

Elements of Programming Languages

Lecture 15: Evaluation strategies and laziness

James Cheney

University of Edinburgh

November 13, 2025

Overview

- Final few lectures: cross-cutting language design issues
- So far:
 - Type safety
 - References, arrays, resources
- Today:
 - Evaluation strategies (by-value, by-name, by-need)
 - Impact on language design (particularly handling *effects*)

Evaluation order

- We've noted already that some aspects of small-step semantics seem arbitrary
 - For example, left-to-right or right-to-left evaluation
- Consider the rules for $+$, $\times$. There are two kinds:
computational rules that actually do something:

$$\frac{}{v_1 + v_2 \mapsto v_1 +_{\mathbb{N}} v_2} \qquad \frac{}{v_1 \times v_2 \mapsto v_1 \times_{\mathbb{N}} v_2}$$

- and *administrative* rules that say how to evaluate inside subexpressions:

$$\frac{e_1 \mapsto e'_1}{e_1 \oplus e_2 \mapsto e'_1 \oplus e_2} \qquad \frac{e_2 \mapsto e'_2}{v_1 \oplus e_2 \mapsto v_1 \oplus e'_2}$$

Evaluation order

- We can vary the *evaluation order* by changing the administrative rules.
- To evaluate right-to-left:

$$\frac{e_2 \mapsto e'_2}{e_1 \oplus e_2 \mapsto e_1 \oplus e'_2} \qquad \frac{e_1 \mapsto e'_1}{e_1 \oplus v_2 \mapsto e'_1 \oplus v_2}$$

- To leave the evaluation order *unspecified*:

$$\frac{e_1 \mapsto e'_1}{e_1 \oplus e_2 \mapsto e'_1 \oplus e_2} \qquad \frac{e_2 \mapsto e'_2}{e_1 \oplus e_2 \mapsto e_1 \oplus e'_2}$$

by lifting the constraint that the other side has to be a value.

Call-by-value

- So far, function calls evaluate arguments to values *before* binding them to variables

$$\frac{e_1 \mapsto e'_1}{e_1 \ e_2 \mapsto e'_1 \ e_2} \quad \frac{e_2 \mapsto e'_2}{v_1 \ e_2 \mapsto v_1 \ e'_2} \quad \frac{}{(\lambda x. e) \ v \mapsto e[v/x]}$$

- This *evaluation strategy* is called *call-by-value*.
 - Sometimes also called *strict* or *eager*
- “Call-by-value” historically refers to the fact that expressions are evaluated before being passed as parameters
- It is the default in most languages

Example

- Consider $(\lambda x.x \times x) (1 + 2 \times 3)$
- Then we can derive:

$$\frac{\frac{2 \times 3 \mapsto 6}{1 + 2 \times 3 \mapsto 1 + 6}}{(\lambda x.x \times x) (1 + 2 \times 3) \mapsto (\lambda x.x \times x) (1 + 6)}$$

- Next:

$$\frac{1 + 6 \mapsto 7}{(\lambda x.x \times x) (1 + 6) \mapsto (\lambda x.x \times x) 7}$$

- Finally:

$$\overline{(\lambda x.x \times x) 7 \mapsto 7 \times 7 \mapsto 49}$$

Interpreting call-by-value

We evaluate subexpressions fully before substituting them for variables:

```
def eval (e: Expr): Value = e match {  
  ...  
  case Let(x,e1,e2) => eval(subst(e2,eval(e1),x))  
  ...  
  case Lambda(x,ty,e) => Lambda(x,ty,e)  
  
  case Apply(e1,e2)  => eval(e1) match {  
    case Lambda(x,_,e) => apply(subst(e,eval(e2),x))  
  }  
}
```

Call-by-name

- Call-by-value may evaluate expressions unnecessarily (leading to nontermination in the worst case)

$$(\lambda x.42) \text{ loop} \mapsto (\lambda x.42) \text{ loop} \mapsto \dots$$

- An alternative: substitute expressions *before* evaluating

$$(\lambda x.42) \text{ loop} \mapsto 42$$

- To do this, *remove* second administrative rule, and *generalize* the computational rule

$$\frac{e_1 \mapsto e'_1}{e_1 \ e_2 \mapsto e'_1 \ e_2} \qquad \frac{}{(\lambda x. e_1) \ e_2 \mapsto e_1[e_2/x]}$$

