Lecture Notes on λ-Calculus: Normal Forms and Booleans

15-814: Types and Programming Languages
Frank Pfenning and Jan Hoffmann

Lecture 2
Thursday, August 27, 2026

1 Introduction

In this lecture, we continue our exploration of the λ-calculus and the representation of data and functions on them. We first discuss nontermination and normal forms. We then discuss ways of representing and manipulate Booleans in the λ-calculus.

2 Nontermination

Consider again the λ-calculus as introduced in Lecture 1. Recall that *normal forms* are expressions that cannot be reduced by β-reduction. We say that e is a normal form of e' if e is a normal form and $e =_\beta e'$. Here, we talk about β-equality only since we consider α-equality (renaming of bound variables) to be implicitly built-in: From now on, we silently identify expressions that are α-equivalent.

It is natural to consider the following questions.

1. Does every expression have a normal form?
2. Can we always compute a normal form if one exists?
3. Are normal forms unique?

The answers to these questions are crucial to understanding to what extent we might consider the λ-calculus a universal model of computation.

Does every expression have a normal form?

If the λ-calculus is to be equivalent in computational power to Turing machines in some way, then we would expect the answer to be "no" because computations of Turing machines may not halt. However, it is not immediate to think of some expression that doesn't have a normal form. If you haven't seen something like this already, you might want to try to come up with one. The simplest one is probably

$$\Omega \triangleq (\lambda x.\, x\, x)\, (\lambda x.\, x\, x)$$

Indeed, there is only one possible β-reduction and it immediately leads to exactly the same term:

$$\begin{array}{lcl} \Omega & = & (\lambda x.\, x\, x)\, (\lambda x.\, x\, x) \\ & \longmapsto_\beta & (\lambda x.\, x\, x)\, (\lambda x.\, x\, x) \\ & \longmapsto_\beta & (\lambda x.\, x\, x)\, (\lambda x.\, x\, x) \\ & \longmapsto_\beta & \ldots \end{array}$$

So Ω reduces in one step to itself and only to itself.

Can we always compute a normal form if one exists?

The answer here is "yes", although it is not easy to prove that this is the case. Let's consider an example (recall that $K = \lambda x.\, \lambda y.\, x$):

$$K\, I\, \Omega \longmapsto_\beta (\lambda y.\, I)\, \Omega \longmapsto_\beta I$$

So the expression $K\, I\, \Omega$ does have a normal form, even though Ω does not. This is because the constant function $K\, I$ ignores its argument. On the other hand we also have

$$K\, I\, \Omega \longmapsto_\beta K\, I\, \Omega \longmapsto_\beta K\, I\, \Omega \longmapsto_\beta \cdots$$

because we have the $\Omega \longmapsto_\beta \Omega$ and reduction can be applied anywhere in an expression.

Fortunately, there is a strategy which turns out to be complete in the sense that if an expression has a normal form, this strategy will find it. It is called *leftmost-outermost* or *normal-order reduction*. This strategy scans through the expression from left to right and when it find a *redex* (that is, an expression of the form $(\lambda x.\, e)\, e'$) it applies β-reduction and then returns to the beginning of the result expression. In particular, it does

not consider any redex in e or e', only the "outermost" one. Also, in an expression $((\lambda x.\, e_1)\, e_2)\, e_3$ it does not consider any potential redex in e_3, only the leftmost one.

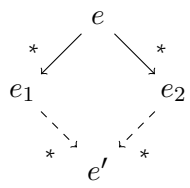
This strategy works in our example: the redex in Ω would not be considered, only the redex $K\, I$ and then the redex $(\lambda y.\, I)\, \Omega$.

In this course we are using LAMBDA, an implementation of the λ-calculus. This implementation of uses a straightforward function for leftmost-outermost reduction, complicated very slightly by the fact that names such as K or I which in the notes are only abbreviations at the mathematical level of discourse, are actual language-level definitions in the implementation. So we have to expand the definition of K, for example, before applying β-reduction, but we do not officially count this as a substitution.

The notion of leftmost-outermost reduction is closely related to the notion of call-by-name evaluation in programming languages (and, with a little more distance, to call-by-need which is employed in Haskell). In contrast, call-by-value would reduce the argument of a function before applying the β-reduction, which is not complete, as our example shows. The analogy is not exact, however, since in programming languages such as ML or Haskell we also do not reduce under λ-abstractions, a fact that represents a sharp dividing line between foundational calculi such as the λ-calculus and actual programming languages. We will justify and understand these decisions in a few lectures.

