

Formal Methods / Nachum Dershowitz
 Lecture no.2: Lambda calculus, 7-Mar-00
 Notes by: Boris Rosin

Introduction

λ -notation was created by Alonzo Church in the 1930's, and was designed to emphasize the computational aspects of functions. Let us recall the definition of a function.

Definition 2.11: A function f is a binary relation, $f = \{(x, y) \mid x \in X, y \in Y\}$ which has the following two properties:

- 1) For each $x \in X$ there exists $y \in Y$ such as $(x, y) \in f$.
- 2) If $(x, y) \in f$ and $(x, z) \in f$ then $y = z$.

The first and quite familiar way to look at functions is to examine their graphical representations. With every ordered pair (x, y) in the relation f we will associate a point in the Cartesian plane. This view is believed to be first suggested by Derichlet.

However, as stated earlier, today's lecture will focus on lambda calculus, which looks at functions as computational operators, and examines the way in which they are applied and combined to form other operators. In today's discussion we will not consider typed lambda expressions. Let us then define lambda expressions:

Definition 2.12: A λ -expression (or a λ -term) is defined inductively as follows:

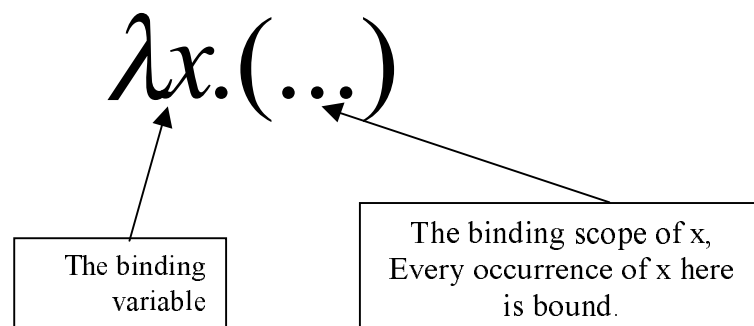
- 1) All variables and constants are λ -terms (called atoms).
- 2) If M and N are λ -terms then (MN) is a λ -term (called an application).
- 3) If M is any λ -term and x is any variable then $(\lambda x.M)$ is a λ -term.

We will need some more basic definitions.

Definition 2.13: In a λ -term of the form $\lambda x.M$ the expression M is called the scope of the occurrence of λ on the left of M . We will also refer to M in a term of such form as the binding scope of the variable x .

Definition 2.14: An occurrence of a variable x in a term P is bound if and only if the occurrence is in the binding scope of x .

Definition 2.15: An occurrence of a variable x in a term P is free if it is not bound.



Definition 2.16: For any λ -terms A, A', B, u, v we will denote the result of substituting every free occurrence of v in A by A' by the following expression:

$$A\{v \mapsto A'\}$$

The precise definition is by induction on A , as follows

$$a) v\{v \mapsto A'\} = A'$$

$$b) u\{v \mapsto A'\} = u$$

For every atom $u \neq v$.

$$c) (AB)\{v \mapsto A'\} = (A\{v \mapsto A'\})(B\{v \mapsto A'\})$$

$$d) \lambda v. B\{v \mapsto A'\} = \lambda v. B$$

$$e) (\lambda u. A)\{v \mapsto A'\} = \lambda u. (A\{v \mapsto A'\})$$

If $u \neq v$ and also u does not appear free in A' or v does not appear free in A .

$$f) (\lambda u. A)\{v \mapsto A'\} = \lambda z. (A\{v \mapsto A'\}\{u \mapsto z\})$$

If $u \neq v$ and also u appears free in A' and v appears free in A . The variable z is chosen to be a variable that does not appear free in both A and A' .

And finally, we will need to define axioms for λ -calculus.

Definition 2.17: The free axioms of non-typed λ -calculus for every A, B, x, y

$$a) \lambda x. A = \lambda y. A\{x \mapsto y\}$$

Provided that y does not appear free in A .

$$\beta) (\lambda x. A[x]) B = A\{x \mapsto B\}$$

$$\eta) \lambda x. B(x) = B$$

Provided that x does not appear free in B .