- This evaluation strategy is called *call-by-name* (the “name” is the expression)

Example, revisited

- Consider $(\lambda x. x \times x) (1 + (2 \times 3))$
- Then in call-by-name we can derive:

$$\overline{(\lambda x. x \times x) (1 + (2 \times 3)) \mapsto (1 + (2 \times 3)) \times (1 + (2 \times 3))}$$

- The rest is standard:

$$\begin{aligned} (1 + (2 \times 3)) \times (1 + (2 \times 3)) &\mapsto (1 + 6) \times (1 + (2 \times 3)) \\ &\mapsto 7 \times (1 + (2 \times 3)) \\ &\mapsto 7 \times (1 + 6) \\ &\mapsto 7 \times 7 \mapsto 49 \end{aligned}$$

- Notice that we recompute the argument twice!

Interpreting call-by-name

We substitute expressions for variables *before* evaluating.

```
def eval (e: Expr): Value = e match {  
  ...  
  case Let(x,e1,e2 ) => eval(subst(e2,e1,x))  
  ...  
  case Lambda(x,ty,e) => Lambda(x,ty,e)  
  
  case Apply(e1,e2) => eval(e1) match {  
    case Lambda(x,_,e) => eval(subst(e,e2,x))  
  }  
}
```

Call-by-name in Scala

- In Scala, can flag an argument as being passed by name by writing `=>` in front of its type
- Such arguments are evaluated only when needed (but may be evaluated many times)

```
scala> def byName(x : => Int) = x + x
byName: (x: => Int)Int
scala> byName({ println("Hi_ there!"); 42})
Hi there!
Hi there!
res1: Int = 84
```

- This can be useful; sometimes we actually want to re-evaluate an expression (see tutorial 8)

Simulating call-by-name

- Using functions, we can simulate passing $e : \tau$ by name in a call-by-value language
- Simply pass it as a “delayed” expression
 $\lambda().e : \text{unit} \rightarrow \tau$.
- When its value is needed, apply to $()$.
- Scala’s “by name” argument passing is basically syntactic sugar for this (using annotations on types to decide when to silently apply to $()$)

Comparison

- Call-by-value evaluates every expression at most once
 - ... whether or not its value is needed
 - Performance tends to be more predictable
 - Side-effects happen predictably
- Call-by-name only evaluates an expression if its value is *needed*
 - Can be faster (or even avoid infinite loop), if not needed
 - But may evaluate multiple times if needed more than once
 - Reasoning about performance requires understanding when expressions are needed
 - Side-effects may happen multiple times or not at all!

Best of both worlds?

- A third strategy: evaluate each expression when it is needed, but then *save the result*
- If an expression's value is never needed, it never gets evaluated
- If it is needed many times, it's still only evaluated once.
- This is called *call-by-need* (or sometimes *lazy*) evaluation.

Laziness in Scala

- Scala provides a `lazy` keyword
- Variables declared `lazy` are not evaluated until needed
- When they are evaluated, the value is *memoized* (that is, we store it in case of later reuse).

```
scala> lazy val x = {println("Hello"); 42}
x: Int = <lazy>
scala> x + x
Hello
res0: Int = 84
```

Laziness in Scala

- Actually, laziness can also be *emulated* using references and variant types:

```
class Lazy[A](a: => A) {  
  private var r: Either[A, () => A] = Right{() => a}  
  def force = r match {  
    case Left(a) => a  
    case Right(f) => {  
      val a = f()  
      r = Left(a)  
      a  
    }  
  }  
}
```

Call-by-need

- The semantics of call-by-need is a little more complicated.
- We want to *share* expressions to avoid recomputation of needed subexpressions
- We can do this using a “memo table” $\sigma : Loc \rightarrow Expr$
 - (similar to the *store* we used for references)
- Idea: When an expression e is bound to a variable, replace it with a *label* ℓ bound to e in σ
 - The labels are *not* regarded as values, though.
 - When we try to evaluate the label, look up the expression in the store and evaluate it

