Lazy Evaluation

Lazy evaluation is the understanding that location of a computation's usefulness in a program is not necessarily the same as the location of the computation within the source code. Such a computation may be deferred until the value is read without affecting the nature of the sequential code. This contrasted with eager or strict evaluation which requires that computations be performed as soon as all relevant bindings are available.

Thunk!

Lazy evaluation is fundamentally a form of computation deferrment. When the evaluator reaches a compuation that is not immediately needed, rather than eagerly evaluating it and storing the result, it stores the information necessary to complete the computation behind a reference to the original value. This block of future computation is called a thunk. Consider this trivial example:

```
x := 3 + 5
...
print(x + 7)
```

The value of x is not actually needed until some future computation, in this case the addition operator in the print statement, attempts to read it. Therefore, `x` is assigned to a thunk representing the information necessary to compute `3 + 5`. When addition operator reads x, and finds the thunk, it will evaluate the computation, store the result in x, and then continue its computation. Note that in the case of:

```
x := 3 + 5
y := x + 7
...
print(y)
```

The computation of both x and y may be delayed until the print statement, resulting in a value of y (prior to evaluation of `thunk(thunk(3 + 5) + 7)` This exposes one of the potential weaknesses of lazy evaluation, in that a long chain of deferrable computations may produce a significant amount of computational state to carry forward. Additionally, deferring a computation requiring significant state necessitates carrying that state until the computation is required. This may be until the end of the program's execution, meaning the program has incurred that memory cost over a significant fraction of its runtime. In reality, the depth of deferrment trees are limited, preventing excessively nested thunks. The maximum state a thunk can carry forward may also be limited, requiring such large computations to be computed eagerly.

Advantages of Lazy Evaluation

Automatic Elision

By delaying computation, there is the potential to spare computation time without explicit optimization. Consider the long-running function `longf :: a -> a` in the following example

```
x := longf(10)
x := longf(11)
print(x)
```

The first call to `longf` may be entirely avoided as its thunk will be overwritten by the thunk for the second line, potentially saving significant computation. However, such elision requires the guarantee that `longf` has no side effects, that is, `longf` does nothing except return a value. Otherwise, the lazy and eager evaluations of this sequence are not identical.

A more interesting case of occurrs in function calls and control structures. Consider the function and an example call

```
f :: Int -> Int -> Int
f a b = a + a

f (3 + 5) (5 + 7)
```

Under eager evaluation, both arguments must be evaluated before beginning the executiton of the function, despite f only relying on its first argument. Under lazy evaluation, evaluation of the second argument would be indefinitely delayed until the variable fell out of scope. While this is underwhelming for addition, if evaluation of the parameters is intensive, the savings could be significant.

Motivating Example: Infinite Linked List (Haskell Syntax)

Computation of infinite lists is an exagerated form of the elision above. Consider the non-negative integers represented as a linked list, `[0, 1, 2, 3, ...]`, which is understood to consist of nested pairs in the form (0, (1, (2, (3, ...)))). This is written more succintly as

```
[0..]
```

Such a list contains infinite elements, and so must take infinite size in memory. However, an understanding of the linked list structure informs us that only the head of such a list can be operated on by value. Therefore, any tail elements of such a list may be elided by lazy evaluation. Rather than a sequence of infinite pairs, the list may be represented as

```
(0, thunk([1..]))
```

At each step, only the first element of the infinite list needs to be evaluated. Therefore, computation of the tail of the list is deferred until such time as its

elements are desired. The list's infinite nature will still prevent operation on the entire list, but operations on finite prefixes or infixes are now possible. The list will be evaluated up until the last element that is required, at each step finding the first element and deferring computation of the rest. Once this point is reached, the thunk representing the computation of the remaining tail simply falls out of scope.

This strategy of infinite list repesentation allows for elegant computation of sequences. The following code defines the Fibonacci sequence not as a generator function, but as an infinite sequence:

```
fibonacci :: [Int]
fibonacci = 0 : 1 : zipWith (+) fibonacci (tail fibonacci)
```

The `zipWith` operator pairs elements from two lists and joins them with the given function The `:` operator is the cons operator. This works by creating a list of the structure

```
0 : 1 : thunk(rest)
```

When asked for the third element, `zipWith` operates on

```
0 : 1 : thunk(rest) and
1 : thunk(rest)
```

It pairs up the first two elements and adds them producing the list `0 : 1 : 1 : thunk(rest)`. Repeating this:

```
0 : 1 : 1 : thunk(rest) and
1 : 1 : thunk(rest)
```

and now pairing the second element gives `0 : 1 : 1 : 2: thunk(rest)`. From this we can see the Fibonacci sequence emerging.

Disadvantages of Lazy Evaluation

As mentioned above, while lazy evaluation can be useful for avoiding unneeded computation, it may incurr large state penalties to deferr some computations. The poster child of this is the function `foldl`. `foldl` takes a list a reduces it to a single valuee

```
foldl (+) 0 [1, 2, 3] -- evaluates to 6
```

An eager, iterative interpretation would suggest the following sequence of operations

```
0 + 1 -> 1
1 + 2 -> 3
3 + 3 -> 6
```

At each step, only the result of the previous calculation is carried forward, a constant factor. When evaluated lazily, each step of that computation may be

deferred, resulting in a large thunk representing the eventual calculation

```
t(t(t(0 + 1) + 2) + 3)
```

Now the state required to be carried forward is linear in the length of the list. This can make lazy evaluation of long lists inefficient or impossible, a seemingly paradoxical result given lazy evaluation's role in enabling the use of infinite lists.

Sources

https://wiki.haskell.org/Non-strict_semantics
https://en.wikipedia.org/wiki/Lazy_evaluation