmf.n(mf). \end{array}$$