Lazy Evaluation

February 13th, 2022

What is it?

The simplest way to explain lazy evaluation is when the compiler waits to evaluation expression until the value of the expression is needed. Functional programming languages like Haskell are known for their use of lazy evaluation but some mainstream languages like Python also have been taking advantage of the strategy.

While discussing lazy evaluation, it is also important to know the other side of the coin, strict evaluation. Strict evaluation, where everything is evaluated immediately. This is used a lot in object-oriented focused languages like Java.

With these two methods of evaluation, lets look at the benefits of lazy evaluation.

Is it useful?

The first benefit is that it is more efficient when compiling code. With the compiler only evaluating expressions as the values are needed, the compiler is able to save time and computing resources by skipping over expressions that will not be used.

The second benefit is that you can work with infinite lists. With a compiler using strict evaluation, it needs to know the length of the list upfront since it needs to evaluate all of the values in the list when it is instantiated. With lazy evaluation, it only needs to evaluate the values of the list as the compiler needs them.

The third benefit is in how short-circuiting `if` statements are handled in the compiler. A compiler implemented using strict evaluation requires extra complexity for short-circuiting to occur. This is because the compiler needs to have special cases when it should not evaluation the right-hand-side. A compiler using lazy evaluation, however, has short-circuiting inherently built in and requires very little extra complexity in the compiler for it to occur. If the right-hand-side doesn't need to be evaluation, it just won't be.

Where are its downfalls?

Lazy evaluation isn't the perfect solution though. Strict evaluation can beat lazy evaluation when there isn't a lot of overhead. Since everything is evaluated at once with strict evaluation, once the program gets going, it is very fast. Lazy evaluation may get skip all that overhead of doing it all at once, but it is slower while it goes along since it needs to evaluating on the go.

Another downfall of lazy evaluation is it can add complexity to the compiler in order for it to determine whether or not it should evaluation an expression. This comes up more when dealing with object-orientated languages and the evaluation of objects. This is it is used more in functional languages over object-orientated languages.

Where would you use it?

Lazy evaluation is huge in functional programming languages like Haskell, but most computer science majors at UNC will never touched Haskell code in their computer science career, so why does it matter? Well, some popular languages have started using it too, like Python 3. Here are a couple of examples where you may have been using lazy evaluation without even knowing it.

One area you have probably used lazy evaluation before without even knowing it is with the `open()` function in Python. When using `open()`, the whole file isn't read at once, but instead, the lines of the file are fed through an iterator one by one. This is beneficial because we don't have to wait for huge files to be loaded in when we only need small bits of information from them. This can be seen by the following code example in the Python REPL.

First, the variable `file` is instantiated using the `open()` function with a simple text file. The second line, `file.readline()`, will print the first line of the file and then the iterator will consume it. After that, the third line, `file.read()`, will print the rest of the file, omitting the first line. It does not include the first line because it has already been consumed by the iterator and it is just running through the rest of the iterator, not actually reading the whole file.

```
>>> file = open("someFile.txt", "r")
>>> file.readline()
'This is Line 1.\n'
>>> file.read()
'This is Line 2.\nThis is Line 3.\n'
```

Another area where you have probably used lazy evaluation is when using lists with the `range` function in Python. This code shows when you create a list using the `range` expression, it is not actually evaluated into a list until a value is referenced from it. Once the value is referenced. It behaves just like a list.

```
>>> some_list = range(5)
>>> some_list
range(0, 5)
>>> some_list[3]
3
```

Why should you care?

Even though lazy evaluation is something that happens more behind the scenes, you should still be aware of it. For example, if you are working on a project that take advantage of lazy evaluation practices, like reading the first line in a huge text file, you could chose a language with it implemented in it's compiler.

Source Links

https://medium.com/background-thread/what-is-lazy-evaluation-programming-word-of-the-day-8a6f4410053f

https://www.seas.upenn.edu/~cis194/fall16/lectures/07-laziness.html

https://towardsdatascience.com/what-is-lazy-evaluation-in-python-9efb1d3bfed0