Optimizations for Non-Strict Semantics

CIS561 Graduate Report

Zayd Hammoudeh

A language's evaluation semantics codify a set of rules that describe how expressions are evaluated. [2] Non-functional programming languages almost exclusively follow *strict*, also known as "*greedy*" or "*eager*," semantic rules where expressions are evaluated immediately after being bound to a variable. Therefore, the order of operations is implicitly defined by the structure of the code written by the programmer.

In contrast, many modern functional programming languages utilize *non-strict*, also known as "lazy" semantics. The basic intuition behind lazy evaluation is that an expression only gets evaluated at the time it is absolutely needed.

This paper focuses on non-strict semantics, specifically their efficient implementation. The remainder of this document is divided into two primary sections. Section 1 reviews basic background on non-strict semantics. This includes a brief introduction regarding the program analysis techniques that underpin lazy evaluation. Section 2 contrasts three schemes for representing a delayed evaluation. Section 3 contains brief concluding remarks.

1 Background

1.1 Motivating Example

Consider the simple program in Figure 1. A function "`f`" takes as input three parameters "`x`", "`y`", and "`z`". Observe that depending on the value of `x` either `y` or `z` will be used, but never both. [1]

```
f(x,y,z) = if (x == 0) then y else g(x, z)
```

Figure 1: Non-strict semantics motivating example

In *applicative-order semantics* semantics (more commonly known as *call-by-value*), a function evaluates *all* of its arguments before execution. Therefore, `x`, `y`, and `z` all are evaluated before invoking `f` despite only at most two of them being required at once.

In lazy evaluation, an argument is evaluated at the exact moment its value is needed. Under this paradigm, only argument `x` must necessarily be evaluated in a given invocation of `f`. In the case that `x`, `y`'s value is still not explicitly required despite being returned. Similarly, whether `z`'s evaluation can be further delayed will depend on function `g`.

1.2 Thunks

A *thunk* is encapsulates a single delayed computation.

1.3 Path Analysis

2 Representing Delayed Computations

2.1 Closure Mode (CL)

2.2 Cell Mode (C)

2.3 Optimized Cell Mode (CO)

3 Conclusions

References

[1] P. Henderson and J. H. Morris, Jr. A lazy evaluator. In *Proceedings of the 3rd ACM SIGACT-SIGPLAN Symposium on Principles on Programming Languages*, POPL '76, pages 95–103. ACM, 1976.

[2] H. J. Hoover. Functional programming: Definition and evaluation.