

Handout 7: Call by Name and Call by Value

In Handout 4, we have examined the reduction semantics of lambda calculus, and noted the difference between the **normal order** reduction, which chooses leftmost-outermost redexes for reduction at every stage, and the **applicative order** reduction, which chooses innermost redexes for reduction at every stage. We noted that the normal order reduction strategy is always able to find the normal form of a lambda calculus term, but the applicative order strategy may fail to find the normal form by carrying out reductions that are “unnecessary” for the normal form.

In programming languages, the evaluation order that corresponds to normal order reduction is termed **call by name**. The evaluation order that corresponds to the applicative order reduction is termed **call-by-value**.

Algol 60 was the first programming language to use the call-by-name evaluation order. However, at this early stage in the evolution of programming languages, the implications of call-by-name were not fully understood and it is said that many implementations of Algol 60 failed to implement it correctly. Many of the Algol derivatives, such as Simula 67, also used the call-by-name evaluation mechanism. Pascal and C, which were also derivatives of Algol, chose to use the call by value evaluation order because it was simpler and it was also felt that it was a more efficient execution mechanism.

In functional programming, Miranda and Haskell use call-by-name evaluation order. (They actually use a variant of call-by-name order called **call by need**.) Other languages, including Lisp, Pop-11 and ML use the call-by-value evaluation order.

In this handout, we look at the two evaluation mechanisms more closely and formalize their behaviour.

1. Call by name, abstractly

The call-by-name evaluation mechanism can be characterised very simply as the use of normal order reduction (i.e., the choice of leftmost-outermost redexes), with two provisos:

- We always attempt to reduce *closed terms*, i.e., terms without any free variables. This is reasonable because we are dealing with programming languages and programs are expected to be self-contained.
- We never reduce inside lambda abstraction terms. If a lambda abstraction term $\lambda x. M$ has some redexes inside M , we leave them alone. This is reasonable because, in programming, we think of $\lambda x. M$ as a block-box function which only does some work when we “call” it, i.e., apply it to arguments.

2. Call by value, abstractly

The call-by-value evaluation mechanism can be characterized as the use of applicative order reduction (i.e., the choice of innermost redexes), with the same two provisos:

- We always attempt to reduce closed terms.
- We never reduce inside lambda abstraction terms.

3. Example

Consider the term

$$\text{and } (\text{not } (\text{null } l)) \ ((\text{car } l) > 0)$$

We would like to evaluate it in a context where the variables have the values:

$$\begin{aligned} \text{and} &= \lambda p. \lambda q. \text{if } p \ q \ \text{false} \\ l &= (\text{list } 2 \ 3 \ 4) \end{aligned}$$

The call-by-name reduction proceeds as follows:

$$\begin{aligned} [\text{and } (\text{not } (\text{null } l)) \ ((\text{car } l) > 0)] &\xrightarrow{*}_{\beta} \text{if } (\text{not } (\text{null } l)) \ ((\text{car } l) > 0) \ \text{false} \\ &= \text{if } [(\text{not } (\text{null } l))] \ ((\text{car } l) > 0) \ \text{false} \\ &\xrightarrow{*}_{\beta, \delta} \text{if } \text{true} \ ((\text{car } l) > 0) \ \text{false} \\ &= [\text{if } \text{true} \ ((\text{car } l) > 0) \ \text{false}] \\ &\xrightarrow{\delta} ((\text{car } l) > 0) \\ &= [((\text{car } l) > 0)] \\ &\xrightarrow{*}_{\beta, \delta} \text{true} \end{aligned}$$

At each stage, we identify a redex, notate it in *square brackets* [...], and then carry out one or more steps of reduction on it using β - and δ -rules. After this, another redex is identified and so on.

The main point is that the two arguments (*not* (null *l*)) and ((*car l*) > 0) are passed to the function “*and*” without advance evaluation. The *and* function is free to evaluate them as the need arises.

