Adnar Lozano
CSE 320
Prof. Andrew Brinker

Lazy Semantics

Lazy semantics deals with a particular evaluation strategy. In other words, all programming languages use evaluation strategies to determine when to evaluate arguments of a function call, and lazy semantics (non-strict evaluation) is one of them. There is a whole family of evaluation strategies that all programming languages use as a design principle, but I will focus just on the two main types: Strict evaluation (*eager*), and non-strict evaluation (*lazy*). A more technical term for the former is *call-by-value*, and *call-by-need* (or *call-by-name*) for the latter. Let me point out that there's a technical distinction between *call-by-name* and *call-by-need*, they both seem to do the same thing but actually *call-by-need* is a memorized version of *call-by-name*. In one hand, *call-by-need* does not evaluate arguments of a function call unless they are needed in the evaluation process of the function body. On the other hand, in a *call-by-value* strategy, the argument expression is evaluated, and the resulting value is bound to the corresponding variable in the function. It does so by making a local copy of the value into a new memory location. Basically, in an eager evaluation strategy, a function's argument is evaluated before being passed to the function. Meanwhile, in a lazy evaluation strategy, a function's argument is not evaluated before the function is called, rather, it is substituted directly into the function body by way of *capture-free substitution*. According to Krishnamurthi, in a lazy evaluation, execution is deferred and evaluation of computations is suspended until they are needed. This means that a lazy evaluation will delay the computation of an expression until its value is needed, and avoid repeated evaluations to be computed (*sharing*). This is where lazy semantics can be beneficial over other evaluation strategies because it can reduce the running time of particular functions by an exponential factor due to sharing. Some of the benefits of lazy evaluation include: the power to define control flow as abstractions instead of primitives, the power to define potentially infinity data structures (*for (;;)*), and the power to increase performance by avoiding needless calculations.

Adnar Lozano
CSE 320
Prof. Andrew Brinker

If we take a look at Haskell, which is a purely functional programming language, it uses lazy semantics by default but it also offers strict evaluation as an option. Lazy evaluation in Haskell is done by *call-by-need*, which is a combination of *call-by-name* and sharing. As mentioned before, call-by-need is a memorized version of call-by-name; therefore, Haskell builds a temporary data structure called *Thunk*, which contains whatever values are needed to evaluate the expression, along with a pointer to the expression itself. This pointer is very important because it allows other function to reference this thunk and also because it's necessary for mutation and recursion. So when the resulting value is needed, Haskell calls the expression and then substitutes the *thunk* with the actual result of the computation for future reference. Clearly, a fantastic way to increase performance and avoid computing repeated expressions. Haskell relies heavily on the use of *thunks* in order to temporarily save their results so that they can be reference later in the future. These *thunks* are explicitly generated by wrapping an argument expression in an anonymous function with no parameters. By doing so, the expression is prevented from being evaluated until a receiving function calls the anonymous function. When a value is needed, the thunk is checked to see if it has been evaluated; if it has, Haskell will simply return the value; if it hasn't, Haskell's evaluation engine will run the expression and store its value for future reference. Haskell has many interesting features but the most defining ones are pure functions and lazy evaluation. Although they have multiple advantages, they also have multiple disadvantages such as making input and output difficult. The reason for this disadvantage is in Haskell's paradigm. By definition of a pure functional language, Haskell functions will always evaluate the same result given by the same argument, it does not depend on external I/O, meaning that it doesn't evaluate any side effects. In addition, I/O is difficult because a function like *putStrLn "Insert line here…"* cannot return a meaningful value that can be evaluated based on the function that it was called. The solution to work with algorithms and side effects in a functional language with lazy evaluations is monads. Haskell introduced this solution based on monads in order to achieve computational builders (from mathematical category theory) that can work with both stateful algorithms and side effects.

As mentioned earlier, lazy evaluation is one type of evaluation strategy, the other one is eager evaluation (*strict application*). There are more strategies to know but these two are the most common among most general programming languages. In the book "Programming Languages: Application and Interpretation" by Shriram Krishnamurthi, we have a nice example between lazy and eager applications. If we are to use the language Racket, and we define a square function to be:

```
(define (sq x) ( * x x) )
```

and invoke it as

```
(sq ( + 2 3) )
```

does it reduce to

```
(* 5 5)  ?
```

or to

```
(* (+ 2 3) (+ 2 3) )  ?
```

In an eager evaluation strategy, the expression would be reduced to

```
(sq (* 5 5) )
```

While in a lazy evaluation strategy, the expression would be reduced to

```
(sq (* (+ 2 3) (+ 2 3) ) )
```

In this particular example, we can see that eager evaluation calculates (2 + 3) before the square function is called, and passes the resulting value of 5 to the function. In a lazy evaluation, the expression is not evaluated at all; instead, the parameters (+ 2 3) are substituted in the square function. Once the function is called, then the parameters will be evaluated. This is the main difference between lazy and eager evaluation. In conclusion, an eager evaluation strategy will always evaluate the arguments of a function call first; while in a lazy evaluation strategy, the arguments of a function call will only be evaluated until they are needed, usually if they appear in the function's body.

Adnar Lozano
CSE 320
Prof. Andrew Brinker

References

Krishnamurthi, S. (n.d.). Programming Languages: Application and Interpretation.

WikiHaskell. (n.d.). *Lazy evaluation.* Retrieved from WikiHaskell: https://wiki.haskell.org/Lazy_evaluation

Wikipedia. (n.d.). *Lazy evaluation* . Retrieved from Wikipedia: https://en.wikipedia.org/wiki/Lazy_evaluation