Are normal forms unique?

The outcome of a computation starting from e is its normal form. At any point during a computation there may be many redices. Ideally, the outcome would be independent of the reduction strategy we choose as long as we reach a normal form. Otherwise, the meaning of an expression (as represented by its normal form) may be ambiguous. Therefore, Church and Rosser [CR36] spend considerable effort in proving the uniqueness of normal forms. The key technical device is a property called *confluence* (also referred to as the *Church-Rosser property*). It is often depicted in the following diagram:



More formally:

Theorem 1 (Church-Rosser) *Let e be an expression. If $e \longmapsto^*_\beta e_1$ and $e \longmapsto^*_\beta e_2$ for some expressions e_1 and e_2 then there exists an expression e' such that $e_1 \longmapsto^*_\beta e'$ and $e_2 \longmapsto^*_\beta e'$.*

The solid lines are given reduction sequences while the reduction sequences represented by dashed lines have to be shown to exist. The reduction relation $\longmapsto^*_\beta$ refers to multiple (possibly zero) steps of β-reduction. Very roughly, the proof shows how to simulate the steps from e to e_2 when starting from e_1 and (symmetrically) simulate the steps from e to e_1 when starting from e_2.

Confluence implies the uniqueness of normal forms. Suppose e_1 and e_2 in the diagram are normal forms. Because they cannot be reduced further, the sequence of reductions to e' must consist of zero steps, so $e_1 = e' = e_2$.

Confluence implies that even though we might embark on an unfortunate path (for example, keep reducing Ω in $K\,I\,\Omega$) we can still recover if indeed there is a normal form. In this example, we might eventually decide to reduce $K\,I$ and then the redex $(\lambda y.\, I)\,\Omega$.

3 Representing Booleans

Before we can claim the λ-calculus as a universal language for computation, we need to be able to represent *data*. The simplest nontrivial data type are the Booleans, a type with two elements: $\overline{\mathit{true}}$ and $\overline{\mathit{false}}$. The general technique is to represent the values of a given type by *normal forms*, that is, expressions that cannot be reduced by β-reduction. Furthermore, they should be *closed*, that is, not contain any free variables. We need to be able to distinguish between two values, and in a closed expression that suggest introducing two bound variables. We then define rather arbitrarily one to be $\overline{\mathit{true}}$ and the other to be $\overline{\mathit{false}}$

$$\begin{array}{rcl} \overline{\mathit{true}} & \triangleq & \lambda x.\, \lambda y.\, x \\ \overline{\mathit{false}} & \triangleq & \lambda x.\, \lambda y.\, y \end{array}$$

The next step is to define *functions* on values of the type. Let's start with negation: we are trying to define a λ-expression *not* such that

$$\begin{array}{rcl} \mathit{not}\ \overline{\mathit{true}} & =_\beta & \overline{\mathit{false}} \\ \mathit{not}\ \overline{\mathit{false}} & =_\beta & \overline{\mathit{true}} \end{array}$$

We start with the obvious:

$$\mathit{not} \triangleq \lambda b.\, \ldots$$

Now there are two possibilities: we could either try to apply b to some arguments, or we could build some λ-abstractions. Let's first try the one where b is applied to some arguments.

$$\mathit{not} \triangleq \lambda b.\, b\,(\ldots)\,(\ldots)$$

We suggest two arguments to b, because b stands for a Boolean, and Booleans $\overline{\mathit{true}}$ and $\overline{\mathit{false}}$ both take two arguments. $\overline{\mathit{true}} = \lambda x.\, \lambda y.\, x$ will pick out the first of these two arguments and discard the second, so since we specified $\mathit{not}\,\overline{\mathit{true}} = \overline{\mathit{false}}$, the first argument to b should be $\overline{\mathit{false}}$!

$$\mathit{not} \triangleq \lambda b.\, b\,\overline{\mathit{false}}\,(\ldots)$$

Since $\overline{\mathit{false}} = \lambda x.\, \lambda y.\, y$ picks out the second argument and $\mathit{not}\,\overline{\mathit{false}} = \overline{\mathit{true}}$, the second argument to b should be $\overline{\mathit{true}}$.

