Implementation of an Evaluator for the Untyped λ-calculus

Yuting Wang

Apr 26th, 2020

1 Introduction to the Untyped λ-calculus

Untyped λ-calculus was originally proposed by Alonzo Church as a formalism for rigorously describing mathematics. It has computational power equivalent to Turing machine. It describes computation through simple syntactic rules for converting λ-terms, which forms the foundation of functional programming.

1.1 The Syntax

The syntax of λ-terms is described below, where we use t (possibly with subscripts) to represent λ-terms:

$$t := x \mid \lambda x.t \mid (t_1\ t_2)$$

There are three ways to construct λ-terms:

- a **variable** (represented by x) is a term.
- if t is a term, then the expression $\lambda x.t$ is a term referred to as an **abstraction** that has the variable x as its binding variable (or binder) and t as its body or scope. Further, all occurrences of x in t that are *not in the scope of any abstraction in t with x as its binder* are considered bound by the abstraction.

  As an example, in $\lambda x.\lambda y.x\ y$, the only occurrence of the variable y is bound by the inner abstraction and the only occurrence of x is bound by outer abstraction.

  As another example, in $\lambda x.\lambda x.x\ x$, both occurrences of x is in the scope of the inner abstraction and hence bound by it. Because of that, the outer abstraction does not bind any variable.

  Note that the scope of a λ abstraction expands as far right as possible. As such, $\lambda x.(\lambda y.y)\ x$ is equivalent to $\lambda x.((\lambda y.y)\ x)$ instead of $(\lambda x.(\lambda y.y))\ x$.
- if t_1 and t_2 are terms, then $t_1\ t_2$ is a term referred to as an **application** that has t_1 as its function part and t_2 as its argument. Applications associate to the left. We omit the parentheses when the ordering of applications is obvious from the context.

1.2 Free and bound variables

We refer to x as a free variable in t if some of its occurrences in t is not bound by any abstraction. Letting $FVs(t)$ represent the set of free variables in t, $FVs(t)$ is defined as follows:

$$FVs(t) = \begin{cases} \{x\} & (t = x) \\ FVs(t_1) \cup FVs(t_2) & (t = t_1\ t_2) \\ FVs(t_1) - \{x\} & (t = \lambda x.t_1) \end{cases}$$

Therefore, x occurs (or does not occur) free in t if and only if $x \in FVs(t)$ (or $x \notin FVs(t)$.

1.3 Conversion rules

The following set of rules give computational interpretation to λ-terms:

- α-conversion. A fundamental idea underlying λ-calculus is that λ-terms should be considered equivalent under consistent renaming of λ-abstraction and the occurrences of variables bound by it. For example, $\lambda x.x\ x$ is equivalent to $\lambda y.y\ y$ and $\lambda z.z\ z$. This equivalence is captured concretely by α-conversion on λ-terms.

  Specifically, a term is α-converted into another term if the second can be obtained from the first by replacing a subterm of the form $\lambda x.t$ by $\lambda y.t'$ where y is a variable that does not occur in t and t' is the result of replacing the free occurrences of x in t by y. To understand the intricacy of the above rule, we need further explanation.

  - First, it requires y to be not in t:
    - to avoid accident capturing of other free variables in t. Consider the following counter-example, if we perform α-conversion on

      $$\lambda x.y\ x$$

      by replacing x with y, then the resulting term is

      $$\lambda y.y\ y$$

      whose computational meaning changes as the originally free variable y becomes bound after the conversion.
    - and to avoid accident capturing of y by abstractions in t. Consider the following counter-example, if we perform α-conversion on

      $$\lambda x.\lambda y.x\ y$$

      by replacing x with y, then the resulting term is

      $$\lambda y.\lambda y.y\ y$$

      whose computational meaning changes as the new occurrence of y representing the original x becomes bound by the inner abstraction instead of the outer one after the conversion.

