CSC 347 - Concepts of Programming Languages

Strict and Nonstrict Evaluation

Instructor: James Riely

Learning Objectives

❓ How much of an expression should be evaluated when computing a result?

- Perform expression evaluation
- Perform function calls and function argument evaluation

Strict and Nonstrict Operators

- A *strict* construct evaluates all of its operands before it runs

Strict constructs

- Arithmetic operators: `+`, `-`, ...
- Comparison operators: `<`, `<=`, ...
- Bitwise operators: `|`, `&`, ...
- Function calls: `f(x,y)`

Non-strict constructs

- `e1 && e2`: strict in `e1`, but not `e2`
- `e1 || e2`: strict in `e1`, but not `e2`
- `if e1 then e2 else e3`: strict in `e1`, but not `e2` or `e3`

Conditional Expression

Conditional expression `if e1 then e2 else e3`

- Evaluate `e1`; if `true` then evaluate `e2`, else evaluate `e3`

```
1 for i <- 0 to 10 do
2   print(i)
3   if i % 2 == 0
4   then println(": even")
5   else println(": odd")
6 end for
```

Conditionals in C

- Conditional statement: executes `return 1` or `return n*...`, never both

```
1 int fact (int n) {
2   if (n <= 1) {
3     return 1;
4   } else {
5     return n * fact (n - 1);
6   }
7 }
```

- Conditional expression: must be non-strict, otherwise non-terminating recursion

```
1 int fact (int n) {
2   return (n <= 1) ? 1 : n * fact (n - 1);
3 }
```

What Happens?

- Function calls are strict

```
1 def f(b: Boolean, t: Unit, f: Unit) : Unit =
2   if b then t else f
3
4 for i <- 0 to 10 do
5   print(i)
6   f(i % 2 == 0,
7     println(": even"),
8     println(": odd"))
9 end for
```

- Both "even" and "odd" are printed every time `f` is invoked

What Happens?

- *Thunks*: pass-by-name for explicit nonstrict evaluation of function arguments

```
1 def f(b: Boolean, t: => Unit, f: => Unit) : Unit =
2   if b then t else f
3
4 for i <- 0 to 10 do
5   print(i)
6   f(i % 2 == 0,
7     println(": even"),
8     println(": odd"))
9 end for
```

- "even" and "odd" are printed again alternating

Call-By-Value: Scala

```
1 def f(x: Double) : Double =
2   val x1 = x
3   val x2 = x
4   x1 - x2
5 end f
6
7 println("f= " + f (Math.random())) // prints 0 (difference of same random Double)
```

Call-By-Value: Scala

- Scala allows functions as parameters (no argument: *delayed value* = *thunk*)

Passing Functions in Scala

```scala
def g(x: () => Double) : Double =
  val x1 = x()
  val x2 = x()
  x1 - x2
end g

// print difference of 2 random doubles
println ("g= " + g (() => Math.random()))
```

Emulate in OOP (Java)

```java
interface java.util.function.Supplier<T> { T get(); }

class GetRnd implements Supplier<Double> {
  public Double get() { return Math.random(); }
}

class Client {
  public static double g(Supplier<Double> t) {
    double x1 = t.get();
    double x2 = t.get();
    return x1-x2;
  }
}

// print difference of 2 random doubles
System.out.println ("g= " + g (new GetRnd()));
// functional notation without explicit class GetRnd
System.out.println ("g= " + g (() -> Math.random()));
```

Call-By-Name: Scala

- Scala has a special syntax for using thunks as parameters
- *Call-by-name* parameters are non-strict

Thunks in Scala

```scala
def h(x: => Double) : Double =
  // discouraged use of thunks
  val x1 = x
  val x2 = x
  x1 - x2
end h

// prints difference of 2 random doubles
println ("h= " + h (Math.random()))
```

Call-By-Name: Scala

- Curried call-by-name functions create new control constructs

```
def myWhile (cond: => Boolean) (body: => Unit) : Unit =
  if (cond)
    body
    myWhile (cond) (body)
end myWhile
```

- Use (brackets `{}` or parentheses `()` are required)

```
var i = 3
myWhile (i > 0) {
  println ("i= " + i)
  i = i - 1
}
```

Summary

Strict Evaluation

- Evaluates **all operands** to determine value of an expression
  - Arithmetic operators etc.
  - Function arguments

Nonstrict Evaluation

- Evaluates **only necessary operands** to determine value
  - Conditional expression
  - Shortcut evaluation of boolean expressions
- Thunks: request explicit nonstrict evaluation of function arguments