$$\mathit{not} \triangleq \lambda b.\, b\,\overline{\mathit{false}}\,\overline{\mathit{true}}$$

Now it is a simple matter to calculate that the computation of *not* applied to $\overline{\mathit{true}}$ or $\overline{\mathit{false}}$ completes in three steps and obtain the correct result.

$$\begin{array}{lcl} \mathit{not}\,\overline{\mathit{true}} & \longmapsto_\beta^3 & \overline{\mathit{false}} \\ \mathit{not}\,\overline{\mathit{false}} & \longmapsto_\beta^3 & \overline{\mathit{true}} \end{array}$$

We write $\longmapsto_\beta^n$ for reduction in n steps, and $\longmapsto_\beta^*$ for reduction in an arbitrary number of steps, including zero steps. In other words, $\longmapsto_\beta^*$ is the reflexive and transitive closure of $\longmapsto_\beta$.

An alternative solution hinted at above is to start with

$$\mathit{not}' \triangleq \lambda b.\, \lambda x.\, \lambda y.\, \ldots$$

We pose this because the result of $\mathit{not}\,b$ should be a Boolean, and the two Booleans both start with two λ-abstractions. Now we reuse the previous idea, but apply b not to $\overline{\mathit{false}}$ and $\overline{\mathit{true}}$, but to y and x.

$$\mathit{not}' \triangleq \lambda b.\, \lambda x.\, \lambda y.\, b\,y\,x$$

Again, we calculate

$$\begin{array}{lcl} \mathit{not}'\,\overline{\mathit{true}} & \longmapsto_\beta^3 & \overline{\mathit{false}} \\ \mathit{not}'\,\overline{\mathit{false}} & \longmapsto_\beta^3 & \overline{\mathit{true}} \end{array}$$

An important observation here is that

$$\mathit{not} = \lambda b.\, b\,(\lambda x.\, \lambda y.\, y)\,(\lambda x.\, \lambda y.\, x) \neq \lambda b.\, \lambda x.\, \lambda y.\, b\,y\,x = \mathit{not}'$$

Both of these are *normal forms* (they cannot be reduced) and therefore represent *values* (the results of computations). Both correctly implement negation on Booleans, but they are *different*. This is evidence that when computing with particular data representations in the λ-calculus it is *not extensional*: even though the functions behave the same on all the arguments we care about (here just $\overline{\mathit{true}}$ and $\overline{\mathit{false}}$), the are not convertible. To actually see that they are not convertible we need the Church-Rosser theorem which says if e_1 and e_2 are $\alpha\beta$-convertible then there is a common reduct e such that $e_1 \longmapsto_\beta^* e$ and $e_2 \longmapsto_\beta^* e$.

There are many possible representations of the Booleans in the λ-calculus. The one we discussed is called the Church encoding of the Booleans. It is the canonical encoding because it corresponds to the elimination form for Booleans: the conditional or if-then-else. Consider the function *not* again:

$$\lambda b.\, b\, \overline{\mathit{false}}\, \overline{\mathit{true}}$$

Now compare *not* with an implementation of the function using a conditional:

$$\lambda b.\, \mathit{if}\ b\ \mathit{then}\ \overline{\mathit{false}}\ \mathit{else}\ \overline{\mathit{true}}$$

If we remove the keywords *if*, *them*, and *else* from this definition then we obtain our function *not* again. This is not a coincidence since our Booleans behave exactly like a conditional. A Boolean b applied to two λ-expressions e_1 and e_2 will "branch" and β-reduce to e_1 if $b =_\beta \overline{\mathit{true}}$ or to e_2 if $b =_\beta \overline{\mathit{false}}$. We therefore define

$$\mathit{if} \triangleq \lambda b.\, \lambda x.\, \lambda y.\, b\, x\, y$$

Exercises

Exercise 1 Define the following functions on Booleans in at least two distinct ways.

1. "*nor*", the negation of disjunction

2. The conditional "*if*" such that

$$\begin{array}{lcl} \mathit{if}\ \overline{\mathit{true}}\ e_1\ e_2 & =_\beta & e_1 \\ \mathit{if}\ \overline{\mathit{false}}\ e_1\ e_2 & =_\beta & e_2 \end{array}$$

References

[CR36] Alonzo Church and J.B. Rosser. Some properties of conversion. *Transactions of the American Mathematical Society*, 39(3):472–482, May 1936.