Properties of λ -calculus

We will use the given above axioms as rewriting rules. Please notice that the α and the η rules are trivial in the sense that their application will affect at most one sub-structure of a λ -term at a time, therefore not changing the structure of the whole term.

Indeed, the most interesting rewriting rule will be β -rule, also called β -reduction rule, since its application involves several sub-expressions of the λ -term. The β -reduction rule is:

$$(\lambda v. A[v]) B \rightarrow A\{v \mapsto B\}$$

For every term A, B and variable v .

We will prove some properties of a rewrite system that includes β -reduction solely, however our conclusions can be easily extended to the whole λ -rewriting system, and we encourage the readers to prove the following properties for α and η rules as well. The use of similar techniques as in our proofs for β will suffice.

Theorem 2.21: The λ rewrite system is Church-Rosser.

Proof:

Let us first prove the following lemma.

Lemma 2.22: The λ rewrite system is confluent.

Proof: Let us remember some definitions and notations from the previous lecture. In the following sub-paragraphs u, s, t and w will denote terms.

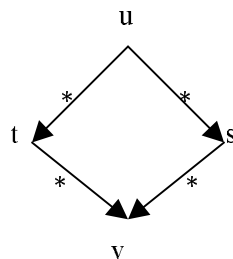
- 1) We said that if s is a result of a single application of a rewrite rule to s then s derives t and t derives from s and denoted this relation as $s \rightarrow t$.
- 2) $s \leftrightarrow t$ if and only if $s \rightarrow t$ or $t \rightarrow s$.
- 3) We defined derivation as an execution of zero or more rewriting rules and denoted the relation as:

$$s \xrightarrow{*} t$$

- 4) s and t were said to be convertible if

$$s \longleftrightarrow t$$

- 5) And finally, a rewrite system is confluent if for every term u if s and t are derived from u then there exists a common v such as v can be derived both from s and from t .



So, why is our system confluent?

Let us look at the structure of the β -reduction rule:

$$(\lambda v. A [v]) B \rightarrow A \{ v \mapsto B \}$$

In every application of the rule on a λ -term of the form $(\lambda v. A)B$ we will distinguish between 3, and only 3, possibilities.

- 1) The rule is applied on the outermost λ , i.e. the result of the application is A after the substitution of all free occurrences of v in it with B .
- 2) The rule is applied to a sub-term of A , thus leaving the general pattern of the whole term the same. If the result of the application in A is A' , then the whole term will have the form $(\lambda v. A')B$
- 3) The rule is applied to a sub-expression of B , thus yielding an expression $(\lambda v. A)B'$, as earlier the general pattern of the whole term remains the same.

Since the different rewriting rules do not overlap, the order of the application of the rules, where the application is possible, does not affect the outcome which is the same. We will show that if we will denote by the ordered pair (1,3) the application of case no.1 and then case no.3 as explained above, we will receive the same outcome as we would have received from applying (3, 1), the other cases can be shown similarly.

And indeed, the order of appliance does not matter, since the substitution of all the free occurrences of v in A with B does not affect B , so it does not matter if we first perform the substitution and then apply β -reduction in B , or otherwise.

Why does this observation help us? Because this is the exact structure of a confluent system. If from a single term u , by applying β -reductions in different places of the term, we can derive two different terms, t and s , then what we have just seen - the fact that the order of appliance of β -reductions is inconsequential, with the help of induction on the number of β -reductions, will yield our claim. We will live the exact formulation of the induction as an exercise to the reader. This completes the poof of lemma 2.12]

In order to complete the proof of Theorem 2.11 it is sufficient to use a claim from the previous lecture, stating that every confluent system is Church-Rosser.