On the other hand, the call-by-value reduction proceeds as follows:

$$\begin{aligned}
 \text{and } [(\text{not } (\text{null } l))] ((\text{car } l) > 0) &\longrightarrow_{\beta, \delta}^* \text{and true } ((\text{car } l) > 0) \\
 &= \text{and true } [(\text{car } l) > 0] \\
 &\longrightarrow_{\beta, \delta}^* \text{and true true} \\
 &= [\text{and true true}] \\
 &\longrightarrow_{\beta}^* \text{if true true false} \\
 &= [\text{if true true false}] \\
 &\longrightarrow_{\delta} \text{true}
 \end{aligned}$$

The important point to notice here is that the two arguments (*not* (null *l*)) and ((*car l*) > 0) of *and* are evaluated *ahead of time* and their values are passed to the function *and*.

So far, this looks unexceptionable. call-by-value evaluates the arguments ahead of time, but the values of these arguments would be the same independent of when they are evaluated. So, what is the big difference?

To see this, consider evaluating the same term in a context where *l* = nil. The call-by-name evaluation proceeds in much the same way:

$$\begin{aligned}
 [\text{and } (\text{not } (\text{null } l)) ((\text{car } l) > 0)] &\longrightarrow_{\beta}^* \text{if } [(\text{not } (\text{null } l))] ((\text{car } l) > 0) \text{ false} \\
 &\longrightarrow_{\beta, \delta}^* [\text{if false } ((\text{car } l) > 0) \text{ false}] \\
 &\longrightarrow_{\delta} \text{false}
 \end{aligned}$$

However, the call-by-value evaluation gives rise to an error:

$$\begin{aligned}
 \text{and } [(\text{not } (\text{null } l))] ((\text{car } l) > 0) &\longrightarrow_{\beta, \delta}^* \text{and false } [(\text{car } l) > 0] \\
 &\longrightarrow_{\beta, \delta}^* \text{and false } \mathbf{error}
 \end{aligned}$$

Because, *car l* is undefined when *l* = nil, evaluating the second argument gives an error. The call-by-name evaluation neatly avoids the problem, by neglecting to evaluate the second argument.

4. Example with infinite data structures

Consider the recursive definition:

$$\text{ints } n = \text{cons } n (\text{ints } (n + 1))$$

Conceptually, the value of an application such as *ints 2* is the infinite list (list 2 3 4 ...). However, call-by-name evaluation can cleverly avoid having to compute all of the infinite structure. It computes just as much of it as necessary in a context.

Recall that there are no δ reductions for applications of cons. We only have δ reductions for applications of *car* and *cdr* to cons-terms. So, a call-by-name evaluator does no evaluation when presented with a term of the form (cons *M N*).

Here is a sample evaluation in the call-by-name system:

$$\begin{aligned}
 \text{car } (\text{cdr } [(\text{ints } 2)]) &\longrightarrow_{\beta} \text{car } (\text{cdr } (\text{cons } 2 (\text{ints } (2 + 1)))) \\
 &= \text{car } [(\text{cdr } (\text{cons } 2 (\text{ints } (2 + 1))))] \\
 &\longrightarrow_{\delta} \text{car } (\text{ints } (2 + 1)) \\
 &= \text{car } [\text{ints } (2 + 1)] \\
 &\longrightarrow_{\beta} \text{car } (\text{cons } (2 + 1) (\text{ints } (2 + 1 + 1))) \\
 &= [\text{car } (\text{cons } (2 + 1) (\text{ints } (2 + 1 + 1)))] \\
 &\longrightarrow_{\delta} [2 + 1] \\
 &\longrightarrow_{\delta} 3
 \end{aligned}$$

In contrast, call-by-value evaluation attempts to evaluate the argument (*ints 2*) fully, before applying *cdr* to it. Unfortunately, this evaluation goes on for ever.

5. Call by name, in detail

Next we look at the evaluation mechanism of call by name in detail. The basic question is, *given a lambda term M, which one of its subterms can be reduced under the call-by-name discipline?*

In pure lambda calculus, there are three kinds of terms: variables *x*, lambda abstractions $\lambda x. M$ and function applications $M_1 M_2$. Of these, variables are not closed terms. So, the evaluator can never be presented with a

simple variable. (There are of course variables used in terms. But, beta-reduction substitutes them with argument terms before the evaluator gets around to evaluating them.) For the remaining two cases, we use the following strategies:

- The evaluation of $\lambda x. M$ does nothing. (Recall that we never reduce inside lambda abstractions.) Call this the *DONE* rule.
- The evaluation of $(M_1 M_2)$ involves evaluating M_1 (if it needs further evaluation). Call this the *GO LEFT* rule.
- If we are to evaluate $(M_1 M_2)$ and M_1 is already evaluated, i.e., if it is a lambda abstraction, then the evaluator does a beta-reduction for $M_1 M_2$. Call this the *BETA* rule.