– Second, it requires replacing only free occurrences of x in t, leaving bound occurrences of x untouched. Consider the following counter-example, if we perform α-conversion on the outer abstraction of

$$\lambda x.(\lambda x.x)\ x$$

by replacing x with y, then if we also replace the bound occurrence of x, we get

$$\lambda y.(\lambda x.y)\ y$$

which has a different computational meaning than that of the original term.

- β-reduction. By treating λ-abstractions as functions, β-reduction captures the idea of *function application*. A term of the form $((\lambda x.t_1)\ t_2)$ is referred to as a β-redex. A term u β-reduces to a term v if v can be obtained by replacing such a β-redex in u by the result of *substituting t_2 for the free occurrences of x in t_1 provided that the free variables in t_2 do not occur bound in t_1*. We denote such a substitution as $t_2[t_1/x]$. The term resulting from the substitution is called the reduct of the original β-redex.

  For example, in the following term

  $$\lambda x.(\lambda y.y\ x)\ x$$

  there is a β-redex $(\lambda y.y\ x)\ x$ which is reducible to $x\ x$ via substitution (Therefore, $x\ x$ is the reduct of $(\lambda y.y\ x)\ x$). As such, the whole term is β-reducible to $\lambda x.x\ x$.

  Note that there are two key points in the definition of β-reduction:

  – Only the free occurrences of x in t_1 are substituted. Bound variables should be left untouched. For example, in the following term

  $$(\lambda x.x\ (\lambda x.x))\ y$$

  there is only one β-redex. The result of β-reduction is

  $$y\ (\lambda x.x)$$

  where the bound occurrence of x is untouched.

  – The free variables in t_2 should not occur bound in t_1. This is to avoid accidental capturing of free variables of t_2 by abstractions in t_1 during substitution. Consider a counter example where $t_1 = \lambda x.(\lambda y.x)$ and $t_2 = y$, the β-redex $t_1\ t_2$ is:

  $$(\lambda x.(\lambda y.x))\ y$$

  It reduces to $\lambda y.y$ which is an identical abstraction whose computational interpretation deviates from the original abstraction because the free variable y accidentally becomes bound.

  Note that it is always possible to avoid unwanted capturing of free variables in β-reduction by using α-conversion. Continuing the previous example, we can first perform an α-conversion on the original term by replacing y with z to get the following equivalent term:

  $$(\lambda x.(\lambda z.x))\ y$$

  Now, we can safely perform β-reduction to get the correct result which is

  $$(\lambda z.y)$$

1.4 Evaluation Strategies

In functional programming, the essence of computation is to reduce terms through repeated application of α-conversion and β-reduction. When there are multiple β-redexes occurring in a term, we can choose different orders in which they are reduced. An *evaluation strategy* does exactly that. Depending on the adopted evaluation strategy, a term may be reduced to a normal form (meaning it can no longer be further reduced) or the reduction can fall into an infinite loop. Consider the following example:

$$(\lambda x.y)\ ((\lambda x.x\ x)\ (\lambda x.x\ x))$$

It is easy to see that the reduction of $((\lambda x.x\ x)\ (\lambda x.x\ x))$ never terminates (the readers should check it). Therefore, if the argument part in the application above is reduced first, the computation falls into an infinite loop. However, if we first reduce the whole term as a β-redex, the computation terminates immediately at the result y.

2 Exercises

We are going to implement an evaluator for the untyped λ-calculus in OCaml.[1] This project is broken down into several parts which we describe below.

2.1 Syntax of λ-calculus

First and foremost, we need to define the syntax of λ-calculus as an (algebraic) datatype. Specifically, we should define a type `term` as a OCaml variant type to represent λ-terms.[2] For simplicity, we should use strings to represent variables. Two variables are the same if they are represented using the same string.

Define the following function that computes the free variables in a term:

$$\texttt{fvars} : \texttt{term} \to \texttt{string list}.$$

2.2 α-conversion

Given the syntax of λ-calculus, define a function:

$$\texttt{alpha} : \texttt{string} \to \texttt{string list} \to \texttt{term} \to \texttt{string} * \texttt{term}$$

that performs α-conversion. It takes three arguments. The third argument is an abstraction to be converted. The second argument is a list of free variables that α-conversion should avoid. The first argument is the desired name (of the variable) to replace the binder for the abstraction. It returns a pair whose right part is the α-converted abstraction and whose left part is the actual binder of the resulting abstraction.