Now we would like to show an additional claim, but first let us remember the following definition from last week:

Definition 2.23: An expression in a rewrite system is said to be in normal form if no rewrite rules of the system can be applied to it.

Claim 2.24: Let A be a λ -term. If A has a normal form N , then it can be derived from A by applying β -reduction each time in the leftmost possible place of the expression.

Proof: Let us assume that A has a normal form N and that it cannot be derived from A by applying β reduction at the leftmost possible place in A and reach a contradiction. Since, according to our assumption, A has a normal form N , there must exist a sequence of applications of β -reductions on A , which yields N . Let B be the result of the application of the first step of this sequence. We will want to show that if our assumption is true, and an application of the leftmost β -reduction from A does not yield N , then the application of the leftmost β -reduction from B also does not yield N . We will prove it by the following: we claim that if our assumption is true then the number of applications of β -reduction in the leftmost possible place in B is not less then the number of such applications in A .

This we will prove by induction on the number of leftmost β -reduction applications from A . Let us denote by C the result of the first such application. We will denote this situation by an ordered triple (A, B, C) .

$$\begin{array}{c} A \xrightarrow{l} C \\ \downarrow \\ B \end{array}$$

Where

$$\xrightarrow{=} \text{ and } \xrightarrow{l}$$

Mean the application of possibly several β -reductions simultaneously (called a parallel reduction step), and the application of the leftmost possible β -reduction, respectively. We will distinguish between 2 possibilities.

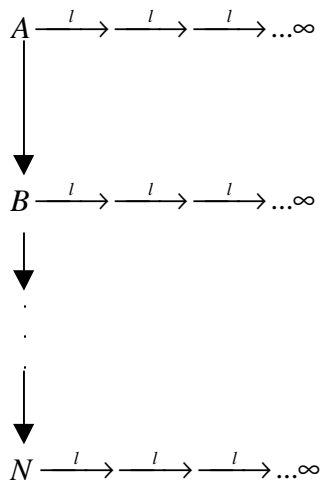
1) $C = B$ and then since the application of the leftmost step from A will not yield N so will not the application of the leftmost steps from C , and therefore also from B since B and C are the same.

2) $C \neq B$. In the proof of lemma 2.22 we have shown that the order of application of β -reduction is inconsequential, hence there exists a leftmost step from A and a parallel step from B that will yield the same expression, which we have denoted by D .

Now, to complete the induction, let us look at the first $n-1$ leftmost steps from A . The induction hypothesis will yield us $n-1$ leftmost steps from B , let us denote the result of their application as Y . We will also denote the result of the application of $n-1$ leftmost possible steps from A as X . According to our assumption there is an infinite number of leftmost reduction steps possible from A , otherwise if the number was finite then A would have a normal form, therefore let us denote the result of the application of the leftmost reduction on X as Z . We now have the situation that we have denoted earlier by an ordered triple (X, Y, Z) and the proof of the claim is the same as in the basis of the induction.

This proves that the number of leftmost reduction steps applicable to B is not less than the number of such steps applicable to A , which according to our assumption is ∞ and therefore the number of such steps applicable to B is also ∞ .

Now, a simple use of induction will yield that for every expression, which is a result of the application of some steps of the sequence that yields N when applied to A , the number of the leftmost applicable β -reductions is ∞ . But, since N is also a part of this sequence, we have reached the desired contradiction -- no β -reductions are applicable to N whatsoever, it is in normal form.



Applications

What can be expressed with λ -calculus? Perhaps one of the most interesting applications is the use of it to assess the power of computational models. If we will succeed in proving that in a certain model we can express λ -calculus then we are actually saying quite a lot. And here is why: every computer program is expressible in λ -calculus. We will try to show that most common operations to all computer languages can be expressed in the language we just have built.

We will start from the representation of Boolean values.

Let us define $\lambda u(\lambda v.u) = T$ the truth value True, and $\lambda u(\lambda v.v) = F$ the truth value False.