That is all there is to evaluating pure lambda calculus terms.

In applied lambda calculus, we have, in addition, the various constants. The data constants, like integers, need no evaluation. The function constants can be divided into two groups:

- All the arithmetic and relational operators, and *car*, *cdr* and *form* form one group. They have the property that they need the values of all their arguments in order to apply a δ -reduction. These are called **strict** operations.
- In the second group, we have *if* and *cons*. These are called **non-strict** operations.

Here are the strategies for the constants:

- The evaluation of a constant c involves doing nothing. Call this the *CONST* rule.
- The evaluation of $(\mathbf{op} M_1 \dots M_n)$ where **op** is a strict operation involves evaluating every one of the argument terms (if they need evaluation). Call this the *FORCED* rule.
- The evaluation of $(\mathbf{if} M N_1 N_2)$ involves evaluating M (if it needs evaluation). Call this the *COND* rule.
- The evaluation of $(\mathbf{cons} M_1 M_2)$ involves nothing. Call this the *LAZY* rule.
- Finally, to evaluate an application of a function constant $(c M_1 \dots M_n)$ which is a δ -redex, the appropriate δ -reduction should be applied. Call this the *DELTA* rule.

As you can see, the *BETA* and *DELTA* rules involve doing reduction. The remaining rules basically tell us how to search for the right redex under the call-by-name discipline.

6. Examples, revisited

Let us check if how these rules apply to a term such as:

$$\mathit{and} \ (\mathit{not} \ (\mathit{null} \ l)) \ ((\mathit{car} \ l) > 0)$$

Remember that the term should be bracketed as $(\mathit{and} \ (\mathit{not} \ (\mathit{null} \ l))) \ ((\mathit{car} \ l) > 0)$ and that *and* stands for $\lambda p. \lambda q. \mathit{if} \ p \ q \ \mathit{false}$.

For the outermost application, we first use the *GO LEFT* rule. That takes us to $(\mathit{and} \ (\mathit{not} \ (\mathit{null} \ l)))$. Since *and* is a lambda abstraction, we have a β -redex and we use the *BETA* rule. Once this is done, we have $(\lambda q. \mathit{if} \ (\mathit{not} \ (\mathit{null} \ l)) \ q \ \mathit{false})$. Now, we have a β -redex at the top level, and we use *BETA* rule again. Doing a β -reduction at the top gives $\mathit{if} \ (\mathit{not} \ (\mathit{null} \ l)) \ ((\mathit{car} \ l) > 0) \ \mathit{false}$

Consider the second example:

$$\mathit{car} \ (\mathit{cdr} \ (\mathit{ints} \ 2))$$

Again, remember that *ints* stands for a lambda abstraction $\lambda n. \dots$. We first use *GO LEFT* rule. That takes us to *car*, but that is just a constant, in fact a strict operation. Since it is a strict operation, we apply the *FORCED* rule. So, we need to evaluate $(\mathit{cdr} \ (\mathit{ints} \ 2))$. Again, *GO LEFT* takes us to a constant, *cdr*, which is again a strict operation and we end up with $(\mathit{ints} \ 2)$ via the *FORCED* rule again. Now, we have a β -redex, which we proceed to reduce.

7. Reduction contexts There is succinct way of describing how to identify the redexes that should be reduced under the call-by-name discipline. At each step of the reduction process, we identify a redex M , and the redex in a surrounding *context* of the overall term. The reduction process can be described by specifying what kinds of contexts can surround the redex. Notice this for the two examples considered above:

$$\begin{aligned} \mathit{and} \ (\mathit{not} \ (\mathit{null} \ l)) \ ((\mathit{car} \ l) > 0) &= [\mathit{and} \ (\mathit{not} \ (\mathit{null} \ l))] \ ((\mathit{car} \ l) > 0) \\ \mathit{car} \ (\mathit{cdr} \ (\mathit{ints} \ 2)) &= \mathit{car} \ (\mathit{cdr} \ [\mathit{ints} \ 2]) \end{aligned}$$

In the first case, *and* (*not* (*null l*)) is the chosen redex, and $[]$ ($(\text{car } l) > 0$) is the surrounding context.