For example, given the abstraction $\lambda x.y\ x$, a list of free variables $[z; w]$ to avoid, and a hint name v, `alpha` should output $(v, \lambda v.y\ v)$.

Note that the hint name may conflict with the free or bound variables in the abstraction or those in the given list. In these cases, you should find a new name to avoid the conflict. For

[1] In fact, several evaluators with different evaluation strategies.

[2] A variant type is similar to inductive types in Coq. You can also consult the chapter named "Variants" in the "Real-World OCaml" book for details.

example, if the hint name is y instead of v, then you should replace it with a non-conflicting name such as y_1 and the final result is $(y_1, \lambda y_1.y\ y_1)$. Similarly, if the hint name is z, then the result of calling α should be $(z_1, \lambda z_1.y\ z_1)$. To generate non-conflicting names, you need to define the following function:

$$\texttt{fresh} : \texttt{string} \to \texttt{string list} \to \texttt{string}$$

The result of invoking `fresh` x l is a new name starting with x that does not occur in l.

2.3 β-reduction

This is broken down into the following steps:

- Define the following function to perform substitution for a free variable in a term:

  $$\texttt{subst} : \texttt{string} \to \texttt{term} \to \texttt{term} \to \texttt{term}$$

  The first argument is a free variable (say x) to be substituted. The second argument is the term to be substituted for x. The third argument is the term on which substitution is performed. For example, `subst` x y $(\lambda z.x)$ should return the result $(\lambda z.y)$.

- Define the following function:

  $$\texttt{beta} : \texttt{term} \to \texttt{term}$$

  that takes a β-redex as input and output its reduct. It should use `subst` to realize the reduction and perform necessary α-conversion to avoid capturing of free variables. There are many ways α-conversion can be applied here to avoid such capturing. Implement one that works first and optimize the way α-conversion is performed later.

2.4 Evaluation strategies

We implement the following commonly used evaluation strategies.

The call-by-value strategy The first evaluation strategies is *call-by-value*. In this evaluation strategy:

- Evaluation terminates if the term is a variable or an abstraction.
- Otherwise, the term is an application. Its function part is evaluated first and its argument part later. If the evaluation terminates and results in a β-redex, the β-redex is reduced and its reduct is recursively evaluated. Otherwise, the evaluation terminates.

Write a function

$$\texttt{cbv} : \texttt{term} \to \texttt{term}$$

that performs the call-by-value evaluation strategy.

The call-by-name strategy The second evaluation strategy is *call-by-name*. It works as follows:

- Evaluation terminates if the term is a variable or an abstraction.

- Otherwise, the term is an application. Its function part is evaluated. If the evaluation terminates and results in an abstraction, then the original term is a β-redex. This β-redex is reduced and its reduct is recursively evaluated. Otherwise, the evaluation terminates.

Note that the difference between call-by-value and call-by-name is that the latter skips the evaluation of the argument part of a β-redex. Write a function

$$\texttt{cbn} : \texttt{term} \to \texttt{term}$$

that performs the call-by-name evaluation strategy.

The normal-order strategy The normal-order evaluation strategy repeatedly performs the following steps:

- Finds the left-most β-redex in the term. The computation terminates if there is no β-redex found. Otherwise, reduce the found β-redex to get a new term. Repeat this step on the new term.

Note that this strategy is different from the previous ones in the sense that it looks into the body of λ-abstraction for evaluation

For instance, evaluation of the following term in normal-order

$$\lambda y.(\lambda x.y)\ ((\lambda x.x\ x)\ (\lambda x.x\ x))$$

results in $\lambda y.y$. However, if we switch the order of two terms in the outermost application in the above expression and evaluate the result—i.e., if we evaluate the following term

$$\lambda y.((\lambda x.x\ x)\ (\lambda x.x\ x))\ (\lambda x.y)$$

—in normal-order, then the evaluation falls into an infinite loop because the left-most β-redex reduces to itself.