Rules for call-by-need

 $\sigma, e \mapsto \sigma', e'$

$$\frac{}{\sigma, (\lambda x. e_1) e_2 \mapsto \sigma[\ell := e_2], e_1[\ell/x]}$$

$$\frac{}{\sigma, \text{let } x = e_1 \text{ in } e_2 \mapsto \sigma[\ell := e_1], e_2[\ell/x]}$$

$$\frac{}{\sigma[\ell := v], \ell \mapsto \sigma[\ell := v], v} \quad \frac{\sigma, e \mapsto \sigma', e'}{\sigma[\ell := e], \ell \mapsto \sigma'[\ell := e'], \ell}$$

- When we reduce a function application or let, add expression to the memo table and replace with label
- When we encounter the label, look up its value or evaluate it (if not yet evaluated)

Rules for call-by-need

As with L_{Ref} , we also need to adjust all of the rules to handle σ .

$$\sigma, e \mapsto \sigma', e'$$

$$\frac{\sigma, e_1 \mapsto \sigma', e'_1}{\sigma, e_1 \oplus e_2 \mapsto \sigma', e'_1 \oplus e_2}$$

$$\frac{\sigma, e_2 \mapsto \sigma', e'_2}{\sigma, v_1 \oplus e_2 \mapsto \sigma', v_1 \oplus e'_2}$$

$$\frac{}{\sigma, v_1 + v_2 \mapsto \sigma, v_1 +_{\mathbb{N}} v_2}$$

$$\frac{}{\sigma, v_1 \times v_2 \mapsto \sigma, v_1 \times_{\mathbb{N}} v_2}$$

$$\vdots$$

Example, revisited again

- Consider $(\lambda x.x \times x) (1 + 2 \times 3)$
- Then we can derive:

$$\overline{[], (\lambda x.x \times x) (1 + 2 \times 3) \mapsto [\ell = 1 + (2 \times 3)], \ell \times \ell}$$

- Next, we have:

$$[\ell = 1 + (2 \times 3)], \ell \times \ell \mapsto [\ell = 1 + 6], \ell \times \ell \mapsto [\ell = 7], \ell \times \ell$$

- Finally, we can fill in the ℓ labels:

$$[\ell = 7], \ell \times \ell \mapsto [\ell = 7], 7 \times \ell \mapsto [\ell = 7], 7 \times 7 \mapsto [\ell = 7], 49$$

- Notice that we compute the argument only once (but only when its value is needed).

Pure functional programming

- Call-by-name/call-by-need interact *badly* with side-effects
- On the other hand, they support very strong *equational* reasoning about programs
- Haskell (and some other languages) are *pure*: they adopt lazy evaluation, and forbid **any** side-effects!
- This has strengths and weaknesses:
 - (+) Easier to optimize, parallelize because side-effects are forbidden
 - (+) Can be faster
 - (−) but memoization has overhead (e.g. memory leaks) and performance is less predictable
 - (−) Dealing with I/O, exceptions etc. requires major rethink

I/O in Haskell

- Dealing with I/O and other side-effects in Haskell was a long-standing challenge
- Today's solution: use a type constructor `IO a` to “encapsulate” side-effecting computations

```
do { x <- readLn::IO Int ; print x }
```

```
123
```

```
123
```

- Note: `do`-notation is also a form of *comprehension*
- Haskell's *monads* provide (equivalents of) the `map` and `flatMap` operations

Lazy data structures

- We have (so far) assumed eager evaluation for data structures (pairs, variants)
 - e.g. a pair is fully evaluated to a value, even if both components are not needed
- However, alternative (lazy) evaluation strategies can be considered for data structures too
 - e.g. could consider a pair (e_1, e_2) to be a value; we only evaluate e_1 if it is “needed” by applying `fst`:

```
ghci> fst (42, undefined) == 42
```

- An example: *streams* (see tutorial 8)

```
ghci> let ones = 1::ones
```

```
ghci> take 10 ones
```

Summary

- Today we covered:
 - Call by value
 - Call by name
 - Call by need (lazy evaluation)
- Next time:
 - exceptions
 - tail recursion
 - continuations [non-examinable]