In the second case, *ints* 2 is the chosen redex, and $\text{car } (\text{cdr } [])$ is the surrounding context.

A *call-by-name reduction context* is a surrounding context that is allowed for a redex under the call-by-name discipline. Let us analyze our rules, using this idea.

- The *BETA* rule (as well as the *DELTA* rule) say that the empty context $[]$ is a reduction context.
- The *GO LEFT* rule says that, if \mathcal{N} is a reduction context, then $\mathcal{N} M_2$ is a reduction context, for any term M_2 .
- The *FORCED* rule says that, if **op** is a strict operation and \mathcal{N} is a reduction context, then $(\text{op } M_1 \dots \mathcal{N} \dots M_n)$ is a reduction context (because *every one* of the arguments can be evaluated).
- The *COND* rule says that, if \mathcal{N} is a reduction context, then $(\text{if } N)$ is a reduction context. We don't need to worry about the then- and else-branches of *if* because they are already considered in the *GO LEFT* rule.
- The *DONE*, *CONST* and *LAZY* rules don't contribute any reduction contexts.

Gathering all this information, we can describe the call-by-name reduction contexts by a succinct grammar:

$$\mathcal{N} ::= [] \mid (\mathcal{N}_1 M_2) \mid (\text{op } M_1 \dots \mathcal{N}_i \dots M_n) \mid (\text{if } \mathcal{N})$$

To answer the question that was posed in paragraph 5, “*which one of the subterms can be reduced under the call-by-name discipline?*”, the answer is as follows. Put the given term M in the form $\mathcal{N}[M']$ where \mathcal{N} is a reduction context by the above grammar. Then the subterm M' can be reduced.

8. Value terms for call-by-name

Whereas in lambda calculus, we speak of normal forms which cannot be reduced any further, the corresponding idea in programming languages is that of **value terms**. A call-by-name value term is one that cannot be reduced further under the call-by-name discipline. More briefly, a value term is one that does not have a call-by-name reduction context. Such value terms can be described by the following grammar:

$$V ::= \lambda x. M' \mid c \mid (\text{cons } M_1 M_2) \mid (\text{op } V_1 \dots V_k) \mid (\text{if } V_1 M_2) \mid (\text{cons } M_1) \mid$$

where k is less than the arity of **op**, i.e., the number of arguments needed to form a δ -redex. For example, the arity of $+$ is 2. So, a term of the form $(+ V)$ cannot be reduced. So, it is regarded as a value term.

The fact that $\lambda x. M'$, c and $(\text{cons } M_1 M_2)$ are value terms was declared in the *DONE*, *CONST* and *LAZY* rules. The remaining cases are bureaucratic. They are function applications that do not have enough arguments, and, so cannot be reduced.

9. Call by value, in detail

For the call-by-value discipline, we again ask a similar question: *given a lambda term M , which one of its subterms can be reduced under the call-by-value discipline?*

The answer is quite straightforward. For the pure lambda calculus part of the language, we add *GO RIGHT* rule:

- The evaluation of $(M_1 M_2)$ involves evaluating M_2 (if it needs further evaluation). Since M_1 also needs to be evaluated (*GO LEFT* rule), we choose to constrain the choice and say, we evaluate M_2 only if M_1 has already been reduced to a value term.

We do not need any separate rules for function constants, since they are treated the same way as the user-defined functions. So, we delete the *FORCED* rule, *COND* rule and the *LAZY* rule.

The resultant grammar for the call-by-value reduction contexts is as follows:

$$\mathcal{R} ::= [] \mid (\mathcal{R}_1 M_2) \mid (V_1 \mathcal{R}_2) \mid (c V_1 \dots \mathcal{R}_i \dots M_n)$$

The third case has been contributed by the *GO RIGHT* above.

The value terms for call-by-value are similar to those of call-by-name:

$$V ::= \lambda x. M' \mid c \mid (c V_1 \dots V_k)$$

where k is less than the arity of the function constant c .