Reduction Strategies for Lambda Calculus

Norman Ramsey

Spring 2019

1 Introduction

If our notion of operational semantics for lambda calculus is "reduce the term until it reaches normal form," we have a little problem: the semantics is nondeterministic. In general, nondeterminism can be simulated by evolving a *set* that includes all possible derivations. But in the case of lambda calculus, we prefer to explore just a single derivation at a time. The algorithm used to select a single derivation among multiple possibilities is called a *reduction strategy*.

Reduction strategies are studied primarily for insight into operational semantics of real languages. *Applicative-order reduction* models typical evaluation rules, which are found in "eager" or "strict" languages like Algol, C, and ML. *Normal-order reduction* models atypical evaluation rules, which are found in "lazy" languages like Clean and Haskell (as well as the legacy feature "call by name," which can found in a few imperative languages). This handout

- Presents reduction rules for untyped lambda calculus
- Explains normal-order reduction and applicative-order reduction
- Sketches what's interesting or useful about each rule or strategy

2 Reduction relations

The operational semantics of the untyped lambda calculus is given by two relations:

$M \to M'$	M reduces to M' in one step
$M \to^* M'$	M reduces to M' in zero or more steps

The interesting relation is the single-step reduction, which is given by these rules (most names are from Panangaden, 2017):

$$\frac{}{(\lambda x.M)N \to M[x \mapsto N]}\ (\textsc{Beta}) \qquad \frac{N \to N'}{MN \to MN'}\ (\textsc{Mu})$$

$$\frac{M \to M'}{MN \to M'N}\ (\textsc{Nu}) \qquad \frac{M \to M'}{\lambda x.M \to \lambda x.M'}\ (\textsc{Xi})$$

Each of these rules has a rough analog in the world of programming:

Beta	Apply a function
Mu	Evaluate an actual parameter
Nu	Evaluate to get a closure that can be applied
Xi	Optimize code, e.g., by inlining

A complete computation reduces in zero or more steps. The $\to^*$ relation is the reflexive, transitive closure of the $\to$ relation. (The star, which you may have seen in regular expressions or in discrete-math class, is a standard way of saying "zero or more.")

$$\frac{}{M \to^* M}\ (\textsc{Rho}) \qquad \frac{M \to M' \qquad M' \to^* N}{M \to^* N}\ (\textsc{Tau})$$

To complete the lambda calculus, we need to be able to rename a bound variable (formal parameter):

$$\frac{y \text{ is not free in } M}{\lambda x.M \to \lambda y.M[x \mapsto y]}\ (\textsc{Alpha})$$

The Alpha rule is needed to establish the Church-Rosser theorem and the other results mentioned below, but it plays no role in reduction. (For purposes of modeling computation, we typically consider that alpha-renaming does not actually change the term. In technical language, if one term reduces to another by a sequence of alpha renamings, we identify those terms as equivalent.)

3 Implementing the rules

The reduction rules are implemented by a function `reduce`, which takes M as input and returns one of the following:

- `ONE_STEPS_TO` M', if $M \to M'$
- `DOESN'T_STEP`, if there is no M' such that $M \to M'$

The implementation can be constructed by following almost exactly the method outlined in the handout "Program Design with Typing Rules." A term has one of three forms (x, $\lambda x.M$, and $M\ N$), and given each form, you look for a rule that has the form in the conclusion. Here is what we know about each form:

- The x form is not reduced by any rule, so the reducer always returns `DOESN'T_STEP`.

- The $\lambda x.M$ form is reduced by exactly one of the rules above (Xi), and there's a reduction judgment above the line, so the reducer always makes a recursive call to try to reduce M.[1]

- The $M\ N$ form (application) might be reduced by Beta, Mu, or Nu.

The application form is where our old method breaks down: when a reducer is given an application form, there's more than one rule to choose from. A nondeterministic computation tries all the choices. A deterministic computation, which always tries rules in a fixed order, is called a *reduction strategy.*

The choices for reducing $M\ N$ are

Nu	Try to reduce M
Mu	Try to reduce N
Beta	(Only if M is a λ form: reduce $M\ N$

Three rules can be ordered in six ways, but in practice, only three orders are used:

- If we try first Beta, then Nu, and finally Mu, we are implementing the *normal-order* reduction strategy. It is also called the "leftmost, outermost" reduction strategy.

  Normal-order reduction is the basis for evaluation in "lazy" languages like "Haskell" and "Clean."

- If we try Mu before Beta, we have an *applicative-order* reduction strategy. There are two typical variants: Mu, Nu, Beta is a *left-to-right* applicative-order reduction strategy, and Nu, Mu, Beta is a *right-to-left* applicative-order reduction strategy.

