In general there may be many possible β-reductions that can be performed on a given λ-term. How do we choose which one to perform next? Does it matter?

A specification that tells which of the possible β-reductions to perform next is called a *reduction strategy*. The λ-calculus does not specify a reduction strategy; it is *nondeterministic*. A reduction strategy is needed in real programming languages to resolve the nondeterminism.

Two common reduction strategies for the λ-calculus are *normal order* and *applicative order*. Under the normal order reduction strategy, the leftmost-outermost redex is always the next to be reduced. By leftmost-outermost, we mean that if e_1 and e_2 are redexes in a term and e_1 is a subterm of e_2, then e_1 will not be reduced next; and among those redexes that are not subterms of other redexes, which are all pairwise incomparable with respect to the subterm relation, the leftmost one is chosen for reduction. The name "normal order" comes from the fact that, if a term has a normal form at all, then normal order reduction will converge to it.

The applicative order reduction strategy is similar, except that the leftmost-innermost redex is chosen. That is, if e_1 and e_2 are redexes in a term and e_1 is a subterm of e_2, then e_2 will not be reduced next; and among those redexes that do not contain other redexes as subterms, which are all pairwise incomparable with respect to the subterm relation, the leftmost one is chosen for reduction.

In real functional programming languages, reductions inside the body of a λ-abstraction are usually not performed (although optimizing compilers may do so in some instances). If we restrict the normal order and applicative order strategies so as not to perform reductions inside the body of λ-abstractions, we obtain strategies known as *call-by-name* (CBN) and *call-by-value* (CBV), respectively. Most functional languages use CBV, with the notable exception of Haskell.

We define a *value* to be a λ-term for which no β-reductions are possible given our reduction strategy. For example, $\lambda x.\, x$ would always be a value, while $(\lambda x.\, x)\ 1$ generally would not be, while $\lambda x.\, (\lambda y.\, y)\ x$ would be a value in CBV and CBN, but not for full β-reduction.

1 Call-By-Value

Under CBV, functions may only be called on values; that is, the arguments must be fully evaluated. Thus the β-reduction step $(\lambda x.\, e_1)\ e_2 \longrightarrow e_1[x \mapsto e_2]$ only applies if e_2 is a value. Here is an example of a CBV evaluation sequence, where we consider 3 and succ (the successor function) to be primitive constants.

$$(\lambda x.\, \mathsf{succ}\ x)\ ((\lambda y.\, \mathsf{succ}\ y)\ 3) \longrightarrow (\lambda x.\, \mathsf{succ}\ x)\ (\mathsf{succ}\ 3) \longrightarrow (\lambda x.\, \mathsf{succ}\ x)\ 4 \longrightarrow \mathsf{succ}\ 4 \longrightarrow 5$$

Note that this reduction is deterministic. No other sequence of reductions is possible in CBV.

We can define a small-step operational semantics for CBV λ-calculus by restricting the operational semantic rules we gave for full β-reduction in a previous lecture. We need a notion of values, which we define simply as λ-abstractions. To be extremely formal, we can give the (extremely simple) BNF grammar for values:

$$v \quad ::= \quad \lambda x.\, e$$

Now we can restrict full β-reduction. First, we entirely remove the rule for evaluating underneath a λ-abstraction. Second, we restrict the β-reduction (function application) rule to apply *only when the argument is already a value*. We can denote this requiring the syntactic form $(\lambda x.\, e)\ v$ for β-reduction to apply. Finally, we want reduction of function application to be deterministic, so we must choose if the function position or argument position is evaluated first. As CBV comes from applicative order, it is left-first, which means we evaluate the expression in function position first. We thus restrict the rule for evaluating the argument to cases

where the function position is already a value. The result is the following small-step operational semantics for CBV λ-calculus.

$$\frac{}{(\lambda x.\, e)\ v \longrightarrow e[x \mapsto v]} \qquad \frac{e_1 \longrightarrow e_1'}{e_1\ e_2 \longrightarrow e_1'\ e_2} \qquad \frac{e_2 \longrightarrow e_2'}{v\ e_2 \longrightarrow v\ e_2'}$$

2 Call-by-Name

Call-by-value is not the only reasonable evaluation order. Another option is to defer evaluating arguments as long as possible. In this approach, called *call-by-name* (CBN), we continue to evaluate left-to-right, but now we apply functions to arbitrary expressions. Using the same example as in Section 1, we would get:

$$(\lambda x.\, \mathsf{succ}\ x)\ ((\lambda y.\, \mathsf{succ}\ y)\ 3) \longrightarrow \mathsf{succ}\ ((\lambda y.\, \mathsf{succ}\ y)\ 3) \longrightarrow \mathsf{succ}\ (\mathsf{succ}\ 3) \longrightarrow \mathsf{succ}\ 4 \longrightarrow 5$$

This strategy produces a form of *lazy evaluation*; function arguments are only evaluated if they are actually needed. This is the basic idea used by languages like Haskell (though Haskell itself uses a more complicated evaluation strategy so that arguments used multiple times are still only evaluated once).

Because we do not need to evaluate arguments in function application, the small-step operational semantic rules for CBN are slightly simpler than those for CBV.

$$\frac{}{(\lambda x.\, e_1)\ e_2 \longrightarrow e_1[x \mapsto e_2]} \qquad \frac{e_1 \longrightarrow e_1'}{e_1\ e_2 \longrightarrow e_1'\ e_2}$$

To understand the differences between CBV and CBN, let's look back at the example we had in a previous lecture that showed the nondeterminism of full β-reduction:

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

Recall that, with full β-reduction, this term had both an infinite (nonterminating) reduction sequence—stepping Ω repeatedly—as well as a terminating reduction sequence—stepping the outer function application once to arrive at $\lambda y.\, y$. Our new reduction strategies are deterministic, but they are fundamentally different, and we can see that because they will produce different results on this term.