2.5 Experiment: Church Encoding

Now you have a working λ-calculus evaluator! Test it out by running some examples.

For instance, you can uses it to do arithmetic using Church encoding of natural numbers (also known as *Church numerals*): `https://en.wikipedia.org/wiki/Church_encoding`. Define the following arithmetic operations on Church encoded natural numbers and apply them to some examples. Note that you need to use the normal-order evaluation strategy for this exercise.

- Successor: $succ\ n = n + 1$;
- Addition: $add\ m\ n = m + n$;
- Multiplication: $mult\ m\ n = m * n$;
- Exponentiation: $exp\ m\ n = m^n$;
- Predecessor: $pred\ n = n - 1$ (Note that $n - 1 = 0$ if n is 0);

- Minus: $minus\ m\ n = m - n$.

You can also implement basic conditional expressions using Church encoding. For instance, you can implement the following expression as a λ-term:

$$IfZero\ x\ y\ z = \begin{cases} y & x = 0 \\ z & x \neq 0 \end{cases}$$

2.6 Experiment: Fixed-point Combinators

It is possible to realize fixed-point (recursive) functions through the use of fixed-point combinators in λ-calculus. A fixed-point combinator *fix* is a term such that $fix\ F = F\ (fix\ F)$. Using such a combinator, we can define recursive functions similar to those defined using `let rec` in OCaml. Consider the following OCaml function that computes factorials:

```
let rec fac n =
  if n = 0 then 1
  else n * fac (n - 1)
```

By replacing the language constructs in this OCaml program with those defined in λ-calculus, we extract the following abstraction:

$$F = \lambda f.\lambda n.IfZero\ n\ 1\ (mult\ n\ (f\ (pred\ n)))$$

By applying *fix* to F and defining $fac = fix\ F$, we have:

$$\begin{aligned} fac\ n &= fix\ F\ n = F\ (fix\ F)\ n = F\ fac\ n \\ &= (IfZero\ n\ 1\ (mult\ n\ (fac\ (pred\ n)))) \end{aligned}$$

That is, $fac = fix\ F$ exactly defines the factorial function.

We shall investigate one of the most famous fixed-point combinators: the Y-combinator discovered first by Haskell Curry.

- Y-combinator is defined as follows:

$$Y = \lambda f.(\lambda x.f\ (x\ x))\ (\lambda x.f\ (x\ x))$$

  Do the following exercises. Use your evaluator to experiment with your implementation.

  - Show that $YF = F\ (Y\ F)$, i.e., Y is a fixed-point combinator.[3]
  - Test out the call-by-value, call-by-name and normal-order evaluation strategies to the applications of Y to functions. Which ones of these evaluation strategies terminate and why?
  - Use Y to define addition on Church numerals as a recursive function. You shall only use *succ* and *IfZero* in your recursive call. We call it add' to distinguished it from the previously defined *add* function.

[3] In fact, this is not exactly true: here the argument $(Y\ F)$ needs to be further β-reduced for the equation to hold.

- Use Y to define multiplication on Church numerals as a recursive function. You shall only use add' and *IfZero* in your recursive call. We call it $mult'$
- Use Y to define exponentiation on Church numerals as a recursive function. You shall only use $mult'$ and *IfZero* in your recursive call.
- Use Y to define minus on Church numerals as a recursive function. You shall only use *pred* and *IfZero* in your recursive call. We call it $minus'$
- Use the above functions and Y to implement factorial on Church numerals.

- A problem with Y-combinator is that it does not terminate on call-by-value evaluation strategy. A slight alternative of it called the Z-combinator fixes the problem. It is defined as follows:

$$Z = \lambda f.(\lambda x.f\ (\lambda v.x\ x\ v))\ (\lambda x.f\ (\lambda v.x\ x\ v))$$

  Do the following exercises. Use your evaluator to experiment with your implementation.

  - Repeat the above exercises on Y using Z.
  - Observe why call-by-value evaluation strategy works with Z. Summarize the differences between Y and Z.