  Applicative-order reduction is the basis for evaluation in "eager" languages like ML, JavaScript, C, and C++. Both left-to-right and right-to-left strategies can be found in the wild—sometimes both in the same language.

4 What difference it makes

Does it matter what reduction strategy we use? Yes: a reduction strategy affects both *performance* (how many reductions are needed to reach a normal form) and *termination* (whether a normal form is reached at all). However, if reduction *does* reach a normal form, it doesn't matter how we got there—all normal forms for a same term are equivalent. This property, which makes lambda calculus a good model for computation, is a consequence of the most important theorem: the *Church-Rosser theorem.* This theorem says that if $M \to A$ and $M \to B$, possibly by different reduction strategies and therefore producing different terms A and B, then there always exists a term C such that $A \to^* C$ and $B \to^* C$. If C is a normal form, then it is "the" normal form of M (and A and B).[2]

Different reduction strategies can always lead to the same normal form, but we still care about other differences. To illustrate those differences by example, consider an application $(\lambda x.M)\ N$ in several possible scenarios:

- Reducing N is expensive
- Reducing N doesn't terminate
- x does not appear free in M
- x appears free in M, multiple times

Who wins?

- If x doesn't appear free in M, normal-order reduction wins. In particular, if reduction of N doesn't terminate, applicative-order reduction doesn't terminate—but normal-order reduction does.

- If x appears free multiple times in M, applicative-order reduction probably wins: normal-order reduction makes multiple copies of N, and chances are those copies will be reduced. In normal-order reduction, each copy must be reduced to normal form separately. But in applicative-order reduction, N is reduced to normal form *before* it is ever substituted for x. That wins, *unless* the result of reducing M doesn't actually depend on x. (This is possible even if x appears free in M, and the question is undecidable.)

Bottom line, the only thing that can prevent us from finding a normal form is an infinite, nonterminating sequence of reductions. Normal-order reduction delays every reduction until the last possible minute, so if an infinite sequence of reductions can be avoided, normal-order reduction will avoid it. It's called "normal-order" reduction because if a normal form exists, normal-order reduction is guaranteed to find it.

5 Eta reduction and expansion

One additional rule is left out of most presentations of lambda calculus.

$$\frac{x \text{ not free in } M}{\lambda x.Mx \to M}\ (\textsc{Eta})$$

[1] When we add the Eta rule below, some λ forms will also be eligible for Eta reduction, complicating the reducer.

[2] Always up to the equivalence induced by Alpha-renaming.

The ETA reduction says, "the function that takes x and applies M to x is just M." In the lingo of lambda calculus, if x is not free in M, then $\lambda x.Mx$ is an "eta-redex."

The introduction of the ETA rule complicates a reducer. Given a form $\lambda x.N$, the reducer has to make a decision:

- Does N have the form $M\ x$? If so, is x a free variable of M? If N hase the form $M\ x$ and x is not free in M, then the whole form reduces to M (the ETA rule).
- Regardless of the form of N, it is always fair game to try to reduce N (the XI rule). But if that reduction fails, the ETA rule must be considered.

If M is not a normal form, then a choice between ETA and XI introduces some nondeterminism. But as far as I know, this nondeterminism has no implications in practice: ties between ETA and XI can be broken arbitrarily. Like XI, ETA is used primarily by optimizing compilers.

Perfectly reasonable variations on lambda calculus omit *both* ETA and XI. When both ETA and XI are omitted, Wikipedia calls the resulting evaluation strategy "call by name."

Henning Makholm has pointed out that the ETA rule justifies program transformations that an optimizing compiler could do anyway, while packing up the justification into a simple, reusable form. For example, no program can tell the difference between $\lambda f.\lambda x.fx$ and simply $\lambda f.f$, and an optimizing compiler may safely replace the long form with the short form. But without ETA, the proof of safety is, as Henning puts it, subtle and long-winded. Introducing the ETA rule makes it easy for the compiler writer to implement a whole family of code-improving transformations—no subtlety required.

That said, the immediate, practical reason to know about ETA is that the rule can used *backwards*. This transformation, from M to $\lambda x.Mx$, is known as "eta-expansion." In eager languages, it can be used to delay the evaluation of M. But its most frequent use is to convince the ML type checker to accept polymorphic functions that are produce by higher-order functions. For example, ML won't accept `null o rev` as a top-level value. But `fn xs => (null o rev) xs` is accepted just fine.

References

Prakash Panangaden, January 2015. "Notes on the Lambda-Calculus: COMP 598 Winter 2015." Obtained from the author, McGill University.