Lazy Evaluation

CIS-3030, Vermont Technical College
Peter C. Chapin

Eager Evaluation

- Consider
  - ```
    def checkValue(x: Int) =
      if (x < 0)
        println(s"The value $x is negative")

    checkValue(a + f())
    ```
  - In an *eager language* the expression `a + f()` is evaluated and the result sent to `checkValue`.
    - That value is used twice in this example.
    - Side effects of `f()` only happen once.
    ```

Lazy Evaluation

- Consider
  - ```
    def checkValue(x: Int) =
      if (x < 0)
        println(s"The value $x is negative")

    checkValue(a + f())
    ```
  - In a *lazy language* the expression `a + f()` is passed to `checkValue` **unevaluated**.
    - Parameter is evaluated twice in this example (maybe).
    - Side effects of `f()` happen twice (maybe).
    ```

Lazy Evaluation More Expressive

- Some programs work
  - ```
    def computeBase(x: Int) =
      if (someCondition) x + 1 else 0

    computeBase( a/b )
    ```
  - What if `b == 0`?
    - In an eager language `a/b` throws an exception
    - In a lazy language it works if `someCondition` is always false when `b == 0` is true.
      - The parameter x is not needed in that case!
    ```

Which is Faster?

- Eager Evaluation
  - Function arguments evaluated only once
    - … even if used multiple times in the function body.
- Lazy Evaluation
  - Function arguments not evaluated at all
    - … if never used in a particular run of the function.
- Conclusion…
  - A wash. Depends on program and compiler.

With Side Effects?

- Eager Evaluation
  - Side effects occur when arguments evaluated
    - … easy to understand and reason about.
- Lazy Evaluation
  - Side effects occur “later.”
    - … confusing (especially when debugging).
- Conclusion
  - Lazy evaluation works better in functional setting.

Popularity?

- Eager Evaluation
  - Overwhelmingly more popular
    - All imperative languages. Many functional languages.
- Lazy Evaluation
  - Haskell
    - … and its dialects and followers.
- Why?
  - Eager evaluation is easier to implement.

What about Scala?

- Eager by default… allows lazy as an option.
  - Simulating lazy evaluation is fairly easy.
    - ```
      def maybeDoOperation(op: () => Unit) =
        if (someCondition) op() else ()

      maybeDoOperation( () => println(a/b) )
      ```
    - Parameter function from Unit => Unit
      - Compiler makes closure out of println(a/b)
    - Function only invoked if someCondition true.
      - That's when side effects of evaluating println(a/b) happen
      ```

By-Name Parameters

- Scala offers syntactic sugar
  - ```
    def maybeDoOperation(op: => Unit) =
      if (someCondition) op else ()

    maybeDoOperation( println(a/b) )
    ```
  - Compiler understands parameter is function taking Unit and returning Unit (in this case).
  - Reduces syntactic burden at call site.
    ```

General Usage

Expression evaluating to Boolean

- Allows expressions to be passed unevaluated
  - `def requiring[A](`

```
    condition: => Boolean,
    action   : => A) = {

    Controller.preconditionsActive match {
      case false => action
      case true  =>
        if (!condition)
          throw new ContractFailureException(
            "Failed precondition“)
        else
          action
    }
```

Contract Usage

- The previous method can be called like this
  - ```
    val result =
      requiring(x > 0, doStuff(myArray(x)))
    ```
  - If precondition checks are off…
    - … the condition is not evaluated
    - … the other expression is evaluated once
  - If precondition checks are on…
    - … the condition is evaluated
    - … the other expression is not evaluated if the condition is false.
    ```

Use Two Parameter Lists

- Allows expressions to be passed unevaluated
  - `def requiring[A](`
    `condition: => Boolean)`
    `(action : => A) = {`

```
Controller.preconditionsActive match {
  case false => action
  case true  =>
    if (!condition)
      throw new ContractFailureException(
        "Failed precondition“)
    else
      action
}
```

This is Scala!

- Now requiring looks like a control structure
  - ```
    val result = requiring(x > 0) {
      val temp = …
      // Code of arbitrary complexity
      myArray(temp + x/2)
    }
    ```
  - Second parameter list enclosed in { … }
  - … passed unevaluated into `requiring`.
  - … evaluated inside `requiring` on demand
    ```

Domain Specific Languages

- Scala is good for internal DSLs because
  - You can define new operators
    - Operators are just method names with funny letters
  - You can define new control structures
    - As methods taking by-name parameters
    - … together with Scala's syntactic abbreviations

Lazy Vals

- A Lazy val is one where the initializer is evaluated only if needed.
  - `lazy val x = f()`
    `if (someCondition) x + 1 else 0`
  - Here `f()` is called only if `someCondition` is true.
    - Avoids side effects when not wanted/needed.
    - Can be faster.

Compare

- Three different ways to compute a value
  - ```
    val x = f()
    def y = f()
    lazy val z = f()
    ```
  - The val…
    - Intialized exactly once (needed or not).
  - The def…
    - Called each time it is used (but not when defined).
  - The lazy val
    - Initialized exactly once but deferred until it is used.
    ```