Evaluating untyped λ-expressions in Haskell

D'Ambrosi Denis
dambrosi.denis@spes.uniud.it
147681

September 30, 2023

Abstract

Contents

Introduction

1 Pure λ-Calculus

Pure λ-Calculus, developed by Alonzo Church in the 1930s, is a formal system that plays a pivotal role in the foundation of theoretical computer science. It is a minimalist, yet immensely powerful, mathematical framework for expressing computation using only anonymous functions.

The set of λ-terms Λ is the minimal set that can be constructed inductively according the following rules (that effectively define the syntax of this formalism):

1. If x is a variable, then $x \in \Lambda$
2. If x is a variable and $M \in \Lambda$, then $(\lambda x.M)$
3. If $M, N \in \Lambda$, then $(MN) \in \Lambda$

Rule number 2 expresses the possibility of creating *abstractions*. An abstraction is an anonymous function that takes as input a λ-term (in this case x) and returns a λ-term (not necessarily a variable, in this case M) as output. Multiple inputs functions can be constructed via a process called *currying*: since higher-order-functions are allowed within λ-Calculus, we can transform any n-ary function into an abstraction that takes only one input and returns another abstraction as output. The latter will, in turn, consume only one input and produce yet another abstraction, and so on, n times. Evaluating the last function will, eventually, produce the desired output.

Rule number 3, on the other hand, allows for the application abstractions to λ-terms: performing such operation will yield a new λ-term according two main reduction operations:

1. *α-conversion* aims at renaiming bound variables for avoiding name collisions: $(\lambda x.M) \to (\lambda y.M[x := y])$
2. *β-reduction* actually normalizes λ-terms by replacing bound variables in the body of the abstraction with the actual parameter: $((\lambda x.M)N) \to M[x := N]$

Before executing a β-reduction step, we want to ensure that the actual parameter substituted within an abstraction's body does not appear *free* in it. Formally, provided a λ-term t, the set of its *Free variables* $FV(t)$ can be computed recursively as follows:

- $FV(x) = x$ if x is a variable
- $FV((\lambda x.M)) = FV(M) \setminus x$
- $FV((MN)) = FV(M) \cup FV(N)$

If the input of the application appears free within the abstraction, then we must execute an process of α-conversion on the latter to avoid any undesired change of semantics due to name clashing. If we did not take this additional care while substituting, we may inadvertly compute incorrect terms while β-reducing.

This quick overview of the syntax and semanticsλ-calculus has laid the groundwork for introducing the actual implementations of λ-expressions evaluators of the upcoming sections. Before continuing, it is worth discussing briefly the reason why is formalism is still relevant, even after all these years. In 1936, Church proved that λ-calculus is Turing complete, meaning that it has the computational power equivalent to a universal Turing machine. This critical property establishes λ-calculus as a foundational formal system in the study of computation: this results implies that any computable function can be expressed and evaluated within the confines of λ-calculus. Such fact underpins its applications not only in theoretical computer science but also in practical domains, where it serves as the basis for functional programming languages and aids in the development of algorithms for various computational tasks.

2 Evaluating λ-expressions

A monad is a monoid in the cathegory of endofunctors

Unknown

2.1 Lexing λ-expressions

2.2 Parsing λ-expressions

2.3 Evaluating λ-terms: the algebraic approach

2.3.1 Alpha conversion method

2.3.2 De Bruijn Indexes method

2.4 Evaluating λ-terms: the computational approach

2.5 Performance metrics

3 Conclusions