Using CBV, we first evaluate the argument, which is Ω, so CBV evaluation on this term will never terminate. With CBN, however, we first perform substitution of the outer redex, immediately reducing to the value $\lambda y.\, y$. One strategy (CBN) terminates, while the other (CBV) does not. The fact that these are fundamentally different shows that the two evaluation strategies are fundamentally different.

Notably, it is impossible for the difference to be reversed. CBN comes from restricting normal order β-reduction to not evaluating under λ-abstraction, and it has the useful property that it will only loop infinite when every other reduction strategy would also loop infinitely. That is, it will find a normal form if one exists, though not necessarily in the most efficient way.

3 Evaluation Contexts

We have seen three different semantics for λ-calculus, and they all have a similar structure: one rule that defines β-reduction, and other rules simply specify the order in which the β-reductions can be performed. Small-step operational semantics always have this distinction, and in fact we can split rules into two classes:

- *reduction rule*, which describe the actual computational steps; and
- *evaluation order rules*, which constrain the choice of reductions that can be performed next.

For example, in the CBV operational semantics in Section 1, the first rule, specifying how to apply functions to values, is a reduction rule (β-reduction), while the other two rules are both evaluation order rules. They say that reduction may always be applied to a redex on the left side of an application, and reduction may be applied to the right side, but only when the right side is fully reduced.

There are two evaluation order rules for CBV λ-calculus, one for CBN, and three for full β-reduction. While these numbers are small, in real programming languages there are typically many more caused by various different data structures and programming constructs. We would like to avoid having to write them all out separately and instead have a more compact representation.

Evaluation contexts provide a simple mechanism to do just that. An evaluation context E, sometimes written $E[\cdot]$, is a metaexpression representing a family of λ-terms with a special variable $[\cdot]$ called the *hole*. If $E[\cdot]$ is an evaluation context, then $E[e]$ represents E with the term e substituted for the hole.

We then include a *context rule* in our operational semantics

$$\text{[E-Ctx]}\ \frac{e \longrightarrow e'}{E[e] \longrightarrow E[e']}$$

which says that we may apply the reduction $e \longrightarrow e'$ in the context E.

For CBV λ-calculus, the two evaluation order rules can be compactly represented using the two evaluation contexts $[\cdot]\ e$ and $v\ [\cdot]$. We could thus specify CBV λ-calculus as

$$\frac{}{(\lambda x.\, e)\ v \longrightarrow e[x \mapsto v]} \qquad \frac{e \longrightarrow e'}{E[e] \longrightarrow E[e']} \qquad E \ ::= \ [\cdot]\ e \mid v\ [\cdot]$$

The CBN λ-calculus has an equally simple specification:

$$\frac{}{(\lambda x.\, e_1)\ e_2 \longrightarrow e_1[x \mapsto e_2]} \qquad \frac{e \longrightarrow e'}{E[e] \longrightarrow E[e']} \qquad E \ ::= \ [\cdot]\ e$$

We can do the same thing for full β-reduction.

$$\frac{}{(\lambda x.\, e_1)\ e_2 \longrightarrow e_1[x \mapsto e_2]} \qquad \frac{e \longrightarrow e'}{E[e] \longrightarrow E[e']} \qquad E \ ::= \ [\cdot]\ e \mid e\ [\cdot] \mid \lambda x.\, [\cdot]$$

The operational semantics for full β-reduction originally had four operational semantic rules, but this simpler representation has only two semantic rules plus a simple definition of evaluation contexts.

3.1 Nested Evaluation Contexts

Note that in the reduction rules above, the definitions of E do not specify *all* contexts in which β-reduction can apply. There may be compound contexts formed by nesting contexts. We can still step these nested contexts by nested applications of the E-Ctx rule.

For example, in CBV λ-calculus the compound context $(v\ [\cdot])\ e$ is also valid, and we can step under it with two applications of E-Ctx:

$$\frac{\dfrac{e_1 \longrightarrow e_2}{v\ e_1 \longrightarrow v\ e_2}}{(v\ e_1)\ e \longrightarrow (v\ e_2)\ e}$$

Here we applied E-Ctx with $E = v\ [\cdot]$ and expression e_1 to prove $v\ e_1 \longrightarrow v\ e_2$. We then use that proof and the E-Ctx rule again with $E = [\cdot]\ e$ and expression $v\ e_1$ to show that $(v\ e_1)\ e \longrightarrow (v\ e_2)\ e$.

We can bypass this need to apply E-Ctx repeatedly to prove a single step by instead making the definition of E inductive so that it captures all possible evaluation contexts. If we were to do that, the evaluation contexts for CBV λ-calculus would then be

$$E \quad ::= \quad [\cdot] \mid E\ e \mid v\ E.$$

Both of these approaches arrive at the same result. Neither is inherently better than the other, though sometimes one makes it simpler to state or prove your theorems.

3.2 Error Propagation

As an additional note, evaluation contexts are also extremely useful for defining the semantics of errors and exceptions. If the language has a special error value error, it is extremely simple to propagate it using the evaluation order rule

$$\frac{}{E[\mathsf{error}] \longrightarrow \mathsf{error}}$$

This approach removes the need to painstakingly spell out all of the situations in which an error can propagate. We will revisit this idea later when talking about exceptions and exception handling mechanisms.