Lecture: Evaluation Order

We use the following rule to evaluate function applications:

$$\beta_v \; \frac{e_1 \Downarrow \lambda x.e \qquad e_2 \Downarrow v \qquad e[x \mapsto v] \Downarrow v'}{e_1 \; e_2 \Downarrow v'}$$

The rule evaluates the argument e_2 to a value v before substituting it into the body. An alternative semantics would simply substitute the argument expression without evaluating it first:

$$\beta \; \frac{e_1 \Downarrow \lambda x.e \qquad e_2 \Downarrow v \qquad e[x \mapsto e_2] \Downarrow v'}{e_1 \; e_2 \Downarrow v'}$$

The former rule is implements *call-by-value*, whereas the latter implements *call-by-name*. In certain situations, call-by-name requires more steps of evaluation than call-by-value. For example, fig. 7.1 shows the steps needed to evaluate the expression $(\mathit{fun}\ x \to x + x)(10 + 10)$ using call-by-value and call-by-name. Using call-by-name, the sum $10 + 10$ is evaluated twice since the expression is first substituted into the body of the function. In contrast, call-by-value evaluates $10 + 10$ only once.

OCaml and most other programming languages have a call-by-value semantics. But, if you didn't already know that, how could you write a test program to determine the order of evaluation? We know that our previous example requires more "steps of evaluation" in call-by-name, but we cannot directly measure the number of steps a program takes.

An alternate approach is to write a program that uses side-effects to count the number of times an expression is evaluated For example, consider the following OCaml program:

```
let n = ref 0 in
let f () = x + x in
let _ = f (n := !n + 1; 0)
printf "n = %d\n" !n
```

This program has a mutable cell called `n` that is initialized to zero and a function `f` that consumes one argument and uses it twice. The program applies `f` to an expression that increments `n` and then produces `0`. Therefore, if OCaml had a call-by-name semantics, then the value value of `n` would be `2`. However, if we run the program, we'll observe that it is `1`, which indicates a call-by-value semantics.

Let's now consider another programming language, called Haskell, which is related to OCaml. Both OCaml and Haskell are principally functional programming languages, but Haskell is more "pure" than OCaml and doesn't have mutable cells. Therefore, it is not easy to translate the OCaml program above into Haskell.

However, both Haskell and OCaml have exceptions, and we can exploit exceptions as follows: in call-by-name, an argument that is never used is never evaluated, whereas call-by-value evaluates arguments immediately. The following two programs, written in OCaml and Haskell respectively, are essentially the same, but the OCaml version raises an exception, whereas the Haskell version terminates normally.

$$\begin{aligned} & (\mathit{fun}\ x \to x + x)(10 + 10) \\ = {} & (\mathit{fun}\ x \to x + x)(20) \\ = {} & 20 + 20 \\ = {} & 40 \end{aligned}$$

(a) Call-by value

$$\begin{aligned} & (\mathit{fun}\ x \to x + x)(10 + 10) \\ = {} & (10 + 10) + (10 + 10) \\ = {} & 20 + (10 + 10) \\ = {} & 20 + 20 \\ = {} & 40 \end{aligned}$$

(b) Call-by-name

Figure 7.1: Call-by-value vs. call-by-name

```
let myfunc x = 100 in
myfunc (failwith "Blow up") (* Raises an exception *)
```

```
let myfunc x = 100
myfunc (error "Blow up") -- Produces 100
```

Think! Suppose a programming language had neither exceptions or mutable state (i.e., what many consider a "pure functional language"). Can you write a program to determine evaluation order?

Lazy Evaluation Haskell actually implements a variant call-by-name known as *call-by-need* or *lazy evaluation*. The high-level idea is that expressions are only evaluated when needed (like call-by-name) and only evaluated once. Therefore, once an expression is evaluated, it is replaced with the result of evaluation.

For example, the following expression prints "Running" when it is evaluated:

```
Prelude> import Debug.Trace
Prelude Debug.Trace> trace "Running" (10 + 10)
Running
20
```

If we use this expression as the argument of a function that uses it twice, it is only evaluated once:

```
Prelude Debug.Trace> let g x = x + x
Prelude Debug.Trace> g (trace "Running" (10 + 10))
Running
40
```

Think! At the start of this chapter, we wrote the reduction rules for call-by-value and call-by-name evaluation. Try to do the same for call-by-need.

1 Programming With Lazy Evaluation

Lazy evaluation is a powerful feature that has several applications. In this section, we will consider two use cases.

1.1 Control Operators

The boolean operators and and or exist in most languages, but the boolean nand operator is far less common. nand is an interesting operator because it can be *short-circuited*. i.e., false nand y is true, regardless of the value of y. nand need not evaluate y at all.

The three programs below implement nand in OCaml, Haskell, and R respectively:

```
let nand x y =
  if x then
    if y then
      false
    else
      true
  else
    true

nand false (failwith "blow up")
```

```
let nand x y =
if x then
  if y then
    False
  else
    True
else
  True

nand False (error "blow up")
```

```
nand <- function(a, b) {
  if (a) {
    if (b) {
      return(FALSE)
    } else {
      return(TRUE)
    }
  } else {
    return(TRUE)
  }
}

nand(FALSE, stop("blow up"))
```

To keep things simple, we have written all three programs using if-statements, which makes the flow of control clear. The Haskell and R versions implement short-circuiting correctly because both support lazy evaluation. In contrast, the OCaml version does not short-circuit and throws the exception.

Think! Is it possible to define a short-circuiting nand function in OCaml?

We can use lazy evaluation to define new control operators in this way.

```
import Data.Maybe

data Exp
  = Int Int
  | Div Exp Exp
  | Id String
  | App Exp Exp
  | Fun String Exp

data Value
  = VInt Int
  | VClosure Env String Exp

type Env = [(String, Value)]

eval :: Env -> Exp -> Value
eval env exp = case exp of
  Int n -> VInt n
  Fun x body -> VClosure env x body
  Div e1 e2 -> case (eval env e1, eval env e2) of
    (VInt m, VInt n) -> VInt (div m n)
    otherwise -> error "expected two numbers"
  Id x -> fromJust (lookup x env)
  App e1 e2 -> case eval env e1 of
    VClosure env' x body -> eval ((x, eval env e2) : env') body
    otherwise -> error "expected closure"
```

Figure 7.2: A transliteration of an OCaml interpreter into Haskell.

1.2 Interpreters

Haskell and OCaml are closely related languages. They both support type inference, algebraic datatypes and pattern matching. So, it is very straightforward to translate our OCaml interpreter into Haskell, as shown fig. 7.2. However, since Haskell has a call-by-need semantics, this interpreter implements call-by-need too!

For example, the following program does not signal an exception:

```
eval [] (App (Fun "x" (Int 200)) (Div (Int 20) (Int 0)))
```

However, the equivalent OCaml program would signal a division-by-zero exception.

The call-by-value semantics of our original interpreter was an accident. It happened to work because we were using OCaml. If we'd chosen to use use Haskell instead, it would have been wrong. We know how to implement first-class functions in a way that doesn't rely on the underlying language supporting first-class functions. Can we do the same for evaluation order? i.e., is there a way to write the interpreter such that it implements call-by-value, irrespective of whether the metalanguage uses call-by-value or call-by-name?