These are arbitrary definitions, axioms if you will, on which we are going to build our computational language. Since computer programs usually compute numerical results, we will want to express natural numbers in our language. Let us first define $0 = \lambda x.x$ and the rest of the numbers will be defined by applying the sequential function on 0, which we will show shortly.

Ordered pairs $\langle y, z \rangle$ will be represented as following: $\lambda u.((u y) z)$. Now, if we will want to obtain the first element of an ordered pair, we will just apply $\langle y, z \rangle(T)$. Let us show that we will indeed get the first element of the pair:

$$\lambda u.((u y) z) (\lambda u.(\lambda v.u)) \rightarrow ((\lambda u.(\lambda v.u) (y)) z) \rightarrow \lambda v.y (z) \rightarrow y$$

Now we are equipped to define the sequential function: given an number x we will define $x+1$ as an ordered pair $\langle F, x \rangle$. Now if we want to know whether $x = 0$ we will just evaluate the expression $x(T)$. If $x \neq 0$ then, since x is an ordered pair with F as it's first element, the evaluation will return F , as we have previously shown. If, on the other hand, $x = 0$ then it is simply the identity function and thus $x(T)$ will return T .

Our previously chosen definitions will also allow us to introduce conditional expressions into our language. An expression of the form (if x then y else z) will in our language be $((x y) z)$. We will leave it as an exercise to the reader to show that the evaluation of this expression gives the desired result.

This is quite a progress, however there still a very important element missing in our language. A lot of basic numeric operations can be defined by recursion, for example the multiplication operation could be defined as such: $a * b := ((a - 1) * b) + a$. Note that we already know how to subtract 1 from numbers, we will just apply $x(F)$ to receive $x - 1$. Addition can also be expressed by recursion: $a + b := (a + 1) + (b - 1)$.

So, let us try to express recursion. We will introduce this notation for the simplicity of the text. $W := \lambda x.f(x(x))$, $Y := \lambda f.W(W)$ where f is the function that we want to apply recursively. Lets see what happens when we evaluate $Y(f)$.

$$Y(f) \rightarrow \lambda f.W(W)(f) \rightarrow \lambda x.f(x(x)) (\lambda x.f(x(x))) \rightarrow f(\lambda x.f(x(x)) (\lambda x.f(x(x)))) = f(W(W) (f)) = f(Y(f)).$$

Example: We want to define addition operation with the recursion we just have built. First we will introduce the following notation: $\lambda x.\lambda y.A[x y] = \lambda x y. A[x y]$, where $A[x y]$ means that x and y appear free in A . We will define our f to be

$$f := \lambda x.(\lambda y z.(if y = 0 then z else if z = 0 then y else Y(x) (y + 1, z - 1)))$$

Let us demonstrate that the recursive application of f on two arbitrary numbers, a and b , will yield $a + b$.

$$Y(f) (a b) \rightarrow f(Y(f)) (a b) \rightarrow \lambda y z.(if (y = 0) then z else if (z = 0) then y else Y(f) (y +1, z-1)) (a b) \rightarrow if (a = 0) then b else if (b = 0) then a else Y(f) (a + 1, b - 1)$$

This is indeed the recursive definition of addition.

Exercise: Formulate the multiplication operator in our language.

On the next page a table with all the basic operations is given for your convenience.

Meaning	The definition
TRUE	$\lambda u(\lambda v.u)$
FALSE	$\lambda u(\lambda v.v)$
If x then y else z	$((x\ y)\ z)$
$\langle x, y \rangle$	$\lambda u.(if\ u\ then\ y\ else\ z)$
* Car ($\langle x, y \rangle$)	$\langle x, y \rangle(T)$
Cdr($\langle x, y \rangle$)	$\langle x, y \rangle(F)$
0	$\lambda u.u$
$x + 1$	$\langle F, x \rangle$
$x = 0$	$x(T)$
Recursive application of f	$Y(f)$, where Y is as defined previously

* Car and Cdr stand for first and the second element of an ordered pair, respectively.